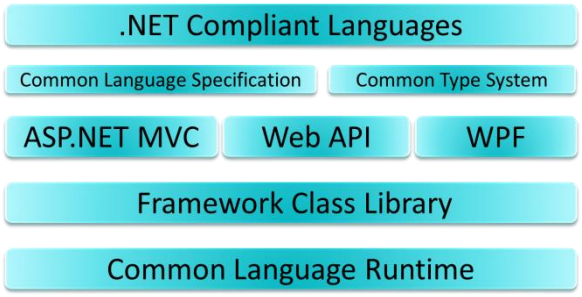
.NET Course

Contents

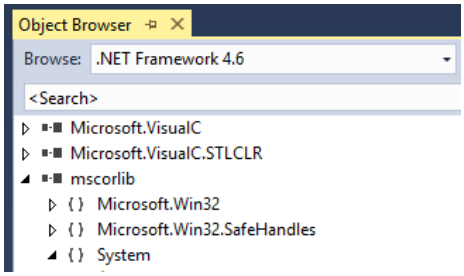
Introduction to the .NET framework

Programs written for the .NET Framework execute in the Common Language Runtime (CLR), a runtime environment that provides services such as security, memory management, and exception handling.

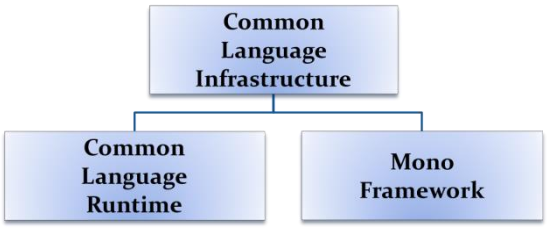


The .NET framework includes a large class library, the Framework Class Library, which provides classes covering database connectivity, web applications and desktop applications.

Start Visual Studio and open the object browser



Common Language Infrastructure



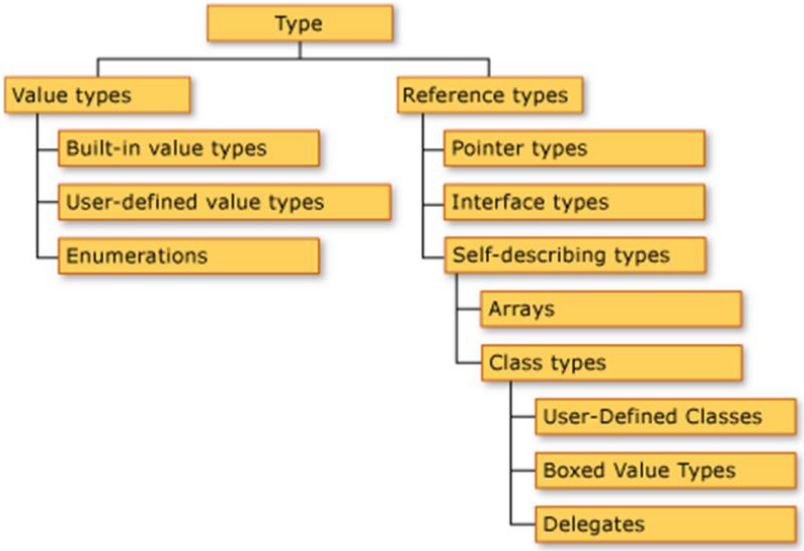
The Common Language Infrastructure (CLI) defines the Virtual Execution System and the executable code, Common Intermediate Language, which runs in it. The Common Language Runtime is Microsoft's implementation of the CLI.

Mono

Common Intermediate Language

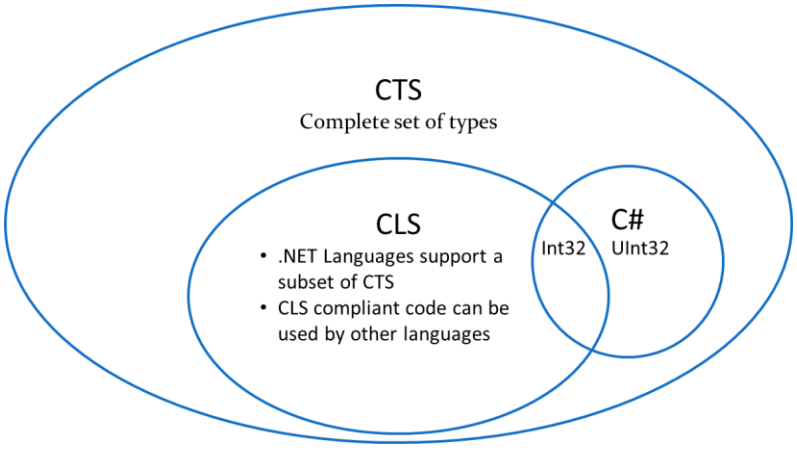
CIL is a stack-based object-oriented assembly language. At compile time, C# source code is translated into CIL. At runtime, the CLR's Just-In-Time (JIT) compiler translates CIL into native code, which is then executed by the computer's processor.

Common Type System



The common type system defines how types are declared, used, and managed in the common language runtime.

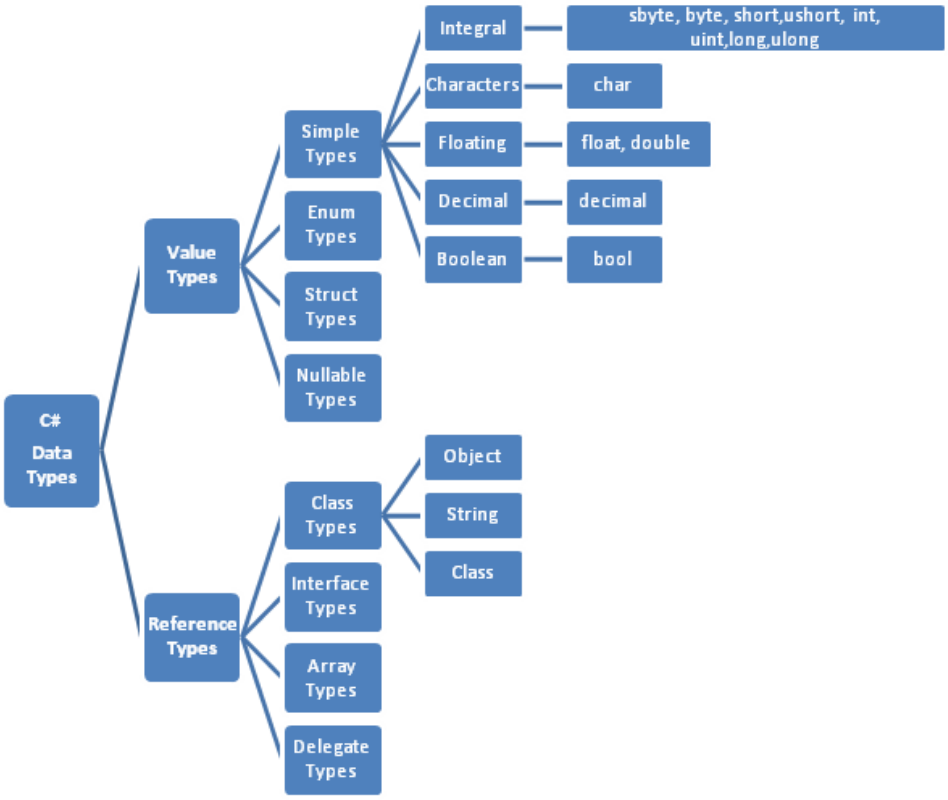
Common Language Specification



To fully interact with other objects written in any language, objects must expose to callers only those features that are common to all languages. This common set of features is defined by the Common Language Specification (CLS). For example UInt32, a 32 bit unsigned integer, is in the CTS and available to C#, but is not in the CLS.

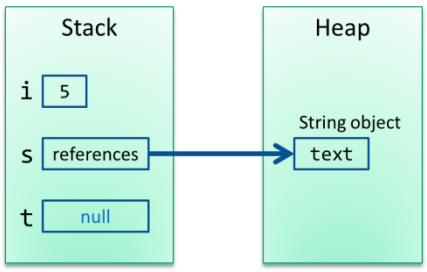
Core language features

Data types

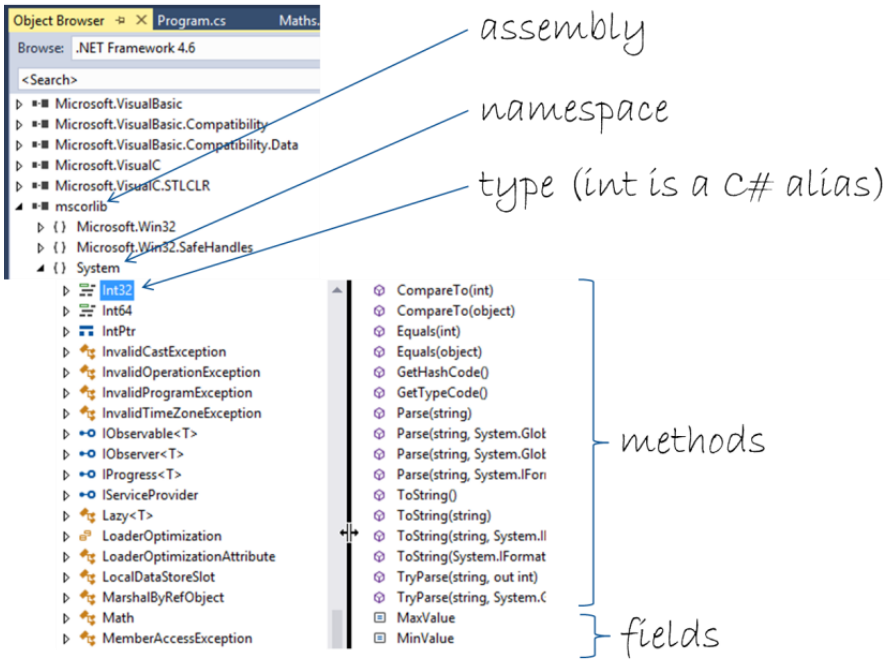


C# is a strongly-typed language. Before a value can be stored in a variable, the type of the variable must be specified.

```
int i = 5;
```



Value types store their contents in an area of memory called the stack. When the variable goes out of scope, because the method in which it was defined has finished executing, the value is discarded from the stack. Reference types, such as strings, are allocated in an area of memory called the heap. When the variable referencing an object becomes out of scope, the object becomes eligible for garbage collection.



Int32 is an integer type in the System namespace. It has a C# alias, int

Numerical types can be converted to wider types or cast to narrower types.

```
int a = 5;
double b = a;

double c = 5;
int d = (int)c;
```

Operators

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right
Multiplicative	* / %	Left
Additive	+ -	Left
Shift	<< >>	Left
Relational	< <= > >=	Left
Equality	== !=	Left
Bitwise AND	&	Left
Bitwise XOR	^	Left
Bitwise OR	|	Left
Logical AND	&&	Left
Logical OR	||	Left
Conditional	?:	Right
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right

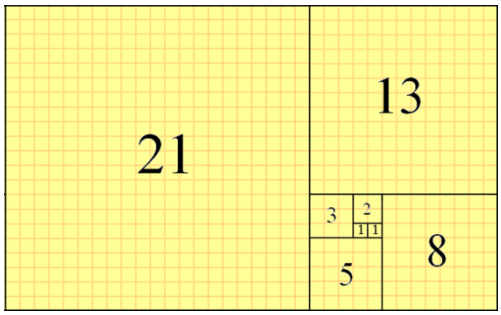
Decision structures

Refer to Visual Studio Console application examples

Repetition

Refer to Visual Studio Console application for examples of for and while loops and switch statements

A tiling with squares whose side lengths are successive Fibonacci numbers



The sequence Fn of Fibonacci numbers is defined by

$F_n = F_{n-1} + F_{n-2}$

with seed values

$F_1 = 1, F_2 = 1$

Using a loop, write the Fibonacci numbers below 100 to the console

Classes and objects

Static methods

Use the static modifier to declare a static member, which belongs to the type itself rather than to a specific object.

```
public class Math
{
   public static double Sqrt(double d)
   {
        return 0;
   }
}
```

Write a Maths.Factorial method

$$C_{(n,r)} = \frac{n!}{r!\,(n-r)!}$$

$$P_{(n,r)} = \frac{n!}{(n-r)!}$$

n = set size: the total number of items in the sample

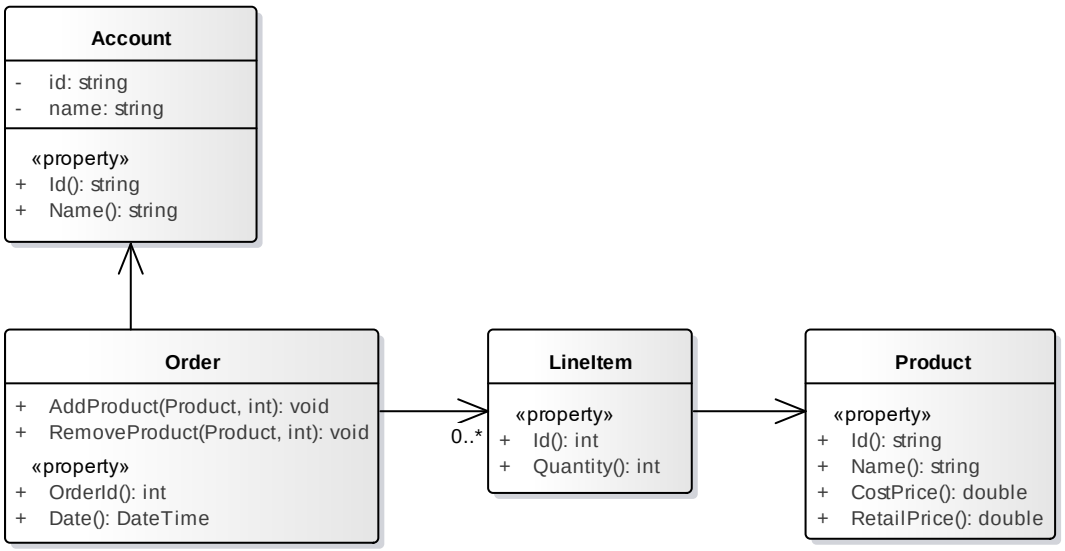
r = subset size: the number of items to be selected from the sample

This is an example or recursion. The method call stack can be viewed by opening Debug > Windows > Call Stack

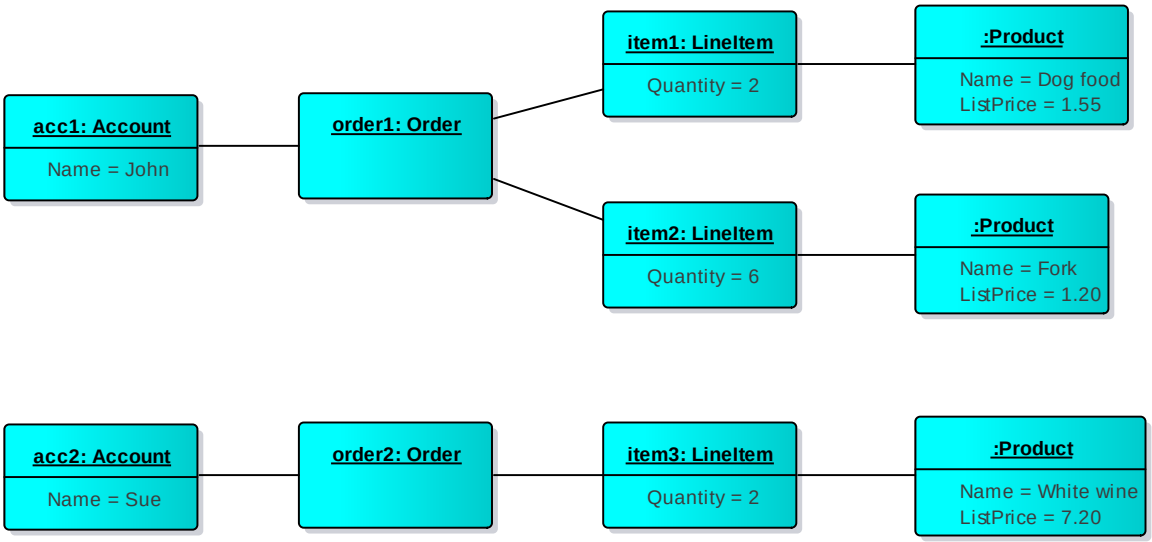
```
public class Maths
{
    public static string ToRoman(int number)
    {
        if (number >= 9) return string.Empty;
        if (number >= 5) return "V" + ToRoman(number - 5);
        if (number >= 4) return "IV" + ToRoman(number - 4);
        if (number >= 1) return "I" + ToRoman(number - 1);
        return string.Empty;
    }
}
```

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1,000

UML class diagram

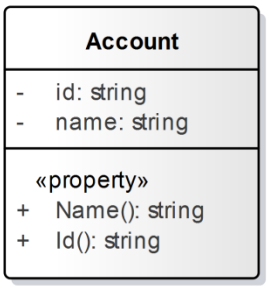


UML object diagram



Fields and properties

Account class



variables - state

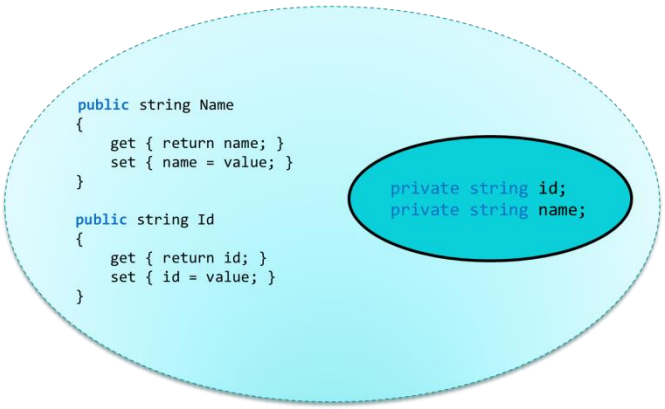
methods and properties - behaviour

```
public class Account
{
    private string id;
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public string Id
    {
        get { return id; }
        set { id = value; }
    }
}
```

state

behaviour

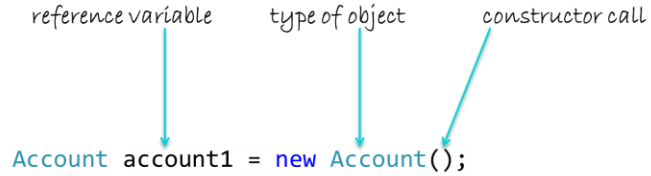
edit > refactor > encapsulate field (ctrl R ctrl E)



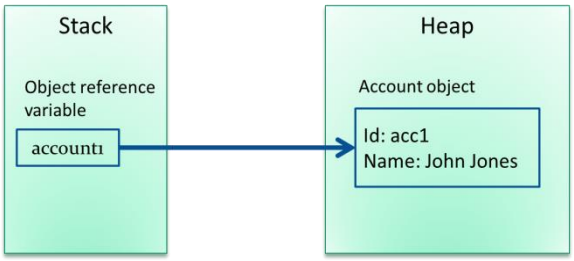
Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state.

Accessibility	
public	Any referencing assembly
internal (the default)	Same assembly
private	Same class

Instantiating a class




```
Account account1 = new Account();
account1.Id = "acc1";
account1.Name = "John Jones";
```



Build the Account class

- Add a new class library project
- Add a class named Account
- See Class View
- Make the class public
  ***public** class Account*
- Add the two instance variables
  private string id;
  private string name;
- Generate the properties
  - Select the two fields
  - Edit menu | Refactor | Encapsulate field
- Open the Class View, right click the class and select View Class Diagram

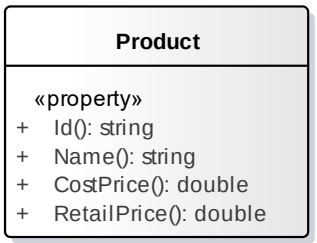
Build an instance of a class

- Add a reference to the class library project from the console application project
- Build an Account object and set its properties

```
using ClassLibrary1;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Account account1 = new Account();
            account1.Id = "acc1";
            account1.Name = "John Jones";
            Console.WriteLine(account1.Id + " " + account1.Name);
```

Auto-implemented properties

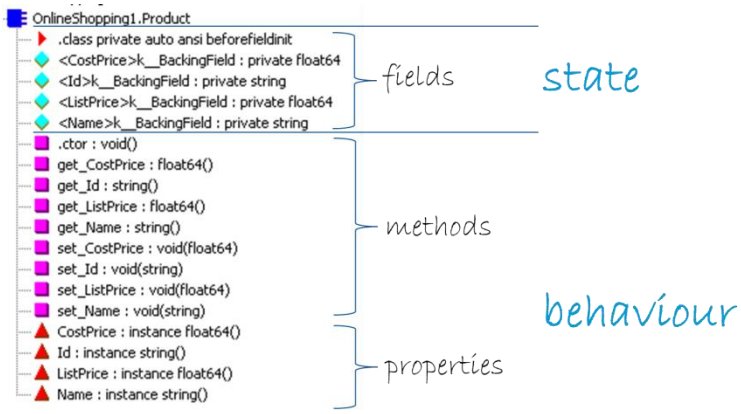
Product class



These property declarations are more concise and are appropriate when no additional logic required. The compiler creates a private, anonymous backing field.

```
namespace OnlineShopping
{
    public class Product
    {
        public string Id { get; set; }

        public string Name { get; set; }

        public double CostPrice { get; set; }

        public double RetailPrice { get; set; }
    }
}
```

prop Tab Tab



The backing field can be viewed by opening the compiled assembly with the intermediate language disassembler (ildasm)

Constructors

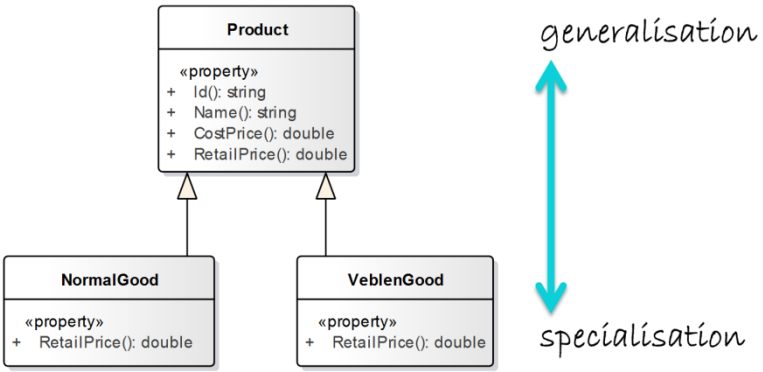
```
public class Product                    ctor Tab Tab
{
    public Product()
    {
    }
    public string Id { get; set; }
    public string Name { get; set; }
    public double CostPrice { get; set; }
    public double RetailPrice { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
```

Constructors are methods that are called when a class is instantiated. The empty brackets following new Product() indicates a call to the no-argument constructor. If there are no constructors in the source code, the compiler will generate a no-argument constructor.

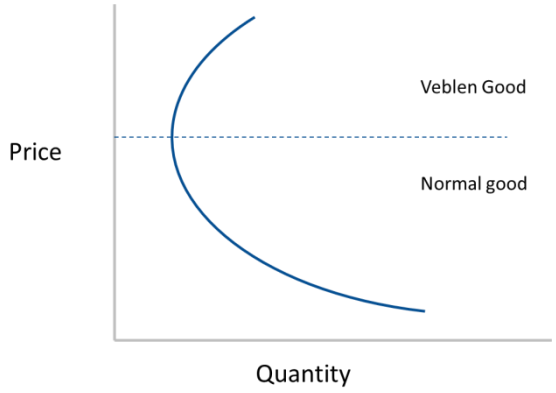
```
public class Product
{
    public Product()
    {
    }
    public Product(string id, string name,
                   double costPrice, double listPrice)
    {
        Id = id;
        Name = name;
        CostPrice = costPrice;
        RetailPrice = listPrice;
    }

class Program
{
    static void Main(string[] args)
    {
        Product product1 = new Product();
        Product product2 = new Product ("p2", "Fork", 0.4, 1.2 );
```

Overloading is the term used to describe multiple methods with the same name but different numbers of types of argument in a class. The above class contains overloaded constructors.

Inheritance



Inheritance enables creating new classes that reuse, extend, and modify the behaviour that is defined in other classes. A derived class is a specialization of the base class.



Veblen goods are types of luxury goods which are in demand because of the high prices asked for them. A reduction in price would reduce their cachet.

Overriding

```
public class Product
{
    public string Id { get; set; }

    public string Name { get; set; }

    public double CostPrice { get; set; }

    public virtual double RetailPrice { get; set; }
}
```

When a base class declares a method as virtual, a derived class can override the method with its own implementation.

The RetailPrice property is overridden in the derived classes of Product. The base class follows the colon in the following code.

```
public class NormalGood : Product
{
    public override double RetailPrice
    {
        get
        {
            return CostPrice * 2.5;
        }
    }
}
```

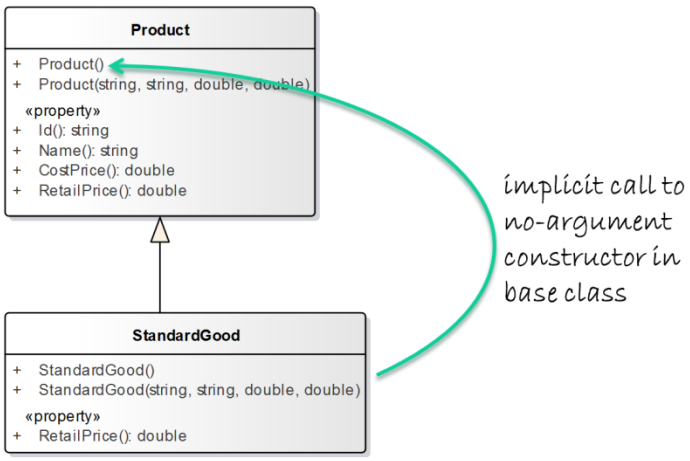
```
public class VeblenGood : Product
{
    public override double RetailPrice
    {
        get
        {
            return CostPrice * 5;
        }
    }
}
```

Accessibility	
public	Any referencing assembly
protected	Same class or derived class
internal (the default)	Same assembly
protected internal	Same assembly or derived class
private	Same class

Constructors in a hierarchy

```
public class NormalGood : Product
{
    public NormalGood()
    {

    }
    public NormalGood(string id, string name,
                      double costPrice, double listPrice)
    {

    }
```

Constructors aren't inherited, so the following derived class includes overloaded constructors.



When a derived class is instantiated, the base class is also instantiated. By default, the no-argument constructor in the base class will be called.

The base keyword is used to invoke a specific constructor

```
public class NormalGood : Product
{
    public NormalGood()
    {

    }
    public NormalGood(string id, string name, double costPrice)
                          : base (id, name, costPrice, 0)
    {

    }
```

Arrays and Collections

Arrays

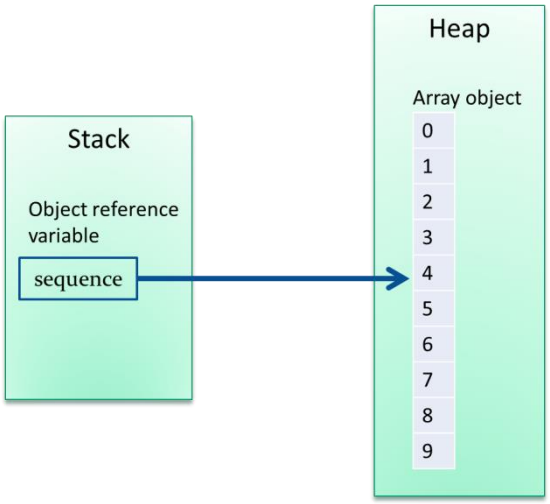
Fibonacci

Arrays are objects; the expression

int[] sequence = new int[10];

`sequence[9] = 34;`

Builds an array of type int containing 10 elements, starting at element 0. The elements in an int array are initialised as 0; the elements in an array of a reference type, such as a string, are initialised as null (a null reference).



Add a method to the Maths class that takes an int argument and returns an array containing the first n Fibonacci numbers, where n is the argument to the method

```
namespace ConsoleApplication.Tests
{
    public class MathsTest
    {
        [Fact]
        [Trait("Category", "Unit Test - State")]
        public void Fibonacci_NumberOfElements_ShouldReturnCorrectSequence()
        {
            int[] expected = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            int[] actual = Maths.Fibonacci(10);
            Assert.Equal(expected, actual);
        }
```

Object reference conversion

An object can be assigned to a base class variable, known as object reference conversion. The following expression uses object initialiser syntax to set the properties of the NormalGood object.

```
Product product1 = new NormalGood { Id = "p1", Name = "Dog Dinner", CostPrice = 0.4 };
```

Polymorphism

Polymorphism can be demonstrated by calling the RetailPrice property of Product objects. Rather than calling the method that is defined by the variable's type, the overriding method in the NormalGood or VeblenGood class is invoked.

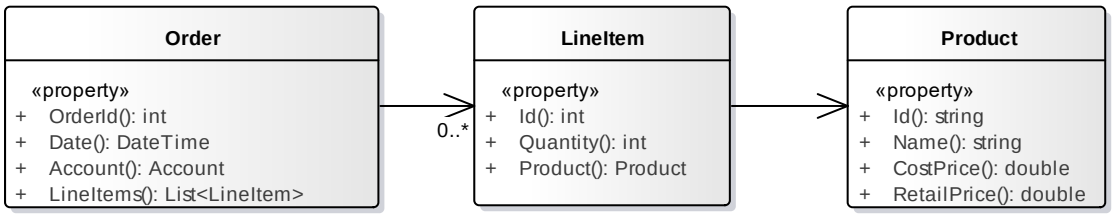
```
namespace ConsoleApplication.Examples
{
    public class Polymorphism
    {
        public static void Main()
        {
              Product product1 = new NormalGood { Id = "p1",
                                  Name = "Dog Dinner", CostPrice = 0.4 };
              Product product2 = new NormalGood { Id = "p2",
                                  Name = "Fork", CostPrice = 0.4 };
              Product product3 = new VeblenGood { Id = "p3",
                                  Name = "Krug Champagne", CostPrice = 25 };
              Product product4 = new VeblenGood { Id = "p4",
                                  Name = "Rolex watch", CostPrice = 700 };
              Product[] products = new Product[4];
              products[0] = product1;
              products[1] = product2;
              products[2] = product3;
              products[3] = product4;
              foreach (Product product in products)
              {
                  Console.WriteLine(product.RetailPrice);
              }
```

Tweet

- Define a class named Tweet in a package named twitter that includes two properties, Username and Text
- Add overloaded constructors
- Build an array of five Tweet objects
    - Uneasy lies the head that wears a crown.
    - The fool doth think he is wise, but the wise man knows himself to be a fool.
    - They make a desert and call it peace.
    - What hath God wrought
    - But at my back I always hear time's winged chariot hurrying near
- Iterate through the array, printing the text of each tweet
- Modify the Text property so that tweets above 140 characters are truncated

Collections

Order and LineItem classes



A LineItem has an associated Product and an Order is associated with multiple LineItems. The LineItems property is of type List<LineItem>. A List is a collection whose elements are accessible by a zero based index, like an array. The <LineItem> syntax declares the type of object that can be stored in the List.

```
public class Order
{
    public int OrderId { get; set; }
    public DateTime Date { get; set; } = DateTime.Now;
    public Account Account { get; set; }
    public List<LineItem> LineItems { get; set; } = new List<LineItem>();
}
```

Declaring the properties of a class type won't instantiate them. Either build the objects in the constructor or use a collection initializer.

Enum

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

Add a property of type OrderStatus to the Order class

```
    public OrderStatus OrderStatus { get; set; } = OrderStatus.NotPlaced;
```

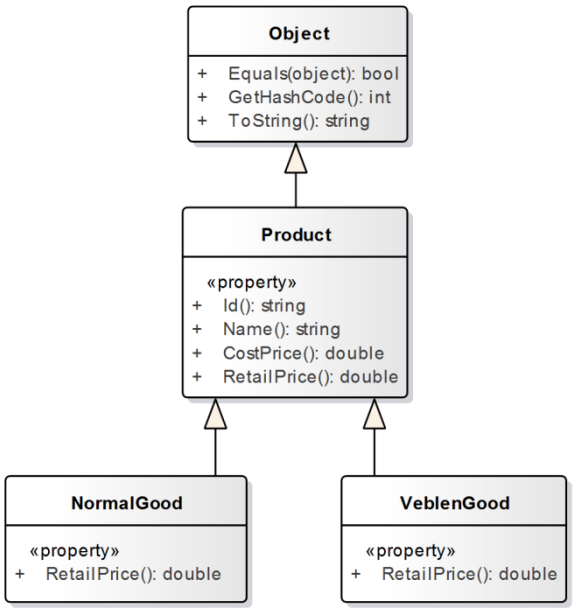
And then generate the following enum

```
public enum OrderStatus
{
    NotPlaced,
    New,
    Packed,
    Dispatched,
    Delivered
}
```

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. An enum constant can be cast to an integral type.

```
int x = (int)OrderStatus.Dispatched;
```

Overriding Equals



The object class is at the top of the class hierarchy.

Classes can override the Equals method to enable object equality to be determined by value rather than by reference. For example, Product objects could be deemed to be equal if they have the same Id.

Classes that override the Equals method should also override the GetHashCode method. A hash code is a numeric value that is used to insert and identify an object in a hash-based collection. Equal objects should have equal hash codes, but unequal objects are not required to have unequal hash codes.

Start with some unit tests describing the expected behaviour

```
namespace Core.UnitTests.Entity
{
    public class ProductTest
    {
        [Fact]
        public void ProductsWithSameIdShouldBeEqual()
        {
            //arrange
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            //act
            bool equal = product1.Equals(product2);
            //assert
            Assert.True(equal);
        }
```

And then add override the Equals method in the Product class

```
public class Product
{
    public override bool Equals(object obj)
    {
        return obj is Product ? (obj as Product).Id == Id : false;
    }
```

Interfaces

Interfaces form a contract between the class and its users. If a class implements an interface, all methods defined by that interface must be implemented by the class.

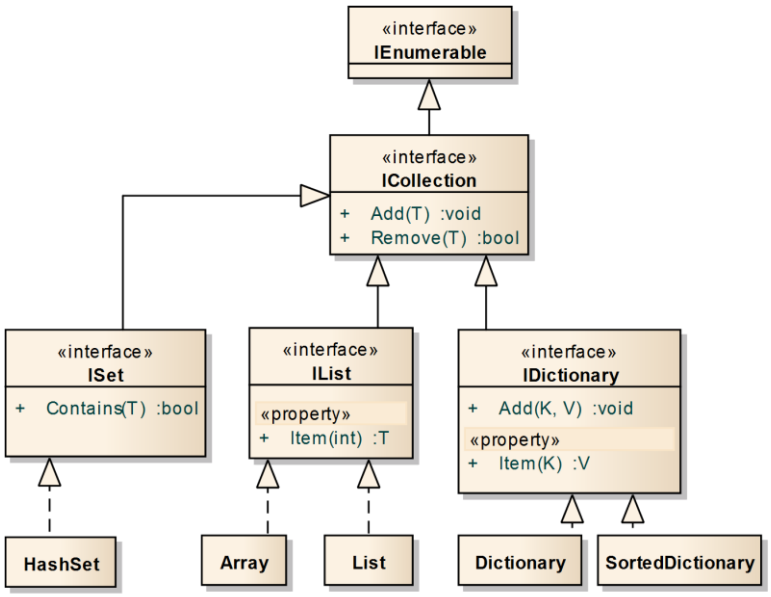
An interface contains only the signatures of methods and properties. For example, the ICollection interface includes Add and Remove methods.

```
public interface ICollection<T>
{
    void Add(T item);
    bool Remove(T item);
}
```

A class that implements the interface must implement these methods

```
class HashSet<T> : ConsoleApplication.Examples.ICollection<T>
{
    public void Add(T item)
    {
        throw new NotImplementedException();
    }

    public bool Remove(T item)
    {
        throw new NotImplementedException();
    }
}
```

Collections framework

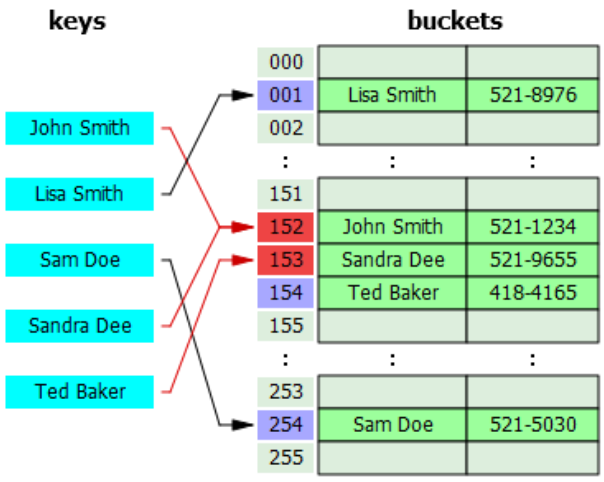


IEnumerable objects can be iterated through with a foreach loop. An ICollection is a very general collection. An ISet contains unique elements in no particular order. HashSet is an implementation of the ISet interface.

```
namespace Core.UnitTests.Entity
{
    public class ProductTest
    {
        [Fact]
        public void ProductsShouldWorkCorrectlyWithList()
        {
            List<Product> products = new List<Product>();
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            products.Add(product1);
            Assert.True(products.Contains(product2));
        }

        [Fact]
        public void ProductsShouldWorkCorrectlyWithHashSet()
        {
            HashSet<Product> products = new HashSet<Product>();
            Product product1 = new Product { Id = "1" };
            Product product2 = new Product { Id = "1" };
            products.Add(product1);
            Assert.True(products.Contains(product2));
        }
    }
}
```

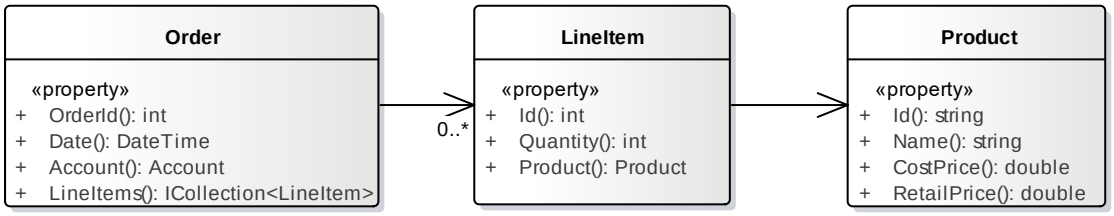
GetHashCode

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash collisions—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way. C# uses a strategy called open addressing, in which all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.



Property with Interface type

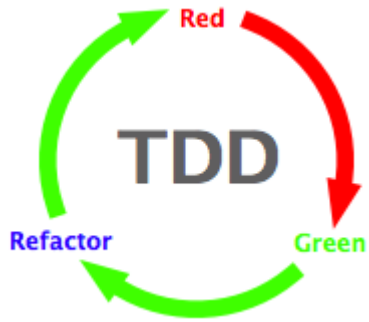
Changing the LineItems property from List to ICollection enables more flexibility in the implementation of the Order class.




```
public class Order
{
    public int OrderId { get; set; }
    public DateTime Date { get; set; } = DateTime.Now;
    public Account Account { get; set; }
    public ICollection<LineItem> LineItems { get; set; } = new List<LineItem>();
}
```

Unit Testing

Test-driven development



TDD is a software development process that relies on the repetition of a very short development cycle

1. Write an (initially failing) automated test case that defines a desired improvement or new function
2. Produce the minimum amount of code to pass that test
3. Refactor the new code to acceptable standards

Benefits

- Test cases force the developer to consider how functionality is used by clients, focussing on the interface before the implementation
- Helps to catch defects early in the development cycle
- Requires developers to think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces.
- Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Refactoring

- KIS (keep it simple) writing the smallest amount of code to make the test pass leads to simple solutions.
- YAGNI "You aren't going to need it" - avoid unnecessary methods.
- DRY (don't repeat yourself)
- SRP (single responsibility principle)

Types of test

Unit tests

- Single classes
- ensure high quality code
- replace real collaborators with mock objects

Integration tests

- the code under test is not isolated
- run more slowly than unit tests
- verify that modules are cooperating effectively
- Integration testing is similar to unit testing in that tests invoke methods of application classes in a unit testing framework. However, **integration tests do not use mock objects to substitute implementations for service dependencies**. Instead, integration tests rely on the application's services and components. The goal of integration tests is to exercise the functionality of the application in its normal run-time environment.

Acceptance tests

- multiple steps that represent realistic usage scenarios of the application as a whole.
- scope includes usability, functional correctness, and performance.

Phases when writing a test

1. Arrange – create objects
2. Act – execute methods to be tested
3. Assert – verify results

XUnit

Visual Studio includes the MSTest framework, but third party frameworks can be integrated with the development environment. xUnit.net is an open source unit testing tool for the .NET framework, written by the original author of NUnit. See https://xunit.github.io/

A widely used naming convention for unit tests is to concatenate
[the name of the tested method]_[expected input / tested state]_[expected behaviour].

```
public class MathsTest
{
    [Fact]
    public void Factorial_ShouldReturnCorrectValues()
    {
        //arrange
        //act
        //assert
    }
```

To set up xUnit, install xunit and xunit.runner.visualstudio from NuGet. To prevent the test explorer prefixing the package qualified class name to the method name, add a file xunit.runner.json containing

```
{
  "methodDisplay":"method"
}
```

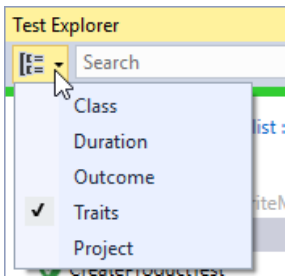
And select “Copy to output directory if newer” in the properties window for the file

Facts and 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.

Traits

To enable categorisation of tests in the Test Explorer, methods can be prefixed with the Trait attribute.



Try it out

Parameters

```
namespace ConsoleApplication.Tests
{
    public class MathsTest
    {
        [Fact]
        [Trait("Category", "Unit Test - State")]
        public void Factorial_ShouldReturnCorrectValues()
        {
            //act
            double actual = Maths.Factorial(5);
            //assert
            Assert.Equal(120, actual);
        }

        [Theory]
        [InlineData(0, 1)]
        [InlineData(1, 1)]
        [InlineData(2, 2)]
        [InlineData(3, 6)]
        [InlineData(6, 720)]
        [Trait("Category", "Unit Test - State")]
        public void Factorial_ParameterizedTest(int n, double expected)
        {
            //act
            double actual = Maths.Factorial(n);
            //assert
            Assert.Equal(expected, actual);
        }
```

Constructor

```
namespace Core.UnitTests.Entity
{
    public class ProductTest
    {
        [Fact]
        [Trait("Category", "Unit Test - State")]
        public void Constructor_WhenPassedParameters_ShouldSetProperties()
        {
            Product product = new Product("p1", "Dog's Dinner", 1.20, 2.50);
            Assert.Equal("p1", product.Id);
            Assert.Equal("Dog's Dinner", product.Name);
            Assert.Equal(1.20, product.CostPrice);
            Assert.Equal(2.50, product.RetailPrice);
        }
```

Interrogating Collections

LINQ Query syntax

Language-Integrated Query (LINQ) enables data from a variety of sources, such as a collection or a database, to be queried using a standard syntax. All LINQ query operations consist of three distinct actions:

1. Obtain the data source. This must be an object which implements the IEnumerable<T> interface
2. Create the query. This specifies what information to retrieve from the data source. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before it is returned. The query expression contains three clauses: from, where and select.
    a. The from clause specifies the data source
    b. The where clause applies the filter,
    c. The select clause specifies the type of the returned elements.
3. Execute the query. The query variable itself only stores the query commands. The actual execution of the query is deferred until iterating over the query variable in a foreach statement.

- Data source

```
int[] numbers =  { 0, 1, 2, 3, 4, 5, 6 };
```

- Query creation

```
IEnumerable<int> numQuery =
       from num in numbers
       where (num % 2) == 0
       select num;
```

- Query execution

```
foreach (int num in numQuery)
{
    Console.WriteLine(num);
}
```

Similarly, to interrogate a collection of objects

- Data source

```
ICollection<Product> products = new List<Product> {
     new Product("p1", "Pedigree Chum", 0.70, 1.42),
     new Product("p2", "Knife", 0.60, 1.31),
     new Product("p3", "Fork", 0.75, 1.57),
     new Product("p4", "Spaghetti", 0.90, 1.92),
     new Product("p5", "Cheddar Cheese", 0.65, 1.47),
     new Product("p6", "Bean bag", 15.20, 32.20),
     new Product("p7", "Bookcase", 22.30, 46.32),
     new Product("p8", "Table", 55.20, 134.80),
     new Product("p9", "Chair", 43.70, 110.20),
     new Product("p10", "Doormat", 3.20, 7.40)
};
```

- Query creation

```
IEnumerable<string> result =
         from p in products
         where p.RetailPrice < 50
         select p.Name;
```

- Query execution

```
foreach (string s in result)
{
    Console.WriteLine(s);
}
```

Extension methods

Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code, there is no apparent difference between calling an extension method and the methods that are actually defined in a type. For example, the following class defines an extension method to the int type named IsPrime.

```
namespace PrimeNumbers
{
    public static class ExtensionMethod
    {
        public static bool IsPrime(this int i)
        {
            for (int j = 2; j < i; j++)
            {
                if (i % j == 0)
                    return false;
            }
            return true;
        }
    }
}

int i = 17;
Console.WriteLine(i.IsPrime());
```

There are extension methods of IEnumerable<T> in the System.Linq namespace that perform projections, filters and aggregate operations. Many of these methods take Delegate arguments.

Delegates

A delegate declaration defines a type that encapsulates a method with a particular set of arguments and return type. Delegates can be instantiated; the constructor takes a method argument. For example Delegate1 defines a delegate that can encapsulate a method that takes an int argument and returns an int.

```
namespace ConsoleApplication.Examples
{
    public delegate int Delegate1(int i);
}
```

To instantiate the delegate, pass a compatible method into the constructor. The method associated with the delegate can then be invoked.

```
    public class Delegates
    {
        public static int DoubleIt(int i)
        {
            return i * 2;
        }

        public static void Main()
        {
            Delegate1 d1 = new Delegate1(DoubleIt);
            Console.WriteLine(d1.Invoke(5));
            //or simply
            Console.WriteLine(d1(5));
        }
    }
```

Generic delegates enable the parameters and return type of the associated method to be determined by the developer. For example

```
namespace ConsoleApplication.Examples
{
    public delegate TResult Delegate2<TSource, TResult>(TSource t);

    public class Delegates
    {
        public static bool IsEven(int i)
        {
            return i % 2 == 0;
        }

        public static void Main()
        {
            Delegate2<int, bool> d2 = new Delegate2<int, bool>(IsEven);
            Console.WriteLine(d2(7));
        }
    }
}
```

Lambda expressions

A lambda expression is an anonymous function and it is mostly used to create delegates in LINQ. Simply put, it's a method without a declaration, i.e., access modifier, return value declaration, and name. For example, the IsEven method could be written as a Lambda expression and passed into the delegate's constructor. The Lambda operator `=>` separates the parameter of the anonymous method from its content.

```
namespace ConsoleApplication.Examples
{
    public delegate TResult Delegate2<TSource, TResult>(TSource t);

    public class Delegates
    {
        public static void Main()
        {
            Delegate2<int, bool> d3 = new Delegate2<int, bool>(i => i % 2 ==0);
            Console.WriteLine(d3(7));
        }
    }
}
```

The System namespace defines a number of delegates, including Func

```
namespace ConsoleApplication.Examples
{
    public class Delegates
    {
        public static void Main()
        {
            Func<int, bool> func = new Func<int, bool>(i => i % 2 == 0);
            Console.WriteLine(func(7));
        }
    }
}
```

LINQ method syntax

```
 public static void MethodSyntax1()
 {
     ICollection<Product> products = new List<Product> {
         new Product("p1", "Pedigree Chum", 0.70, 1.42),
         new Product("p2", "Knife", 0.60, 1.31),
     };
     //The Where and Select extension methods of IEnumerable<T> both take Func
     //arguments
     IEnumerable<string> result = products.
              Where(p => p.RetailPrice < 50).
              Select(p => p.Name);
     result.ToList().ForEach(n => Console.WriteLine(n));

     int count = products.Count(p => p.RetailPrice > 50);

     //what is the average percentage markup on the cost price ?
     double percent = products.Average(
                               p => (p.RetailPrice / p.CostPrice) - 1) * 100;
     Console.WriteLine($"{percent:F1}%");
 }
```

An aggregation operation computes a single value from a collection of values. Methods of the IEnumerable interface include Average, Max, Min, Count, FirstOrDefault, SingleOrDefault

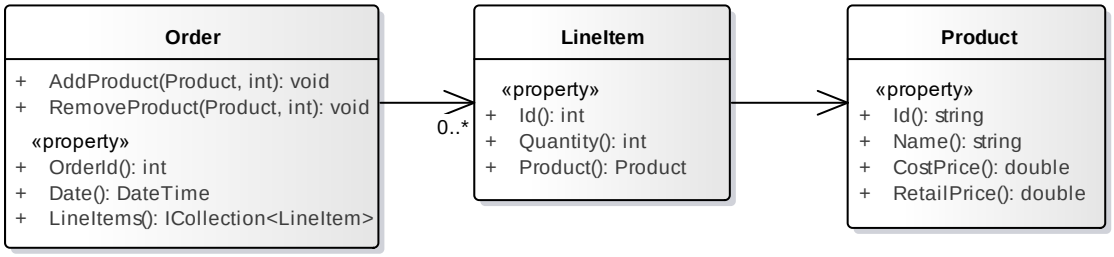
Anonymous types

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. You create anonymous types by using the new operator together with an object initializer. Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var.

```
var result = products.
    Where(p => p.RetailPrice < 50).
    Select(p => new { p.Id, p.Name } );

result.ToList().ForEach(x => Console.WriteLine(x.Name));
```

Adding Products to an Order

Test Driven Development

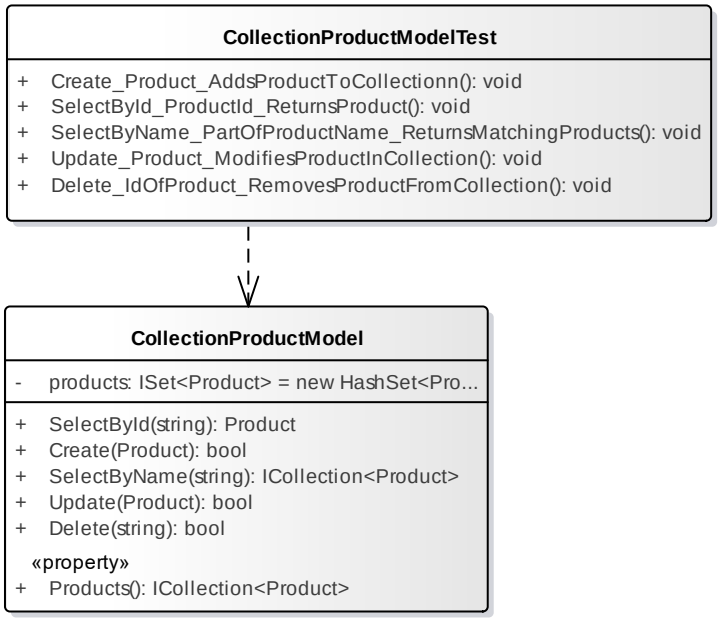



```
namespace Core.UnitTests.Entity
{
    public class OrderTest
    {
        [Fact]
        [Trait("Core.Entities", "Unit Test")]
        public void AddProduct_WhenPassed2Products_
                                    ShouldAddOneLineItemContaining2Products()
        {
            //arrange
            Order order = new Order();
            Product product = new Product("p1", "Dog Dinner", 1.20);
            //act
            order.AddProduct(product, 2);
            //assert
            Assert.NotNull(order.LineItems);
            Assert.Equal(1, order.LineItems.Count);
            Assert.Equal(2, order.LineItems.First().Quantity);
            Assert.True(order.LineItems.All(li => li.Product.Id == "p1"));
        }
```

First, generate the AddProduct and RemoveProducts methods and run the initially failing tests. Next, complete the methods so that the tests pass. Finally, refactor the code if required.

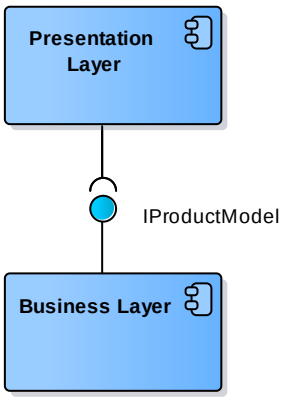
CollectionProductModel

Test Driven Development



1. Generate the CollectionProductModel class in the Core assembly
2. Generate the methods
3. Run the tests – this is the red phase
4. Complete the methods – the green phase
5. Optionally refactor

Extract Interface



The UML component diagram shows two components connected by an interface, IProductModel. An interface is a contract between components, providing just a specification for properties and methods, deferring implementation details to a class.

File Handling and Exceptions

Exceptions

Unhandled Exceptions

The following method would terminate with an unhandled exception

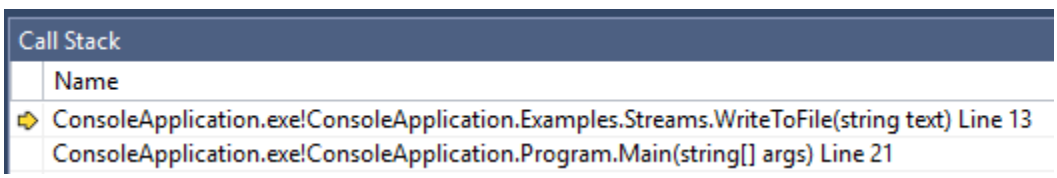
```
namespace ConsoleApplication.Examples
{
    class Streams
    {
        public static void WriteToFile(string text)
        {
            //UnauthorizedAccessException
            File.WriteAllText(@"C:\file.txt", text);
            //DirectoryNotFoundException
            File.WriteAllText(@"Z:\file.txt", text);
        }
    }
```

Catching Exceptions

Exception handling is the process of responding to the occurrence of exceptional conditions requiring special processing, often changing the normal flow of program execution. The documentation for the Streams.WriteToFile method indicates that it can throw a number of Exceptions, including UnauthorizedAccessException and DirectoryNotFoundException.

```
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Streams.WriteToFile("something");
            }
            catch (UnauthorizedAccessException e)
            {
                Console.WriteLine(e);
            }
            catch (DirectoryNotFoundException e)
            {
                Console.WriteLine(e);
            }
```

Debug > Windows > Call Stack



Exceptions are thrown down the call stack until they're caught

Throwing Exceptions

```
namespace ConsoleApplication.Tests
{
    public class MathsTest
    {
        [Fact]
        [Trait("Category", "Unit Test - State")]
        public void Factorial_NegativeNumber_ShouldThrowArgumentOutOfRangeException ()
        {
            Assert.Throws<ArgumentOutOfRangeException>(() => Maths.Factorial(-1));
        }
    }
}
```

Assert.Throws takes a The Func<object> delegate. This encapsulates a method (the test code) that has no parameters and returns an object.

```
namespace ConsoleApplication.Solutions
{
    public class Maths
    {
        public static double Factorial(int n)
        {
            if (n < 0)
                throw new ArgumentOutOfRangeException("argument can't be negative");
```

Move the following test from Green to Red

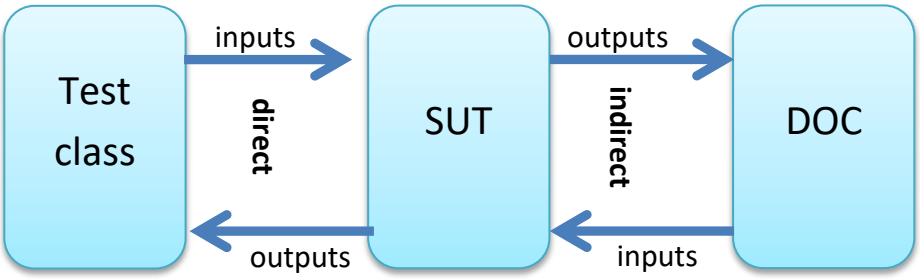
```
namespace Core.UnitTests.Entity
{
   public class OrderTest
   {
        [Fact]
        [Trait("Core.Entities", "Unit Test")]
        public void RemoveProduct_WhenPassedProductNotInLineItem_
                                                            ShouldThrowException()
        {
            //arrange
            Order order = new Order();
            order.LineItems = new List<LineItem> {
                new LineItem(new Product("p1", "Dog Dinner", 1.20), 5),
                new LineItem(new Product("p2", "Cutlery", 5.20), 2)
            };

            //act
            //assert
            Assert.Throws<InvalidOperationException>(()=>
                order.RemoveProduct(new Product("p3", "Cat Food", 1.15), 1)
            );
        }
```

Object serialization

Serialization is the process of converting the state of an object into a form that can be persisted or transported. The reverse is deserialization.

Binary serialization and XML serialization are part of the framework. A third party framework can be used for JSON serialization.

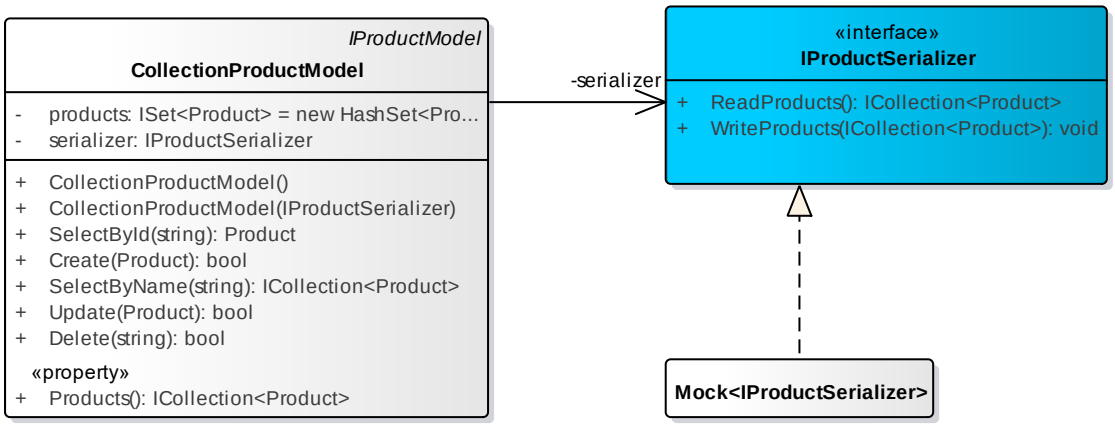
Interactions tests



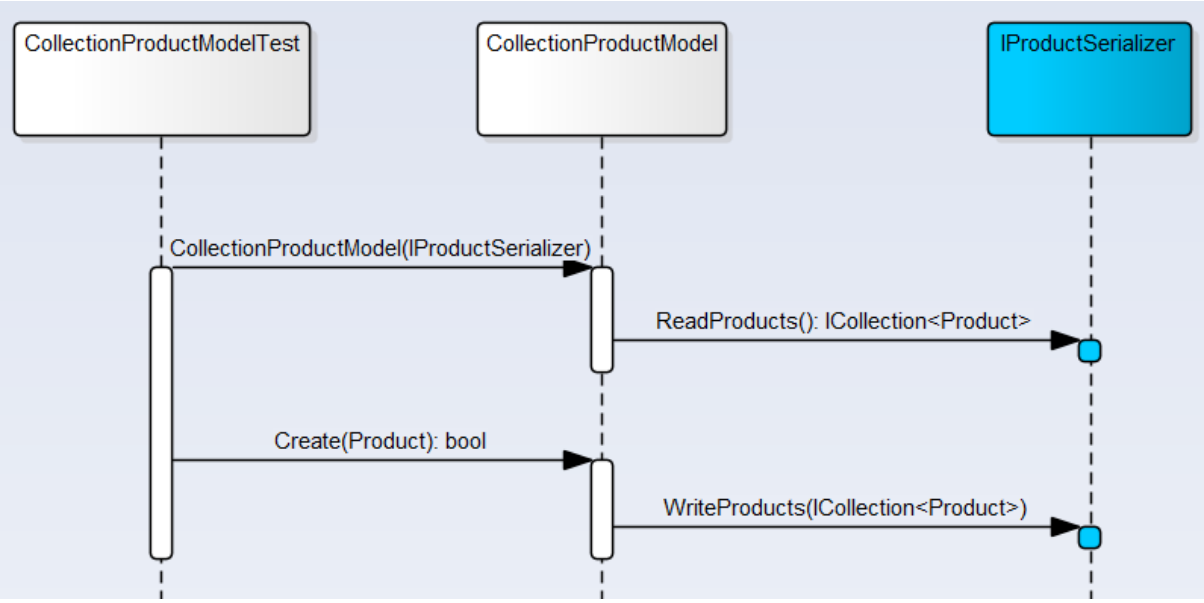
Unit tests apply to classes in isolation; the System Under Test (SUT) in the above diagram. They are intended to run fast and to pinpoint bugs with accuracy. **State testing** involves writing tests for direct inputs and outputs, while **interactions testing** verifies the way that the SUT interacts with collaborators (Depended On Components or DOCs).

Test doubles look and behave like their release-intended counterparts, but are actually simplified versions. Mocks are a category of test double that's pre-programmed with expectations which form a specification of the calls they are expected to receive.

Moq is a library for creating mock objects and verifying interactions between them and the SUT. For example, a mock implementation of IProductSerializer could be used to verify the interactions with the CollectionProductModel.



The sequence diagram shows the expected interactions

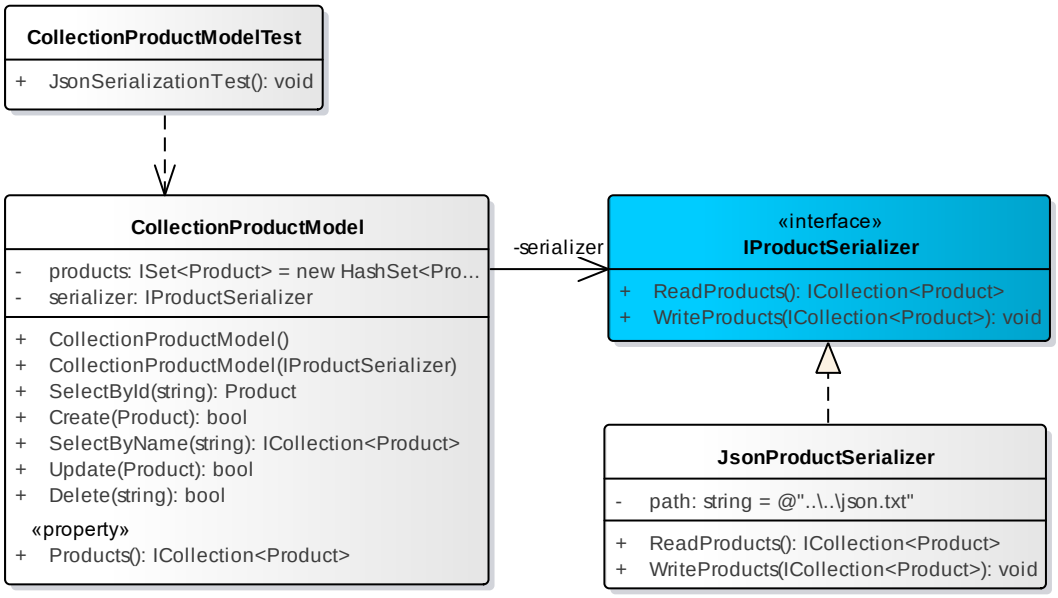


These are verified in the following tests

```
namespace Core.UnitTests.DataAccess
{
    public class CollectionProductModelTest
    {
        [Fact]
        [Trait("Category", "Unit Test - Interactions")]
        public void ConstructorCallsReadProductMethodOfSerializer()
        {
            //arrange
            Mock<IProductSerializer> serializer = new Mock<IProductSerializer>();
            serializer.Setup(s => s.ReadProducts()).Returns(new List<Product>());
            //act
            new CollectionProductModel(serializer.Object);
            //assert
            serializer.Verify(s => s.ReadProducts());
        }

        [Fact]
        [Trait("Category", "Unit Test - Interactions")]
        public void CreateProductCallsWriteProductMethodOfSerializer()
        {
            //arrange
            Mock<IProductSerializer> serializer = new Mock<IProductSerializer>();
            serializer.Setup(s => s.ReadProducts()).Returns(new List<Product>());
            CollectionProductModel model =
                                new CollectionProductModel(serializer.Object);
            Product product1 = new Product("p1", "Dog Dinner", 1.20);
            //act
            bool created = model.Create(product1);
            //assert
            serializer.Verify(s => s.WriteProducts(It.IsAny<ICollection<Product>>()));
        }
```

Integration tests

Unit tests focus on individual classes, while integration tests focus on the integration of different modules. This example tests the methods of the CollectionProductModel class, which uses the JsonProductSerializer class to persist the ICollection to a file.



Generate the JsonProductSerializer class in the Core project and run the test (the red phase)

```
namespace Core.IntegrationTests
{
    public class CollectionProductModelTest
    {
        [Fact]
        [Trait("Category", "Integration Test")]
        public void JsonSerializationTest()
        {
            File.Delete(@"..\..\products.json");
            CollectionProductModel model =
                      new CollectionProductModel(new JsonProductSerializer());
            Product product1 = new Product("p1", "Dog Dinner", 1.20);
            //serializes product collection
            bool created = model.Create(product1);
            //deserializes product collection
            model = new CollectionProductModel(new JsonProductSerializer());
            //assert
            Assert.True(model.Products.Contains(product1));
        }
```

```
//Requires Newtonsoft.Json.dll from NuGet
namespace Core.DataAccess.Collection
{
    public class JsonProductSerializer : IProductSerializer
    {
        private string path = @"..\..\products.json ";

        public ICollection<Product> ReadProducts()
        {
            if (!File.Exists(path))
                return null;
            string json = File.ReadAllText(path);
            return JsonConvert.DeserializeObject<ICollection<Product>>(json);
        }

        public void WriteProducts(ICollection<Product> products)
        {
            string output = JsonConvert.SerializeObject(products);
            File.WriteAllText(path, output);
        }
    }
}
```

SQL

To create and populate a table named Accounts in a SqlServer database, run the following script (which is in the Create folder of the Database project)

```
use store;

create table Accounts (
       Id nvarchar(128) not null primary key,
       Name nvarchar (max)
);

insert into accounts (id, name) values ('acc1','John Smith');
insert into accounts (id, name) values ('acc2','Jane Jones');
insert into accounts (id, name) values ('acc3','Brian Johnson');
insert into accounts (id, name) values ('acc4','Sue Smedley');

select * from accounts;
```

Some Transact-SQL Data Types

Data Type Categories	SQL Data types	C# type
Character strings	varchar	string
Unicode character strings	nvarchar	string
Exact numerics	int	int
Approximate numerics	float	double
Date and time	datetime2	DateTime
Binary data	varbinary	byte[]
Other Data Types	rowversion	byte[]

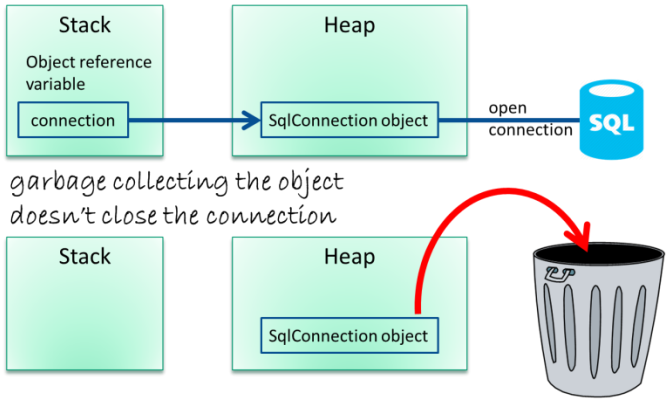
Connecting to SqlServer

The connection string can be stored a settings file. This stores key-value pairs. Open the project properties and select Setings.

```
namespace ConsoleApplication.Examples
{
    public class ConnectToDatabase
    {
        public static void Main()
        {
            string connectionString = Settings.Default.sqlserver;
            SqlConnection connection = new SqlConnection(connectionString);
            connection.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = connection;
            cmd.CommandText = "insert into accounts (id, name) values
                                                        ('acc1','John Smith');";
            int rowsInserted = cmd.ExecuteNonQuery();
            connection.Close();
        }
    }
}
```

Unmanaged Resources

Objects that wrap operating system resources, such as files, windows, network connections, or database connections are known as unmanaged resources, as they're not managed by the CLR. Objects that no longer have a reference become eligible for removal from the heap (garbage collection). Removing a SqlConnection object from memory won't close the connection to the database though.




```
string connectionString = new Settings().sqlserver;
SqlConnection connection = new SqlConnection(connectionString);
connection.Open();
SqlCommand cmd = new SqlCommand();
cmd.Connection = connection;
cmd.CommandText = "insert into account (id, name) values ('acc1'
int rowsInserted = cmd.ExecuteNonQuery();
connection.Close();
```

SqlException was unhandled

By placing the expression that can cause an exception in a try block, the following finally block which is always executed can be used to close the connection.

```
string connectionString = Settings.Default.sqlserver;
SqlConnection connection = new SqlConnection(connectionString);
try
{
    connection.Open();
    SqlCommand cmd = new SqlCommand();
    cmd.Connection = connection;
    cmd.CommandText = "insert into account (id, name) values
                                                    ('acc1','John Smith');";
    int rowsInserted = cmd.ExecuteNonQuery();
}
finally
{
     connection.Close();
}
```

Using blocks

A more concise alternative structure is to include a using block. This can be used with an object that implements the IDisposable interface. When the block exits, the unmanaged resource will be closed.

```
        public static void Main()
        {
            using (//an IDisposable object)
            {
                //connection to unmanaged resource is closed
                //when using block exits
            }
        }
```

Although the above code closes the unmanaged resource, it won't handle any exceptions, so a try and catch block can be included.

```
namespace ConsoleApplication.Examples
{
    public class ConnectToDatabase
    {
        public static void UsingBlock()
        {
            string connectionString = Settings.Default.sqlserver;
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand cmd = new SqlCommand())
                {
                    cmd.Connection = connection;
                    try
                    {
                        cmd.CommandText = "insert into account (id, name)
                                                values ('acc1','John Smith');";
                        int rowsInserted = cmd.ExecuteNonQuery();
                    }
                    catch (Exception e)
                    {
                        Console.WriteLine(e.Message);
                    }
                }
            }
```

Transactions

To ensure that multiple updates leave the database in a consistent state, a transaction can be started and either committed or rolled back. Transactions other than *Read Uncommitted* use row-level locking, and both will wait if they try to change a row updated by an uncommitted concurrent transaction (pessimistic concurrency)

```
public static void WithTransactions()
{
    string connectionString = Settings.Default.sqlserver;
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        using (SqlCommand cmd = new SqlCommand())
        {
            SqlTransaction transaction = connection.BeginTransaction(
                                                IsolationLevel.Serializable);
            cmd.Connection = connection;
            cmd.Transaction = transaction;
            try
            {
                //valid expression
                cmd.CommandText = "insert into accounts (id, name)
                                                values ('acc1','John Smith');";
                int rowsInserted = cmd.ExecuteNonQuery();
                //invalid expression
                cmd.CommandText = "update account set name = 'Jane Smith'
                                                where id = 'acc1';";
                int rowsUpdated = cmd.ExecuteNonQuery();
                transaction.Commit();
            }
            catch (Exception e)
            {
                transaction.Rollback();
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

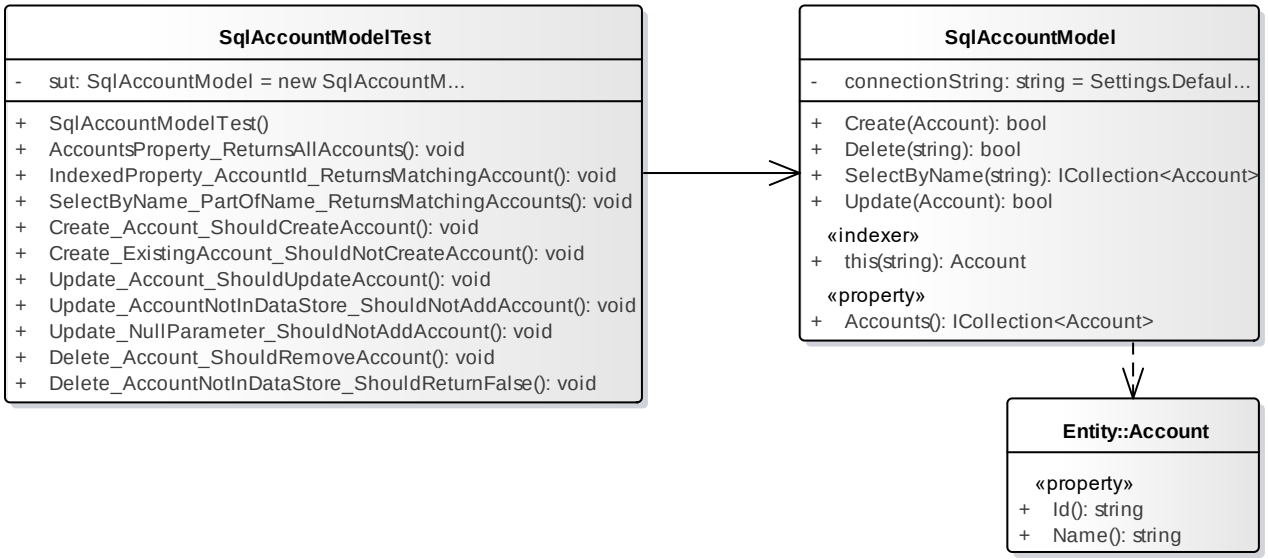
Isolation Level	Read	Update	Insert
Read Uncommitted	Reads data which is yet not committed.	Allowed	Allowed
Read Committed (Default)	Reads data which is committed.	Allowed	Allowed
Repeatable Read	Reads data which is committed.	Not Allowed	Allowed
Serializable	Reads data which is committed.	Not Allowed	Not Allowed

SqlAccountModel

Following a Test Driven Development approach, a class to store and retrieve Account objects can be generated from the test class.

The system under test, SqlAccountModel, includes an indexer, which enables Account objects to be indexed by the account id, in a similar way to an array. Use the *indexer* code snippet to generate it.

```
public class SqlAccountModel
{
    public Account this[string id]
    {
        get
        {
            throw new NotImplementedException();
        }
    }
```



Select Methods

```
namespace Core.DataAccess.SQL
{
    public class SqlAccountModel : IAccountModel
    {
        private string connectionString = Settings.Default.sqlserver;

        public ICollection<Account> SelectByName(string partOfName)
        {
            using (SqlConnection connection =
                      new SqlConnection(connectionString))
            {
                connection.Open();
                string sql = "select * from accounts where name like @name";
                SqlCommand cmd = new SqlCommand(sql, connection);
                cmd.Parameters.AddWithValue("name", "%" + partOfName + "%");
                ISet<Account> accounts = new HashSet<Account>();
                using (SqlDataReader dataReader = cmd.ExecuteReader())
                {
                    while (dataReader.Read())
                    {
                        accounts.Add(
                            new Account
                            {
                                Id = (string)dataReader["id"],
                                Name = (string)dataReader["name"]
                            });
                    }
                }
                return accounts;
            }
        }
```

Try it out – As well as adding the SelectByName method, complete the Accounts property and the indexer, so that the unit tests move from red to green.

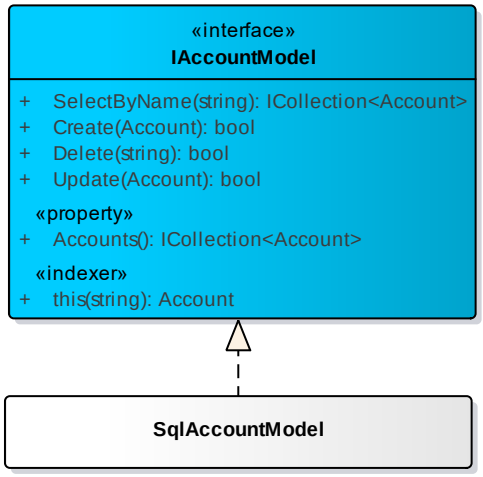
Create method

Complete the Update and Delete methods, which will be similar to the Create method, so that the unit tests move from red to green. The SQL expressions would be

```
insert into Accounts (id, name) values (@id, @name)";

delete from accounts where id = @id

update accounts set name = @name where id = @id
```

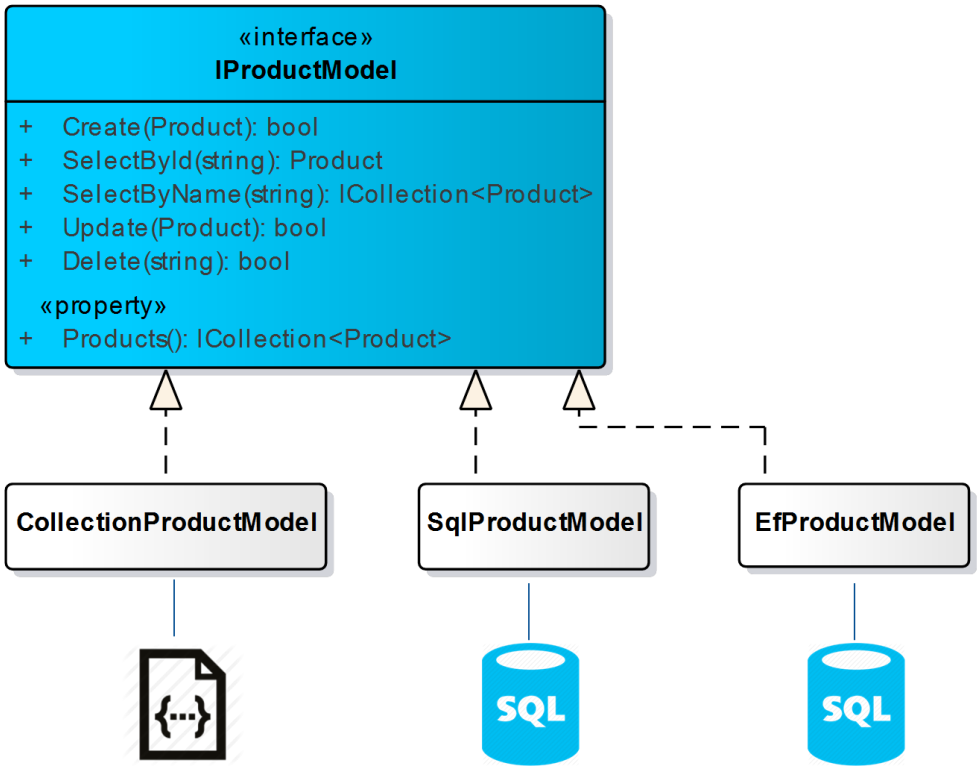
```
namespace Core.DataAccess.SQL
{
    public class SqlAccountModel
    {
        private string connectionString = Settings.Default.sqlserver;

        public bool Create(Account account)
        {
            using (SqlConnection connection =
                                new SqlConnection(connectionString))
            {
                connection.Open();
                using (SqlCommand cmd = new SqlCommand())
                {
                    cmd.CommandText = "insert into Accounts (id, name)
                                                        values (@id, @name)";
                    cmd.Connection = connection;
                    cmd.Parameters.AddWithValue("id", account.Id);
                    cmd.Parameters.AddWithValue("name", account.Name);
                    try
                    {
                        return cmd.ExecuteNonQuery() == 1;
                    }
                    catch (Exception)
                    {
                        return false;
                    }
                }
            }
        }
```

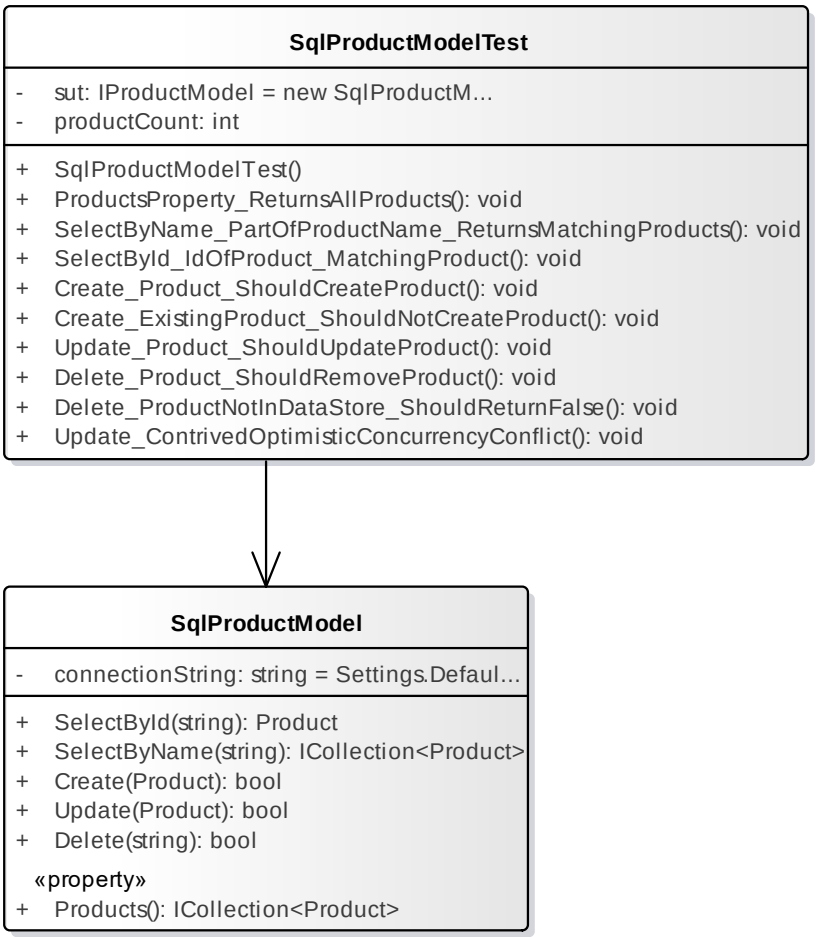
Extract the interface



Edit > Refactor > Extract interface

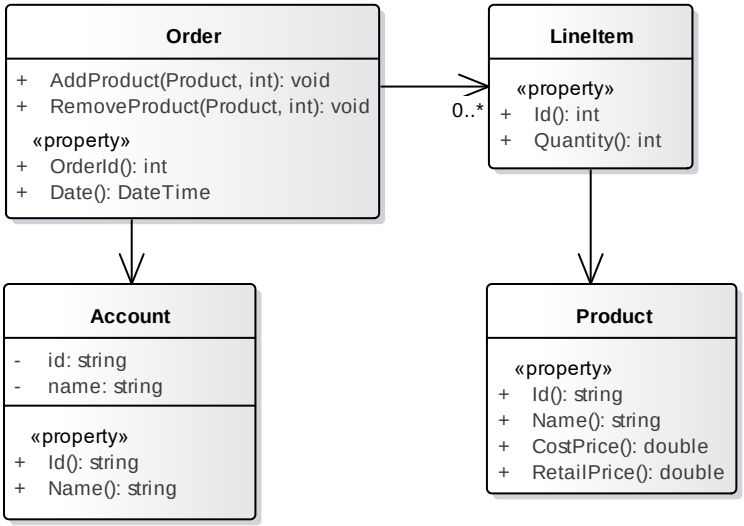
SqlProductModel

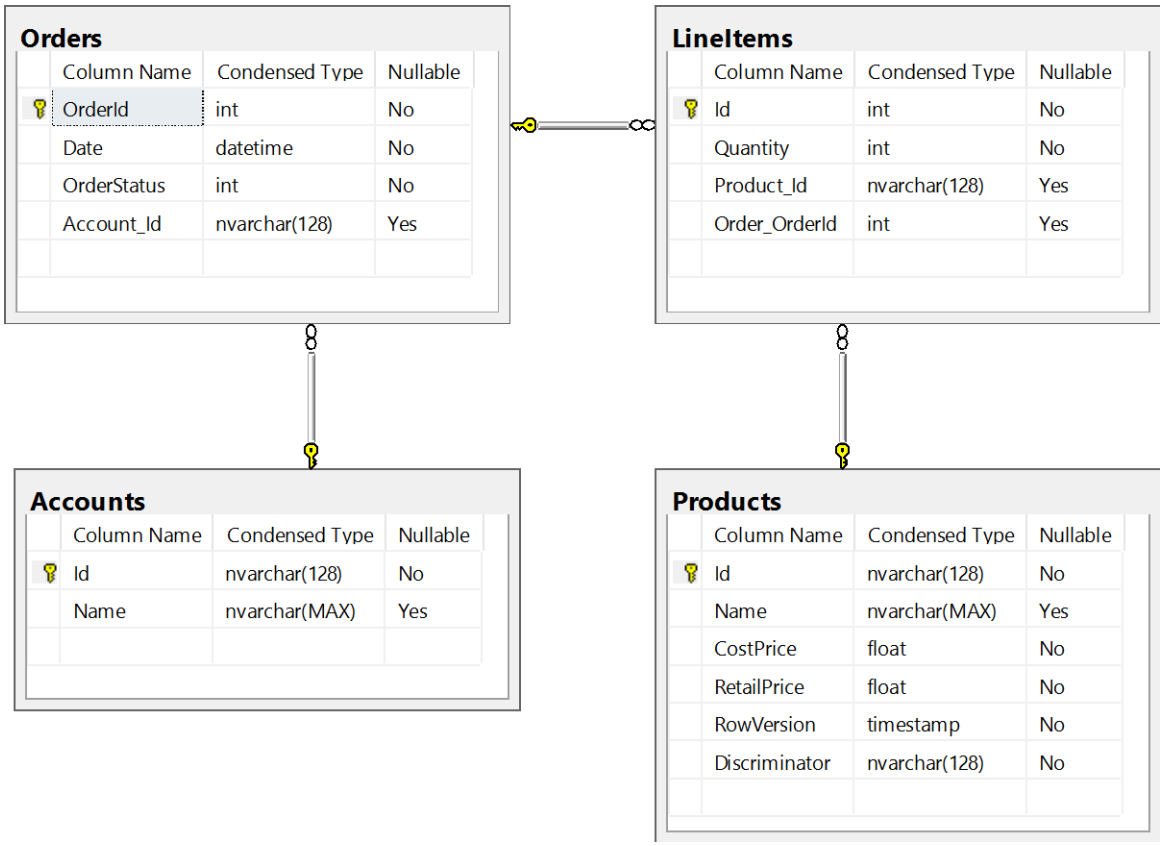



SqlProductModelTest
- sut: IProductModel = new SqlProductM...
- productCount: int
+ SqlProductModelTest()
+ ProductsProperty_ReturnsAllProducts(): void
+ SelectByName_PartOfProductName_ReturnsMatchingProducts(): void
+ SelectById_IdOfProduct_MatchingProduct(): void
+ Create_Product_ShouldCreateProduct(): void
+ Create_ExistingProduct_ShouldNotCreateProduct(): void
+ Update_Product_ShouldUpdateProduct(): void
+ Delete_Product_ShouldRemoveProduct(): void
+ Delete_ProductNotInDataStore_ShouldReturnFalse(): void
+ Update_ContrivedOptimisticConcurrencyConflict(): void
SqlProductModel
- connectionString: string = Settings.Defaul...
+ SelectById(string): Product
+ SelectByName(string): ICollection<Product>
+ Create(Product): bool
+ Update(Product): bool
+ Delete(string): bool
«property»
+ Products(): ICollection<Product>

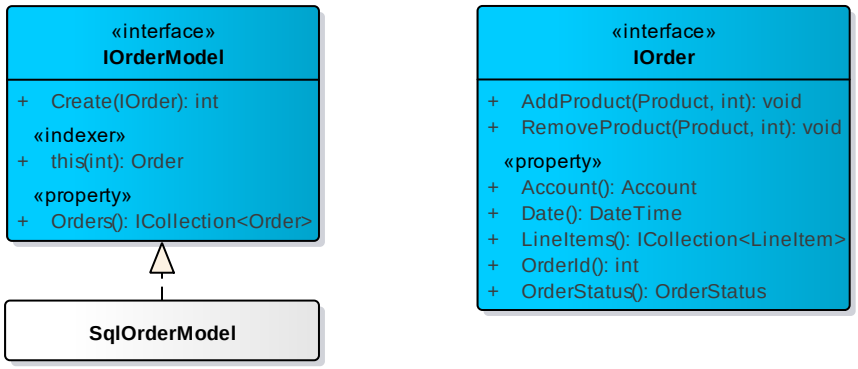
SqlOrderModel

Related tables





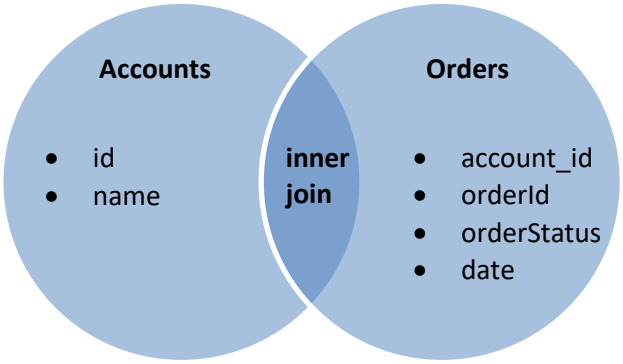
Create



Inserting an Order into the database involves the following steps

1. Insert a row into the Order table and retrieve the generated primary key
    a. `insert into orders(date, orderStatus, account_id) values(@date, @orderStatus, @account_id) select scope_identity()`
2. For each LineItem in the Order, insert the quantity, product_id and order_id
    a. `insert into LineItems (quantity, product_id, order_orderId) values (@quantity, @product_id, @order_orderId)`

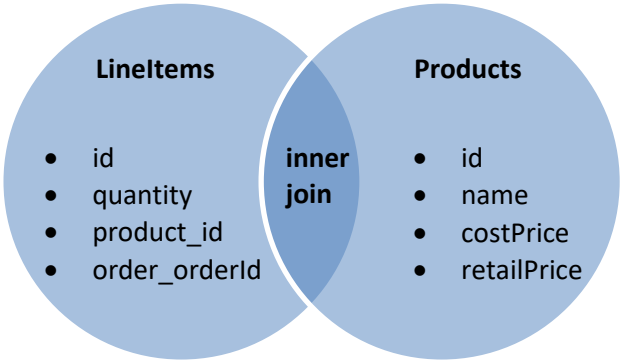
Table Joins



Because SQL is based on set theory, each table can be represented as a circle in a Venn diagram. The ON clause in the SQL SELECT statement that specifies join conditions determines the point of overlap for those circles and represents the set of rows that match. For example, in an inner join, the overlap occurs within the interior or "inner" portion of the two circles. An outer join includes not only those matched rows found in the inner cross section of the tables, but also the rows in the outer part of the circle to the left or right of the intersection.

To get the id and name of an account for a specified order id, join the two tables:

```
select id, name from Accounts
inner join Orders on Orders.Account_Id = Accounts.Id
where OrderId = 1
```



To get the details of a lineItem, including the associated product, join the LineItems and Product tables. The point of overlap is where the LineItems.Product_Id = Products.Id

```
select LineItems.Id as LineItemId, LineItems.Quantity, Products.Id as ProductId,
Products.Name, Products.CostPrice, Products.RetailPrice from Products
inner join LineItems on LineItems.Product_Id = Products.Id
where LineItems.Order_OrderId = 1;
```

Orders Property

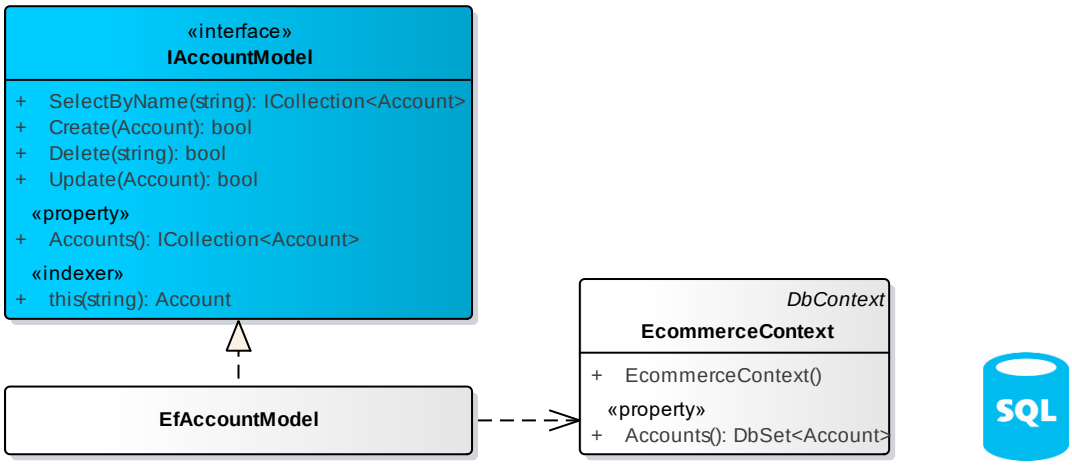
Selecting Orders from the database involves the following steps

1. Add orders to a collection
    a. `select * from Orders`
2. For each order
    a. Read from Accounts table into account object
        i.
```
select id, name from Accounts
inner join Orders on Orders.Account_Id = Accounts.Id
where OrderId = @orderId
```
    b. Add LineItem objects to the Order

```
select LineItems.Id as LineItemId, LineItems.Quantity,
Products.Id as ProductId, Products.Name, Products.CostPrice,
Products.RetailPrice from Products
inner join LineItems on LineItems.Product_Id = Products.Id
where LineItems.Order_OrderId = @orderId
```

Entity Framework

IAccountModel



DbContext

The methods in the IAccountModel class use the EcommerceContext class; this is illustrated in the previous diagram by a UML dependency.

The DbContext class enables querying a database, grouping together changes and then writing the changes back to the database as a unit. DbContext is typically used with a derived type that contains DbSet<TEntity> properties for the root entities of the model. These sets are automatically initialized when the instance of the derived class is created. The DbContext constructor takes the connection string of the database as an argument.

DbContext is the primary class responsible for interacting with data as objects. It manages the entity objects during run time, which includes populating objects with data from a database, change

tracking, and persisting data to the database. Generally, a class deriving from DbContext is build that exposes DbSet properties that represent collections of entities. The constructor takes a string argument; this is the connection string for the database.

```
namespace Core.DataAccess.EntityFramework
{
    public class EcommerceContext : DbContext
    {
        public EcommerceContext() : base(Settings.Default.sqlserver)
        {
        }
        public DbSet<Account> Accounts { get; set; }
    }
}
```

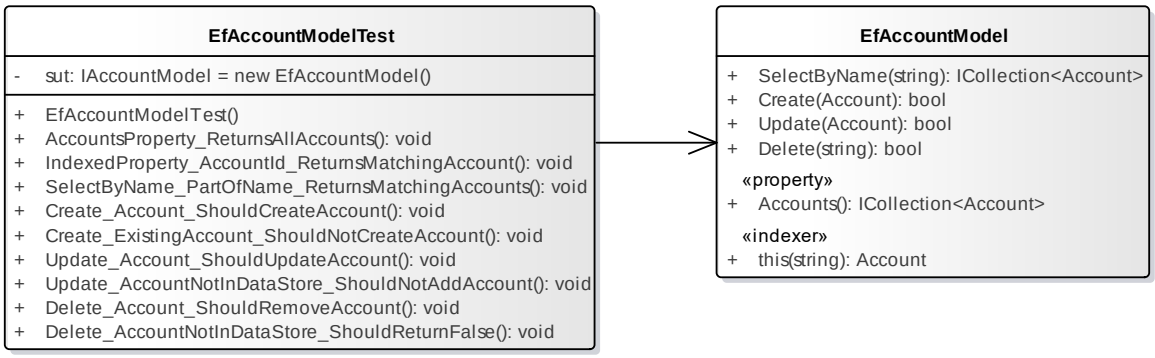
Code first migrations

Included with the Entity Framework, code first migrations can be used to keep the data model in sync with the database.

- Add DbSet<Account> to DbContext
- Select Core project in package manager console
- enable-migrations
- Add-Migration Accounts
- Update-database -verbose
- To migrate to a specific version, Update-Database -TargetMigration: Accounts

By default, AutomaticMigrationsEnabled is set to false. If set to true, the Add-Migration command isn't required; just make changes to the model and execute "update-database"

EfAccountModelTest



Build an outline for the EfAccountModel class in the Core project and run the integration test. Next, move the test results from red to green.

```
namespace Core.DataAccess.EntityFramework
{
    public class EfAccountModel : IAccountModel
    {
        public ICollection<Account> Accounts
        {
            get
            {
                using (EcommerceContext context = new EcommerceContext())
                {
                    return context.Accounts.ToList();
                }
            }
        }

        public Account this[string id]
        {
            get
            {
                using (EcommerceContext context = new EcommerceContext())
                {
                    return context.Accounts.Find(id);
                }
            }
        }

        public ICollection<Account> SelectByName(string partOfName)
        {
            using (EcommerceContext context = new EcommerceContext())
            {
                return context.Accounts.Where(
                          account => account.Name.Contains(partOfName)).ToList();
            }
        }

        public bool Create(Account account)
        {
            using (EcommerceContext context = new EcommerceContext())
            {
                if (context.Accounts.Find(account.Id) != null)
                    return false;
                context.Entry(account).State = EntityState.Added;
                int rows = context.SaveChanges();
                return rows == 1;
            }
        }

        public bool Update(Account account)
        {
            using (EcommerceContext context = new EcommerceContext())
            {
                if (!context.Accounts.Any(a => a.Id == account.Id))
                    return false;
                context.Entry(account).State = EntityState.Modified;
                int rowsUpdated = context.SaveChanges();
                return rowsUpdated == 1;
            }
        }
```

```
public bool Delete(string id)
        {
            using (EcommerceContext context = new EcommerceContext())
            {
                Account account = context.Accounts.Find(id);
                if (account == null)
                    return false;
                context.Entry(account).State = EntityState.Deleted;
                int rowsDeleted = context.SaveChanges();
                return rowsDeleted == 1;
            }
        }
    }
}
```

Entity State

The DbContext needs to know the state of an object to save changes back to the data source. ObjectStateEntry objects store EntityState information. The SaveChanges method of the DbContext processes entities that are attached to the context and updates the data source depending on the EntityState of each object.

Member name	Description
Detached	The object exists but is not being tracked.
Added	The object has been added to the object context, but the SaveChanges method has not been called.
Unchanged	The object has not been modified since it was attached to the context
Deleted	The object has been deleted from the object context. After the changes are saved, the object state changes to Detached.
Modified	One of the properties on the object was modified and the SaveChanges method has not been called.

Running tests in parallel

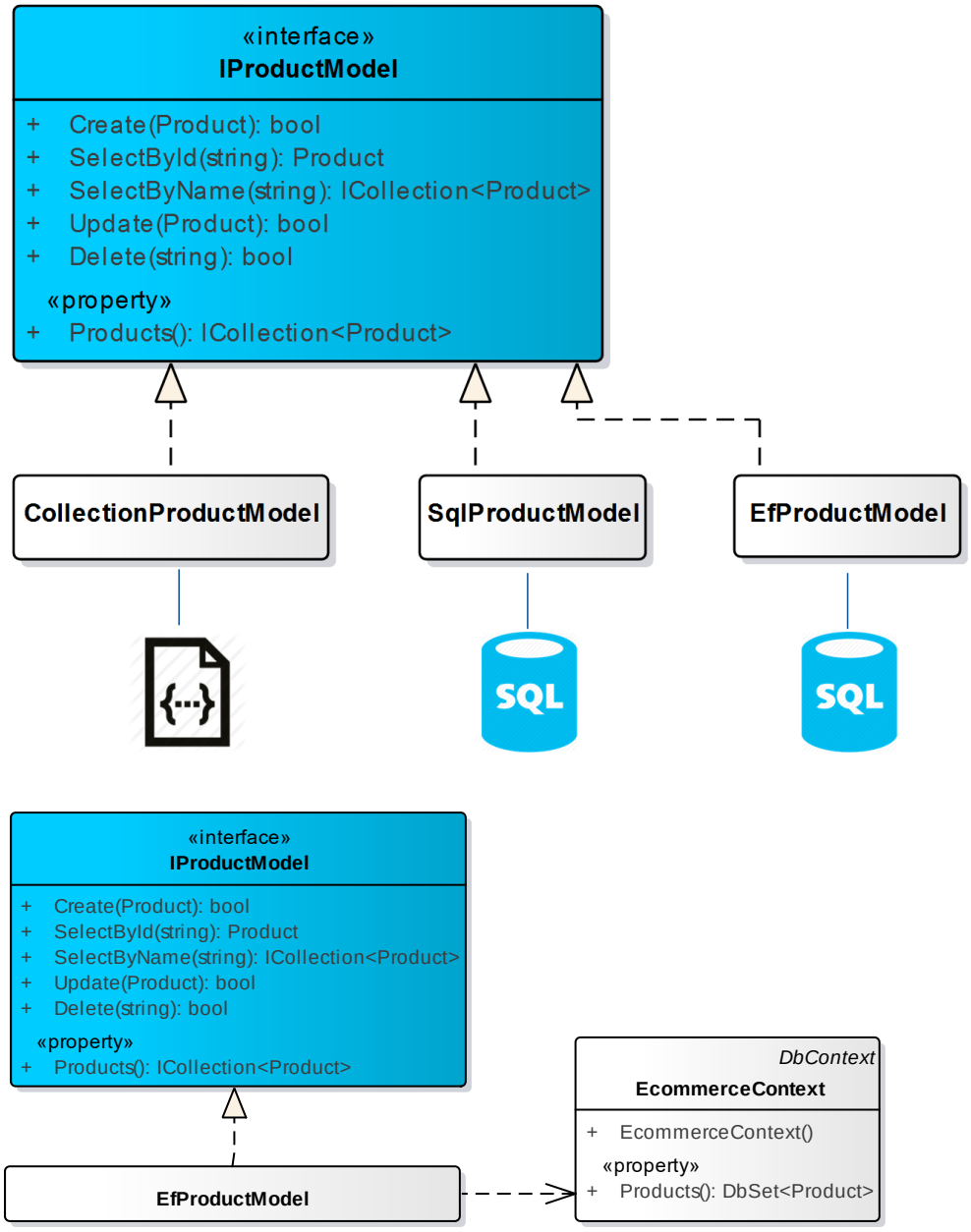
By default, xUnit tests within the same class will not run in parallel, whereas test classes in the same assembly will run in parallel. This parallelisation can cause exceptions when several test methods write to the same database.

To indicate that multiple test classes should not be run in parallel, place them into the same test collection by annotating each test class with [Collection("Collection 1")] or alternatively by adding an assembly level attribute.

```
AssemblyInfo.cs
[assembly: CollectionBehavior(DisableTestParallelization = true)]
```

IProductModel

Alternative implementations of the IProductModel interface could be written. SqlProductModel queries and updates a SqlServer database using SQL expressions, while EfProductModel uses the Entity Framework to map between objects and rows in a database.



Optimistic concurrency

In an optimistic concurrency model, a violation is considered to have occurred if, after a user receives a value from the database, another user modifies the value before the first user has attempted to modify it.

You can resolve conflicts by handling OptimisticConcurrencyExceptions that the Entity Framework throws. In order to know when to throw these exceptions, the Entity Framework must be able to detect conflicts. One option is to include a tracking column that can be used to determine when a

row has been changed. The data type of the tracking column is typically rowversion. The rowversion value is a sequential number that's incremented each time the row is updated.

```
    public class Product
    {
        public string Id { get; set; }
        public string Name { get; set; }
        public double CostPrice { get; set; }
        public virtual double RetailPrice { get; set; }
        [Timestamp]
        public byte[] RowVersion { get; set; }
```

When the TimestampAttribute attribute is used with a Dynamic Data field, the column is not displayed unless the ScaffoldColumnAttribute attribute of the column is explicitly set to true.

Discriminator

Due to the inheritance hierarchy, the Entity Framework generates a Discriminator column in the Product table to enable VeblenGood and NormalGood objects to be mapped to the database.

```
namespace Core.IntegrationTests
{
    [Collection("Collection 1")]
    public class EfProductModelTest
    {
        private IProductModel sut = new EfProductModel();
            //arrange
            VeblenGood product = new VeblenGood
            {
                Id = "v1",
                Name = "Rolex watch",
                CostPrice = 700
            };
            //act
            sut.Create(product);
            Product retrievedProduct = sut.SelectById("v1");
            //assert
            Assert.IsType(typeof(VeblenGood), retrievedProduct);
        }
    }
}
```

Code first migrations

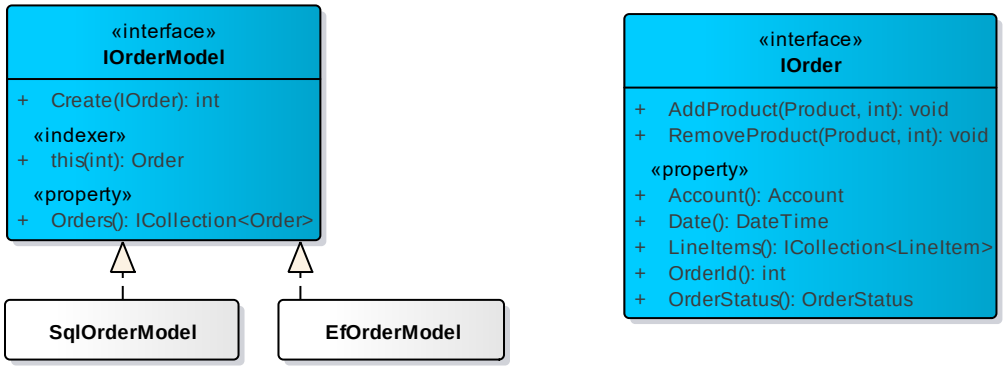
- Add DbSet<Product> to DbContext
- enable-migrations
- Add-Migration Products
- Update-database -verbose

IOrderModel

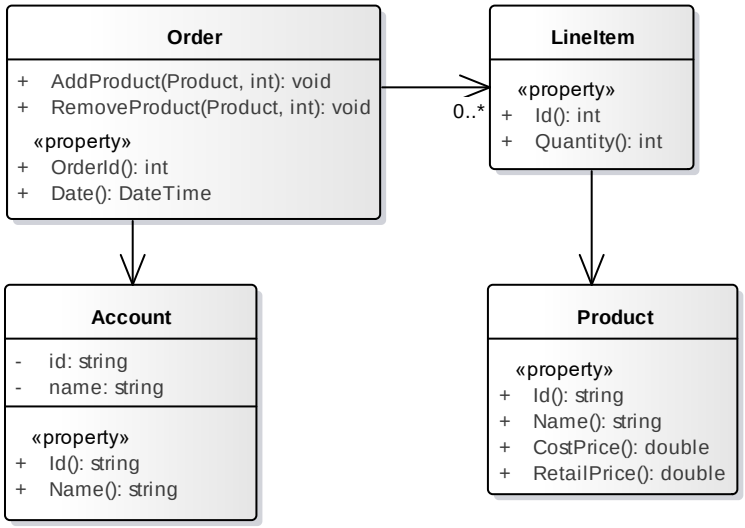
Code first migrations

- Add DbSet<Order> to DbContext
- enable-migrations
- Add-Migration Order
- Update-database -verbose

EfOrderModel



Associated objects



Create

Adding an Order to the database with the Entity Framework involves the following steps

1. Checking that the Account object associated with the Order already exists in the Accounts table
2. Change the EntityState of the Order from Detached to Added
3. Change the EntityState of the associated Account and Product objects to Unchanged, so that the Entity Framework won't attempt to insert them into the database
4. Call the SaveChanges method of the DbContext. This will add a row to the Order table and rows to the LineItems table for each LineItem in the order.
5. Return the generated id of the Order table

```
namespace Core.DataAccess.EntityFramework
{
    public class EfOrderModel : IOrderModel
    {
        public int Create(IOrder order)
        {
            using (EcommerceContext context = new EcommerceContext())
            {
                context.Database.Log = s => System.Diagnostics.Debug.WriteLine(s);
                order.Account = context.Accounts.Find(order.Account.Id);
                if (order.Account == null)
                    throw new InvalidOperationException("Account doesn't exist");
                context.Entry(order).State = EntityState.Added;
                context.Entry(order.Account).State = EntityState.Unchanged;
                order.LineItems.ToList().ForEach(lineItem =>
                    context.Entry(lineItem.Product).State = EntityState.Unchanged);
                context.SaveChanges();
                return order.OrderId;
            }
        }
```

Orders property

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved by use of the Include method. For example

```
namespace Core.DataAccess.EntityFramework
{
    public class EfOrderModel : IOrderModel
    {
        public ICollection<Order> Orders
        {
            get
            {
                using (EcommerceContext context = new EcommerceContext())
                {
                    //Eager loading of referenced objects
                    return context.Orders
                                  .Include(order => order.LineItems.Select(
                                           listItem => listItem.Product))
                                  .Include(order => order.Account)
                                  .ToList();
                }
```

Indexer

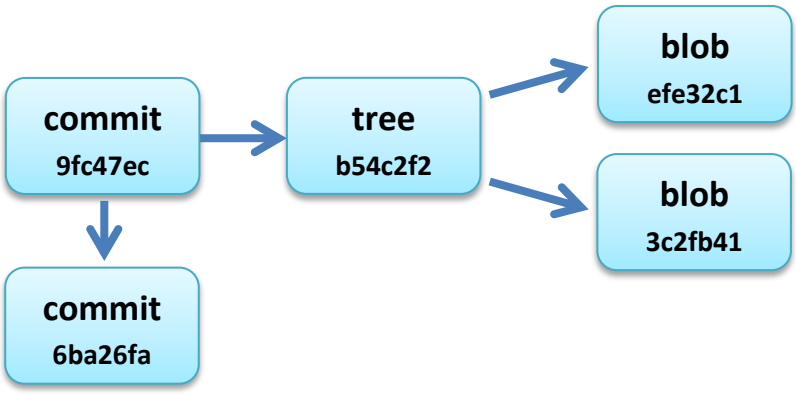
```
        public Order this[int id]
        {
            get
            {
                using (EcommerceContext context = new EcommerceContext())
                {
                    //method syntax
                    return context.Orders
                        .Where(order => order.OrderId == id)
                        .Include(order => order.LineItems.Select(
                                         listItem => listItem.Product))
                                         .Include(order => order.Account)
                                         .First();
                }
            }
```

Workflow

Git is a distributed revision control system with an emphasis on speed, data integrity and support for distributed, non-linear workflows.

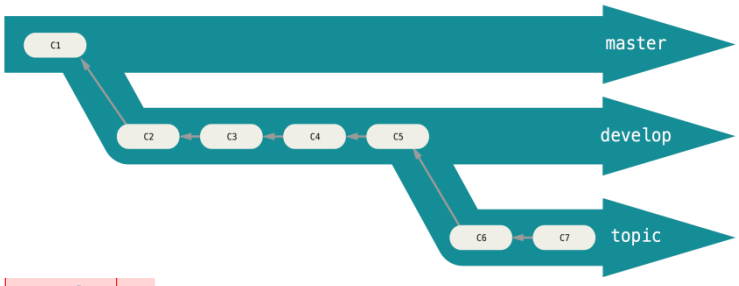
Every Git working directory is a full-fledged repository with complete history and full version-tracking capabilities, independent of network access or a central server.

1. Create a local repository
    - `>git init`
2. Add a .gitignore file, listing files that Git should ignore
3. Add files to the index. SHA-1 checksum is generated for each file and the bytes are stored in the repository as a blob
    - `>git add *`
4. commit files in the index; this creates a snapshot of the project
    - `>git commit –m “message”`
    - tree object with checksum is generated for project directories
    - commit object with checksum includes metadata and points at root directory and previous commit



5. Optionally add a remote repository
    - `>git remote add origin https://dineen701@bitbucket.org/dineen701/.net-course.git`
6. Push to the remote
    - `>git push origin master`
7. Create and checkout develop branch
    - `>git branch develop`
    - `>git checkout develop`

A branch is a pointer to a commit. The default branch name in Git is **master**. A pointer called **HEAD** tracks the current branch. A workflow might maintain a master branch for stable code that will be released; a branch named develop to test stability and short-lived topic branches



Merging[U1]

Id	Message
f6adda2	**develop** HEAD C2
ea3363c	master Main method

1. Checkout master
   - `>git checkout master`
2. and merge in the develop branch
   - `>git merge develop`

Id	Message
f6adda2	develop **master** HEAD C2
ea3363c	Main method

This is called a fast forward merge, as there's no divergent work to merge together

3. Merge branches that have updated the same file
   - `>git merge develop`

This results in a conflict. The version in HEAD (the master branch) contains changes that conflict with the develop branch. Edit the file

```
<<<<<<< HEAD
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).info("Hello");
        //modified in master branch
=======
        Logger.getLogger(Logger.GLOBAL_LOGGER_NAME).info("Hello");
        //added in develop branch
>>>>>>> refs/heads/develop
```

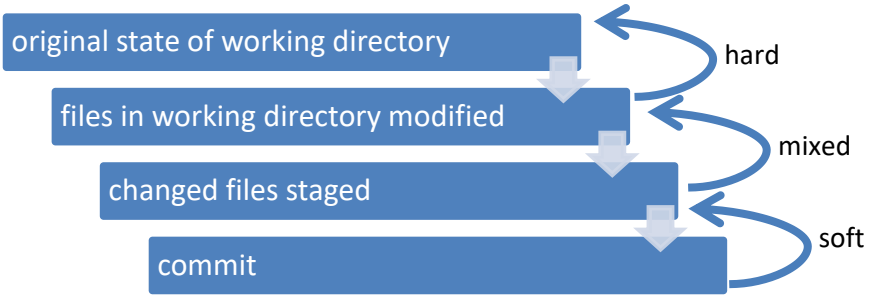
Save the amended file, then stage and commit

- `> git add *`
- `> git commit –m 'conflict resolved'`

Reset

Checkout moves HEAD, the pointer to the current branch. This changes the working directory but does not delete previous commits

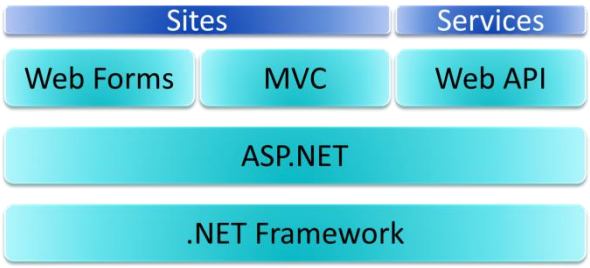
1. A **soft reset** undoes previous commits, moving the files to the index (see git staging area). The working directory is unchanged.
   a. `>git reset –soft ea3363c`
2. A **mixed reset** differs from a soft reset in that changes are unstaged
3. A **hard reset** undoes changes to the working directory. Unlike checking out a commit, this is not reversible.



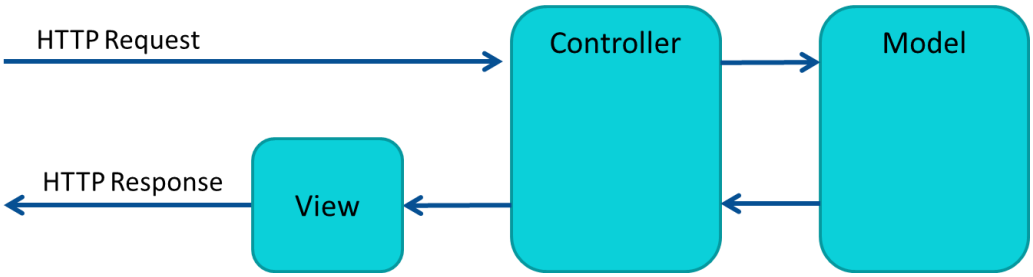
MVC

Overview

ASP.NET MVC is a framework for building web applications that applies the Model View Controller pattern to the ASP.NET framework. Web API offers the MVC development style but is tailored to writing HTTP services.



- The model comprises classes that describe the data you're working with as well as the business rules for how the data can be accessed and modified
- The View is the application's user interface, which with MVC is a template to dynamically generate HTML.
- The Controller handles communication with the user, overall application flow and application-specific logic. With MVC, the controller responds to user input, communicates with the model and determines which view to render (if any).



Add a Web Application project to the solution, selecting MVC and Web API.

Directory	Purpose
Controllers	Classes handling requests
Models	Data classes can be in a separate assembly
Views	Template files for rendering HTML
Scripts	Javascript files
Content	Image files, CSS

Controllers

Add a MVC controller – Empty named ProductController.

```
public class ProductController : Controller
{
    public ContentResult Index()
    {
        return Content("Home Page"); //returns text directly from the Controller
    }
}
```

Routing

The controller is central to a MVC application; URLs map to methods (actions) in a controller. For example, the URL */product/index* maps to the Index method in the ProductController class. This mapping can be configured in the RegisterRoutes method of the RouteConfig class in the App_Start folder

```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
       routes.MapRoute(
          name: "Default",
          url: "{controller}/{action}/{id}",
```

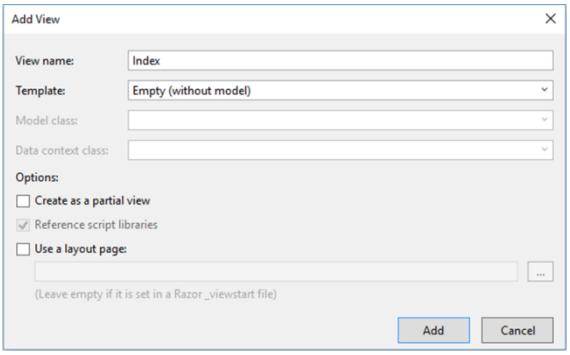
/Product/Index
maps to the Index method in the ProductController class

Views

The View method of the Controller class returns a ViewResult, which represents a class that is used to render a view.

```
public class ProductController : Controller
{
    public ViewResult Index()
    {
        return View();
    }
}
```

The Index method will render a view with the same name, Index, in the Product folder. Right click the method and select “Add View”. Select the Empty template and uncheck the “use a layout page” box.

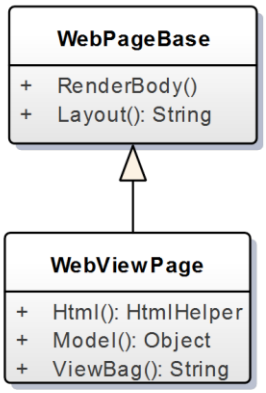


This will create a cshtml page; a mixture of HTML and C#. The C# is prefixed with the @ symbol. The page inherits from WebViewPage, so has access to its methods and properties, including the Layout property.

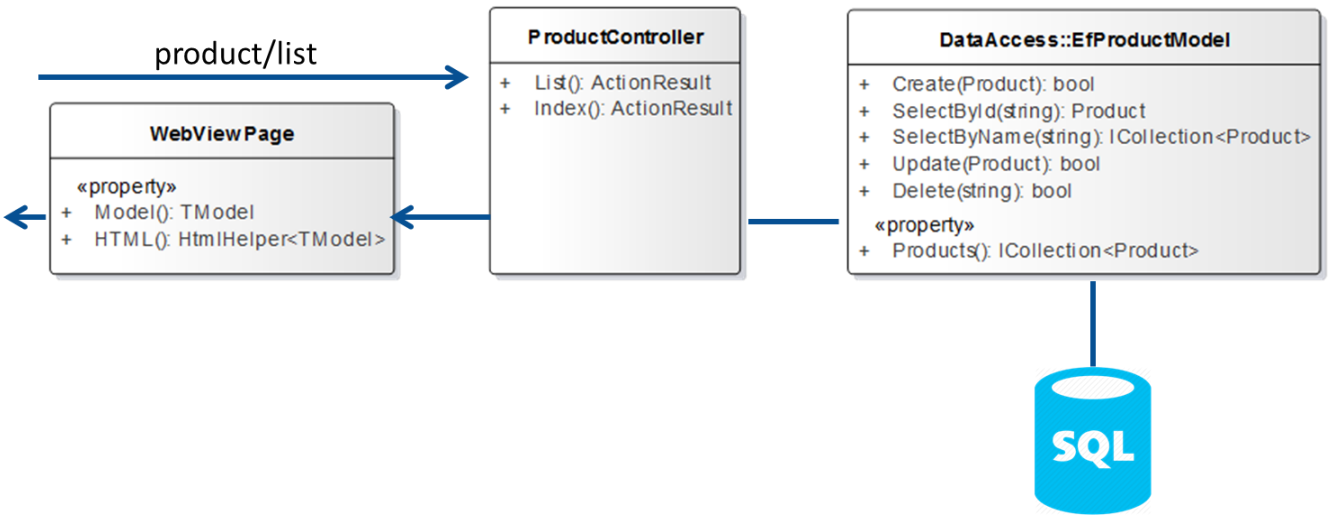
```
@{
    Layout = null;
}
```

Razor code block

```
<!DOCTYPE html>

<html>
<head>
    <title>Index</title>
</head>
<body>
    <div>
        <h1>Home Page</h1>
    </div>
</body>
</html>
```



Models



Add a reference to the project containing the EfProductModel class. The List method of the ProductController class calls the Products property of the Model, and passes this List to the View.

```
namespace WebClient.Controllers
{
    public class ProductController : Controller
    {
        private IProductModel productModel = new EfProductModel();

        // GET: Product
        public ActionResult List()
        {
            ICollection<Product> products = productModel.Products;
            return View(products);
        }
    }
}
```

The View can iterate through this List, generating a HTML table

```
@using Core.Entity
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <title>List</title>
</head>
<body>
    <table>
            @foreach (Product product in Model)
            {
                <tr>
                    <td>@product.Id</td>
                    <td>@product.Name</td>
                    <td>@product.CostPrice.ToString("C")</td>
                    <td>@product.RetailPrice.ToString("C")</td>
                </tr>
            }
    </table>
</body>
</html>
```

Forms

Method	Desired action
GET	retrieve data
POST	add new data passed with request
PUT	modify data at specified URL
DELETE	delete resource

The following Controller class contains an overloaded Create method. The first responds to an HTTP GET request, returning a View containing a form. The second responds to an HTTP POST containing the form values. The request values are used to set the properties of the Product object, a process known as binding.

```
public class ProductController : Controller
{
    private IProductModel productModel = new EfProductModel();
    public ActionResult Create()
    {
        return View();
    }

    [HttpPost]
    public ActionResult Create(Product product)
    {
        productModel.Create(product);
        return RedirectToAction("List");
    }
```

GET request to /product/create

POST request to /product/create

The form contains HTML input elements whose values are posted to the sever

```
<!DOCTYPE html>
<html>
<body>
    <h1>Add a product</h1>
    <form action="Create" method="post">
        Id: <input type="text" name="Id"/><br/>
        Name: <input type="text" name="Name" /><br />
        Cost Price: <input type="text" name="CostPrice" /><br />
        Retail Price: <input type="text" name="RetailPrice" /><br />
        <input type="submit" value="Create" />
    </form>
</body>
</html>
```

Improving the user interface

HTML helper methods

Add a Controller named ShoppingController and generate the View for the Index method, selecting the checkbox for using a layout page.

WebViewPage, the base class for an MVC View, includes an Html property of type HtmlHelper.

The System.Web.Mvc.Html namespace contains classes that help render HTML controls in an MVC application. The namespace includes classes that support forms, input controls, links, partial views, validation, and more. For example the ActionLink method generates an HTML anchor.

DI

Using NuGet, add a reference to the Unity.Mvc assembly. The Unity container injects dependencies in to a constructor.

When you register a type, the default behaviour is for the container to use a transient lifetime manager. It creates a new instance of the type each time you call the Resolve method or when the dependency mechanism injects instances into other classes.

```
namespace WebClient.App_Start
{
    public class UnityConfig
    {
        public static void RegisterTypes(IUnityContainer container)
        {
            container.RegisterType<IProductModel, EfProductModel>(
                                                new TransientLifetimeManager());
```

Having registered the dependency, it will be injected into the controller's constructor

```
namespace WebClient.Controllers
{
    public class ShoppingController : Controller
    {
        private IProductModel productModel;

        public ShoppingController(IProductModel productModel)
        {
            this.productModel = productModel;
        }
```

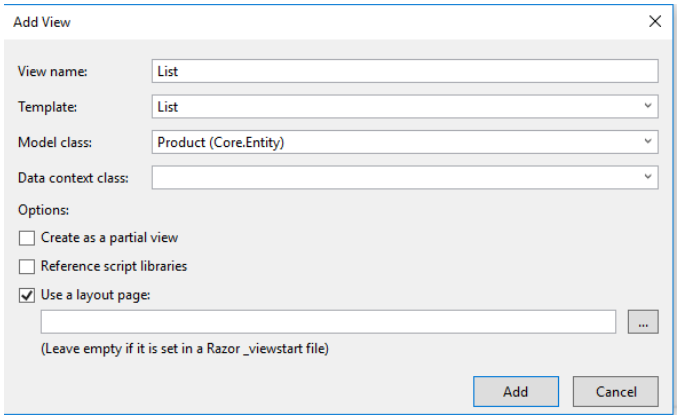
Scaffolding

Scaffolding, in MVC, uses templates to generate controllers and views for CRUD functionality in an application. Annotating the Product class's Id property with the ScaffoldColumn attribute will include this property in the generated views.

```
public class Product
{
    [ScaffoldColumn(true)]
    public string Id { get; set; }
```

Add a method named ProductList to the ShoppingController class and generate a View using the List template.

```
public class ShoppingController : Controller
{
    private IProductModel productModel;

    public ViewResult List(string partOfName)
    {
        return View(partOfName==null?
                    productModel.Products :
                    productModel.SelectByName(partOfName));
    }
```



The helper method
`@Html.ActionLink("Select", "AddProduct", new { id = item.Id })`
Generates an anchor that points at the AddProduct method in the controller class.

ASP.NET State Management

- HttpContext exists for the lifetime of a single request.
- The Session object enables data to be stored for a specific user while they navigate the pages in the site. Several storage options are available: the default is to use the worker process; alternatives are to use a separate process or a SQL Server database.

```
<sessionState mode="[Off|InProc|StateServer|SQLServer|Custom]"
              cookieless="[true|false|AutoDetect|UseCookies|UseUri]"
              timeout="20" />
```

- The Application object is an instance of the HTTPApplication class. Typically this would be initialised in the Application_Start handler of Global.Asax. Data in the Application exists as

long as the worker process exists; this can restart if web.config is modified, the application is idle, or consuming too much RAM. Note that the Application object is specific to a computer and can't be easily shared across web farms.

- The Application Cache is similar to application state. However, the data in the application cache is volatile; ASP.NET manages the cache and removes items when they expire or become invalidated, or when memory runs low. You can also configure application caching to notify your application when an item is removed.

Session state

The IOrder could be instantiated with session scope and injected into the constructor

```
    public class ShoppingController : Controller
    {
        private IOrder order;
        private IOrderModel orderModel;
        private IOrder order;

        public ShoppingController(IProductModel productModel,
                                        IOrderModel orderModel, IOrder order)
        {
            this.productModel = productModel;
            this.orderModel = orderModel;
            this.order = order;
        }

        public ViewResult AddProduct(string id)
        {
            Product product = productModel.SelectById(id);
            order.AddProduct(product, 1);
            return View("Basket", order.LineItems);
        }

        public ViewResult Purchase()
        {
            orderModel.Create(order);
            return View("Purchase");
        }
```

```
namespace WebClient.App_Start
{
    public class UnityConfig
    {
        public static void RegisterTypes(IUnityContainer container)
        {
            container.RegisterType<IProductModel, EfProductModel>(
                                                new TransientLifetimeManager());
            container.RegisterType<IOrderModel, EfOrderModel>(
                                                new TransientLifetimeManager());
            container.RegisterType<IOrder, Order>(new SessionLifetimeManager());
```

SessionLifetimeManager is a user defined LifetimeManager that enables the instance created by the container to be maintained for the duration of the session.

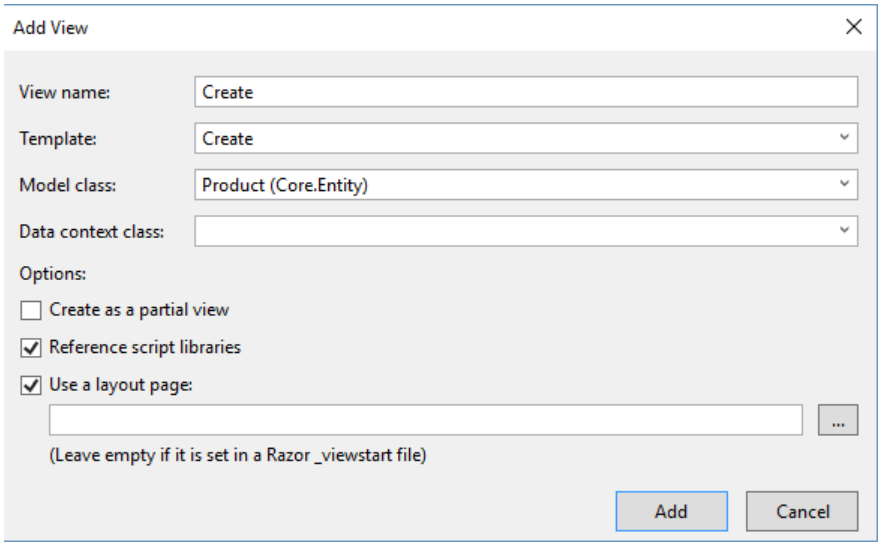
```
namespace WebClient.App_Start
{
    public class SessionLifetimeManager : LifetimeManager
    {
        private string key = Guid.NewGuid().ToString();

        public override object GetValue()
        {
            return HttpContext.Current.Session[key];
        }

        public override void SetValue(object value)
        {
            HttpContext.Current.Session[key] = value;
        }

        public override void RemoveValue()
        {
            HttpContext.Current.Session.Remove(key);
        }
    }
}
```

Validation

Add two Create methods to ShoppingController

```
        public ActionResult Create()
        {
            return View();
        }

        [HttpPost]
        public ActionResult Create(Product product)
        {
            productModel.Create(product);
            return RedirectToAction("List");
        }
```

Then generate the View, selecting “Reference Script Libraries” to add client side validation



ValidationAttribute

```
public class Product
{
    [ScaffoldColumn(true)]
    public string Id { get; set; }
    // Allow uppercase and lowercase characters and spaces.
    [RegularExpression(@"[a-zA-Z'\s]+",
                       ErrorMessage = "Name is not valid.")]
    [Required(ErrorMessage = "Name is required.")]
    public string Name { get; set; }
```

Use code first migrations to update the database to match the model

- Add-Migration NameRequired
- Update-Database

Bootstrap

Bundled with the MVC application template, Bootstrap is a HTML, CSS, and JS framework for developing web applications. Documentation is at getbootstrap.com

CSS

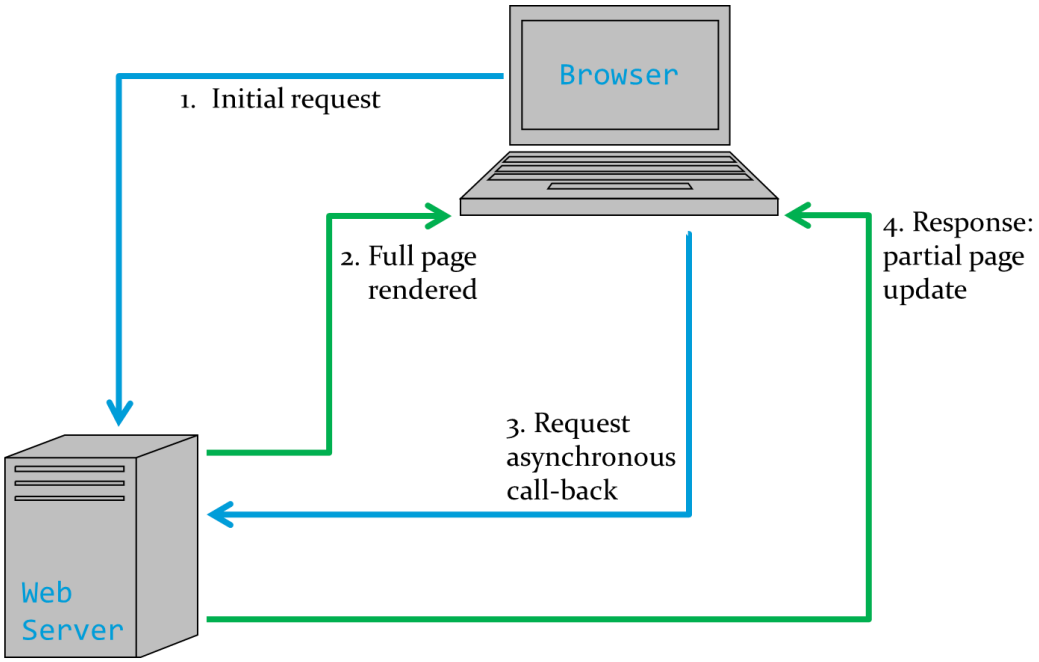
Bundling and Minification

Search box

This passes an HTTP request parameter named “partOfName” to the List method of the ShoppingController

```
@using (Html.BeginForm("List", "Shopping"))
{
    <input name="partOfName" id="search" type="text" />
    <input type="submit" value="Search" />
}

namespace WebClient.Controllers
{
    public class ShoppingController : Controller
    {
        public ViewResult List(string partOfName)
        {
            return View("ProductList",
                partOfName==null ? productModel.Products :
                                        productModel.SelectByName(partOfName));
        }
```

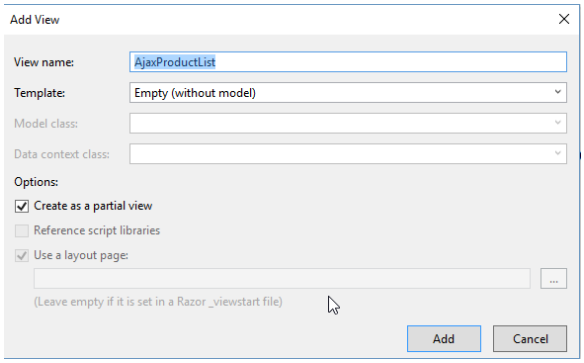
Ajax

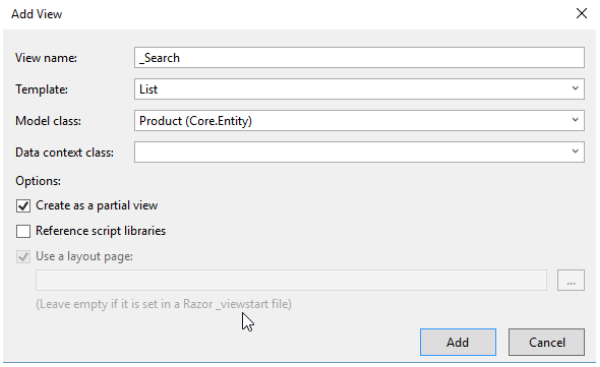


Use NuGet to add the Microsoft.jQuery.Unobtrusive.Ajax library, then add two action methods to the controller

```
namespace WebClient.Controllers
{
    public class ShoppingController : Controller
    {
      public ViewResult AjaxProductList()
        {
            return View();
        }

        public PartialViewResult _Search(string partOfName)
        {
            ICollection<Product> products = productModel.SelectByName(partOfName);
            return PartialView(products);
        }
```

Create two Views





AjaxProductList

```
@using (Ajax.BeginForm("_search", "shopping",
        new AjaxOptions { UpdateTargetId = "searchResults" }))
{
    <input name="partOfName" id="search" type="text" />
    <input type="submit" value="Search" />
}

<div id="searchResults"></div>

@section Scripts {
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"
              type="text/javascript"></script>
    <script src="~/Scripts/search.js" type="text/javascript"></script>
}
```

_search

```
@model IEnumerable<Core.Entity.Product>
<table class="table">
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.CostPrice)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.RetailPrice)
        </td>
        <td>
            @Html.ActionLink("Select", "AddProduct", new { id = item.Id })
        </td>
    </tr>
}
</table>
```

When the user clicks the submit button, the browser sends an asynchronous GET request to the _search action in the ShoppingController class. Ajax helpers, such as BeginForm, depend on the jquery.unobtrusive-ajax script, which can be added with Nuget. This script can be included through the use of @RenderSection in _Layout.cshtml.

jQuery

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. jQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.

While JavaScript provides the load event for executing code when a page is rendered, this event is not triggered until all assets such as images have been completely received. In most cases, the script can be run as soon as the DOM hierarchy has been fully constructed. The handler passed to the ready method is executed after the DOM is ready, so this is usually the best place to attach all other event handlers and run other jQuery code.

```
$( document ).ready(function() {
  // Handler for .ready() called.
});
```

Can be abbreviated to

```
$(function() {
  // Handler for .ready() called.
});
```

```
$(function () {
    $("#search").focus();
    $("form").submit();

    $("#search").keyup(function ()
    {
        $("form").submit();
    });
  }
)
```

`$("form")` selects all form elements on the page
`$("form").first()` selects the first form element on the page
`$("#search")` selects the element with the id “search”

Security

Set up security

1. Create a database named security and add a connection string named DefaultConnection
2. To store the user's name when they register
    a. Add the Name property to ApplicationUser class in Models/IdentityModel.cs
    b. Add the Name property to RegisterViewModel class in Models/AccountViewModel.cs
    c. Add a TextBox for the name in Views/Account/Register.cshtml
    d. In the AccountController's Register method
        i. Set the property value

```
var user = new ApplicationUser { UserName = model.Email,
                        Email = model.Email, Name=model.Name };
```

        ii. Add a new Account object to the database and store in Session State

```
Account account = new Account{Id = model.Email,
                                                Name = model.Name};
IAccountModel accountModel = new EfAccountModel();
accountModel.Create(account);
Session["account"] = account;
```

        iii. In the Login method, retrieve the Account object and store the ShoppingBasket in Session state

```
case SignInStatus.Success:
       IAccountModel accountModel = new EfAccountModel();
       Account account = accountModel[model.Email];
       Session["account"] = account;
```

        iv. Prefix the AddProduct method in ShoppingController with [Authorize] and retrieve Account abject from Session

```
[Authorize]
public ViewResult AddProduct(string id)
{
       order.Account = Session["account"] as Account;
```

3. Run code first migrations to update the database schema

Preventing cross-site request forgery (CSRF) attacks

CSRF exploits the trust that a site has in a user's browser by forcing an end user to execute unwanted actions on a web application in which they're currently authenticated.

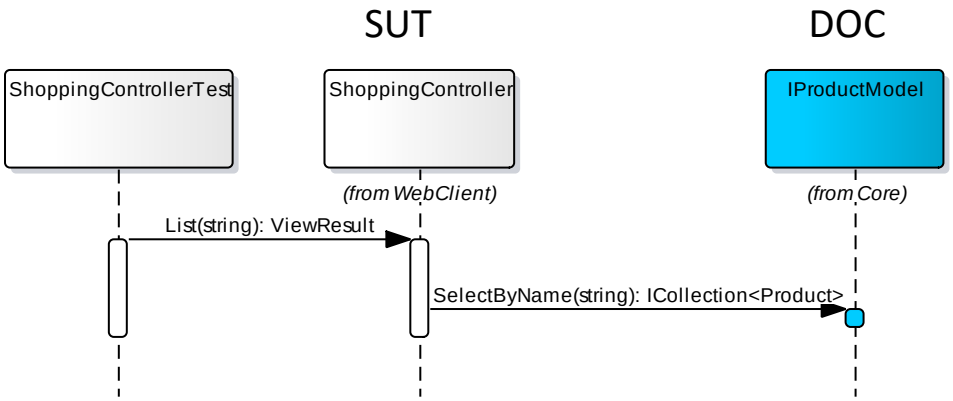
The HtmlHelper.AntiForgeryToken method generates a hidden form field (anti-forgery token) that is validated when the form is submitted. The Html property of the view returns as HtmlHelper object.

The corresponding action method in the controller is prefixed by the ValidateAntiForgeryToken attribute

Unit Tests

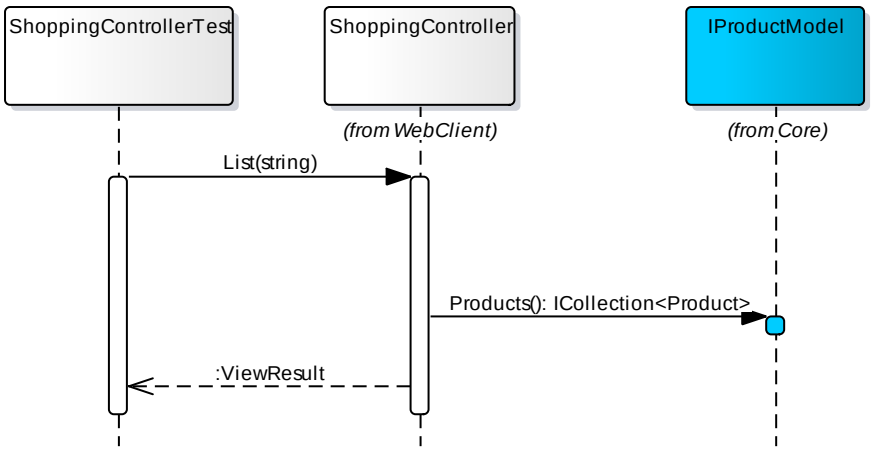
By using Mocks of the Controller's dependencies, the class can be tested in isolation. The phases are:

1. arrange
    a. instantiate the Controller, passing mock objects into the constructor
    b. define how the mock objects should respond to their methods being invoked.
2. act – call the method in the SUT that's being tested
3. assert – verify that the response from the method matches the expected result



The following test verifies that when the ShoppingController's List method is invoked, the IProductModel's SelectByName method is invoked.

```
namespace Store.WebClient.Tests.Controllers
{
    public class ShoppingControllerTest
    {
        private ShoppingController controller;
        private Mock<IProductModel> productModel;
        public ShoppingControllerTest()
        {
            productModel = new Mock<IProductModel>();
            orderModel = new Mock<IOrderModel>();
            order = new Mock<IOrder>();
            // Arrange
            controller = new ShoppingController(
                productModel.Object, orderModel.Object, order.Object);
        }

        [Fact]
        public void List_CallsSelectByNameMethodOfProductModel()
        {
            // Act
            ViewResult result = controller.List("partOfTitle");
            // Assert
            productModel.Verify(pm => pm.SelectByName("partOfTitle"));
        }
```

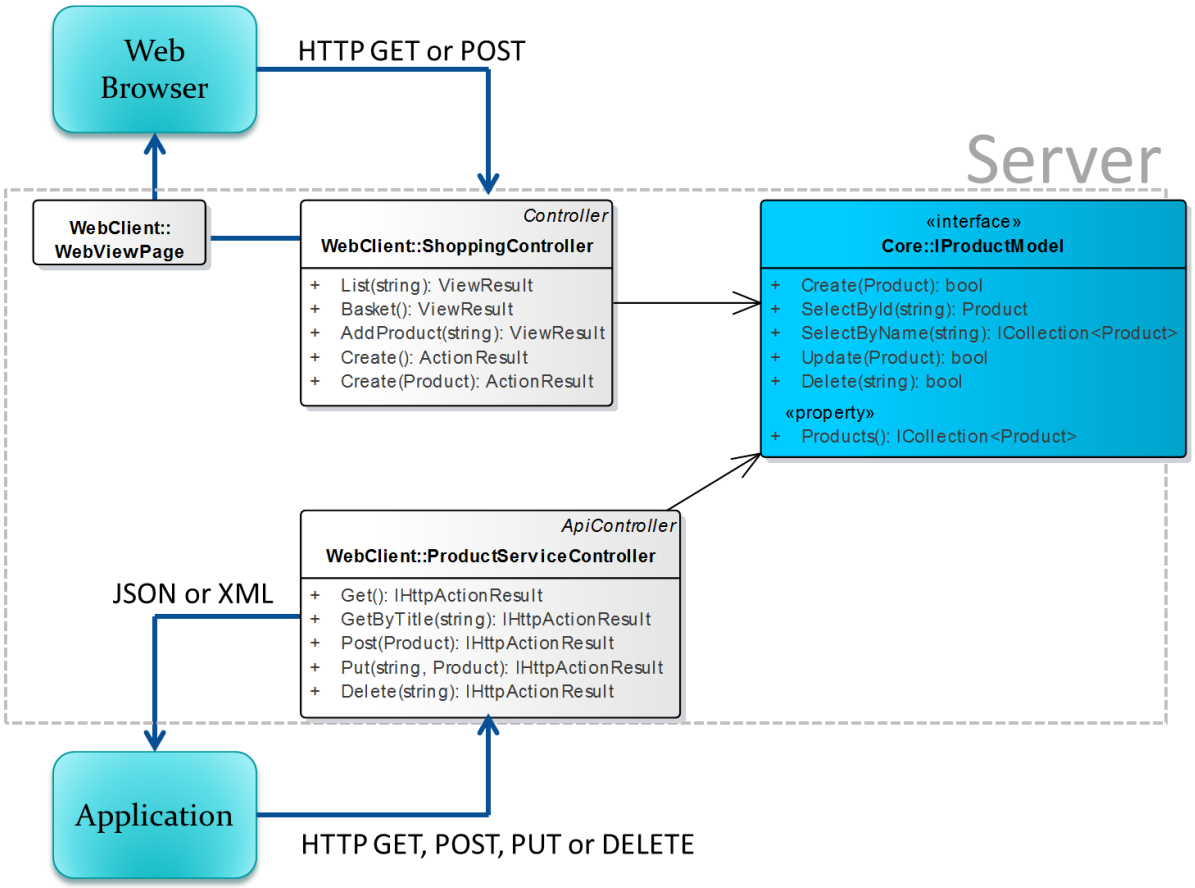


The arrange step of the following test first sets up the mock implementation of the DOC, IProductModel, such that when its Products property is invoked, it returns an ICollection of Product objects. The ShoppingController's List method is then invoked and it's asserted that the Model contained in the ViewResult returned by this method is the collection of Products.

```
public void List_PassesCollectionOfProductsToView()
{
    // Arrange
    Product[] productCollection = {
        new Product("p1", "Dog's Dinner", 1.50),
        new Product("p2", "Knife", 0.60) };

    productModel.Setup(pm => pm.Products).Returns(productCollection);

    // Act
    ViewResult result = controller.List(It.IsAny<string>());

    // Assert
    var products = result.ViewData.Model as ICollection<Product>;
    Assert.Equal(2, products.Count);
}
```

Web API

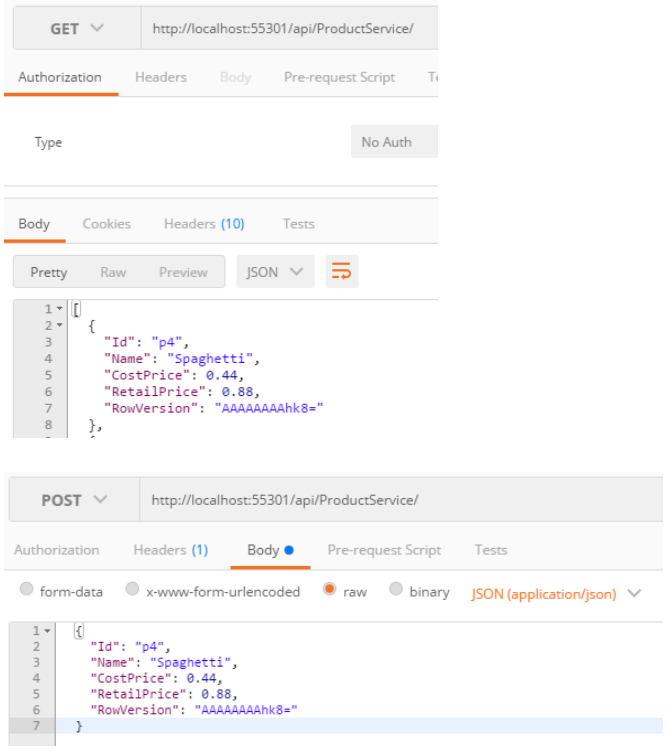
Web API follows the MVC development style but is tailored to writing HTTP REST services. There's a controller class, which derives from ApiController instead of Controller. Rather than associating methods with a URL, the methods match the HTTP method. For example the Get method in ProductController responds to an HTTP GET request. Rather than serving HTML to the client, the server can return objects encoded using either XML or JSON.



URL mappings are configured in App_Start/WebApiConfig.

```
namespace WebClient.Controllers
{
    public class ProductServiceController : ApiController
    {
        private IProductModel productModel = new EfProductModel();

        // GET: api/productService
        public IHttpActionResult Get()
        {
            ICollection<Product> products = productModel.Products;
            //return HTTP status code 200 and serialize films as JSON
            return Ok(products);
        }

        public IHttpActionResult Post([FromBody]Product product)
        {
            bool added = productModel.Create(product);
            return Ok();
        }
```

An HTTP Client application such as Chrome Postman can be used to test web services.



HTTP Response

ApiController methods

2xx: Success - The action was successfully received, understood, and accepted	
Ok	200
Created	201
4xx: Client Error - The request contains bad syntax or cannot be fulfilled	
BadRequest	400
NotFound	404
Conflict	409
5xx: Server Error - The server failed to fulfil an apparently valid request	
InternalServerError	500

The Post method in the ProductController could be improved

1. Return a HTTP created response
2. Handle binding failures
3. Handle exceptions

```
public class ProductServiceController : ApiController
{
    private IProductModel productModel = new EfProductModel();

    // POST: api/productService
    public IHttpActionResult Post([FromBody]Product product)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState); //400
        try
        {
            bool added = productModel.Create(product);
            if (!added)
                return BadRequest("Product already exists");
            //201
            return Created(new Uri(Request.RequestUri + product.Id), product);
        }
        catch (Exception e)
        {
            return InternalServerError(e); //500
        }
    }
```

Routing with attributes

Routing is how Web API matches a URI to an action. For example, a Get request to `api/productService/id/dog` would match the following action

```
// GET: api/productService/id/[p1]
[Route("api/productService/id/{id}")]
public IHttpActionResult Get(string id)
{
    Product product = productModel.SelectById(id);
    if (product == null)
        return NotFound(); //404
    return Ok(product); //200
}
```

Constraints can be applied to parameters; for example `{partOfName:alpha}`

```
// GET: api/productService/[name]
[Route("api/productService/{partOfName:alpha}")]
public IHttpActionResult GetByTitle(string partOfName)
{
    ICollection<Product> products = productModel.SelectByName(partOfName);
    return Ok(products);
}
```

Constraint	Description	Example
alpha	Matches uppercase or lowercase characters	{x:alpha}
bool	Matches a Boolean value.	{x:bool}
datetime	Matches a DateTime value.	{x:datetime}
decimal	Matches a decimal value.	{x:decimal}
double	Matches a 64-bit floating-point value.	{x:double}
float	Matches a 32-bit floating-point value.	{x:float}
guid	Matches a GUID value.	{x:guid}
int	Matches a 32-bit integer value.	{x:int}
long	Matches a 64-bit integer value.	{x:long}
max	Matches an integer with a maximum value.	{x:max(10)}
maxlength	Matches a string with a maximum length.	{x:maxlength(10)}
min	Matches an integer with a minimum value.	{x:min(10)}
minlength	Matches a string with a minimum length.	{x:minlength(10)}
range	Matches an integer within a range of values.	{x:range(10,50)}
regex	Matches a regular expression.	{x:(^\d{3}-\d{3}-\d{4}$)}

Client

```
namespace Core.DataAccess.WebApi
{
    public class WebApiProductModel : IProductModel
    {
        //private string uri = "http://www.sdineen.uk/api/productService/";
        private string uri = "http://localhost:55301/api/productService/";

        public ICollection<Product> SelectByName(string name)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = client.GetAsync(uri + name).Result;
                IEnumerable<Product> products =
                     response.Content.ReadAsAsync<IEnumerable<Product>>().Result;
                return products.ToList();
            }
        }

        public bool Create(Product product)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response =
                                client.PostAsJsonAsync(uri, product).Result;
                return response.StatusCode == HttpStatusCode.Created;
            }
        }
    }
}
```

Integration tests

```
namespace Core.IntegrationTests.WebApi
{
    //test classes in the same collection don't run in parallel
    [Collection("Collection 1")]
    public class WebApiProductModelTest
    {
        [Fact]
        [Trait("Core.WebApi", "Integration Test")]
        public void SelectByName_ReturnsProducts()
        {
            //arrange
            WebApiProductModel sut = new WebApiProductModel();
            //act
            ICollection<Product> products = sut.SelectByName("");
            //assert
            Assert.NotNull(products);
            Assert.True(products.Count > 0);
        }
```

WPF

Introduction

Windows Presentation Foundation is a system for building desktop client applications for Windows. One feature that distinguishes it from the earlier Windows Forms is the separation of presentation from application logic. XAML (an XML-based language) describes the layout of the components on the user interface, while a C# code-behind class handles events. The default XAML namespace brings in CLR namespaces in the PresentationFramework assembly including System.Windows, System.Windows.Controls and System.Windows.Data. The x namespace abbreviation brings in XAML language features, including the Class directive for joining markup and code-behind.

Add a WPF project to the solution and drag the TextBox, Button and TextBlock components from the toolbox.

```
<Window x:Class="WinClient.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WinClient"
        mc:Ignorable="d"
        Title="MainWindow" Height="300" Width="300">
    <Grid>
        <TextBox x:Name="textBox" HorizontalAlignment="Left" Height="23"
              Margin="26,24,0,0" TextWrapping="Wrap" Text="1000000"
              VerticalAlignment="Top" Width="120"/>
        <Button x:Name="button" Content="Start" HorizontalAlignment="Left"
              Margin="26,68,0,0" VerticalAlignment="Top" Width="75"/>
        <Label x:Name="label" Content="Label" HorizontalAlignment="Left"
              Margin="26,112,0,0" VerticalAlignment="Top"/>
    </Grid>
</Window>
```

Events

Events enable an object to notify other objects when something of interest occurs.

To respond to the button being clicked, add the following to the constructor in the code behind class:

```
button.Click += TAB
```

This will generate a method that will be called when the event is raised.

Click is an Event field in the Button class, which would be written in C# as follows:

```
public event RoutedEventHandler Click;
```

Events are invoked from within the class that declares them:

```
Click(this, e)
```

Delegates

Delegates enable methods to be passed as arguments to other methods. A delegate represents a method with a particular parameter list and return type. For example, RoutedEventHandler is a delegate associated with a method that has a void return type and takes two arguments of type Object and RoutedEventArgs. The following method matches the signature of the delegate:

```
private void Button_Click (object sender, RoutedEventArgs e)
{
}
```

Like classes, delegates can be instantiated. The argument to a delegate's constructor is a method name. For example:

```
RoutedEventHandler buttonHandler = new RoutedEventHandler(Button_Click);
```

This delegate instance can be associated with the Click event of the button

```
button.Click += buttonHandler;
```

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Stopwatch sw = new Stopwatch();
    sw.Start();
    int limit = Convert.ToInt32(textBox.Text);
    int result = (from n in Enumerable.Range(2, limit)
                  where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
                  select n).Count();
    sw.Stop();
    textBlock.Text = $"{result} prime numbers. Calculated in
                                    {Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
}
```

Improvements

- Layout that adapts to different screen sizes
- Centrally defined styles defining component properties
- Asynchronous methods

Layouts

WrapPanel

```
<WrapPanel Orientation="Vertical">
    <TextBox Name="textBox" Width="100"/>
    <Button Name="button" Content="Start" Margin="0,5,0,5"/>
    <Label Name="label" Content="Label" />
</WrapPanel>
```

StackPanel

```
<StackPanel Orientation="Vertical">
    <TextBox Name="textBox" Width="100" HorizontalAlignment="Left"/>
    <Button Name="button" Content="Start" Width="100" HorizontalAlignment="Left"
                                                          Margin="0,5,0,5"/>
    <Label Name="label" Content="Label" Width="100" HorizontalAlignment="Left" />
</StackPanel>
```

DockPanel

```
<DockPanel>
    <TextBox DockPanel.Dock="Top" Name="textBox" Width="100"
                                                    HorizontalAlignment="Left"/>
    <Label DockPanel.Dock="Bottom" Name="label" Content="Label" Width="100"
                                                    HorizontalAlignment="Left"/>
    <Button DockPanel.Dock="Left" Name="button" Content="Start" Margin="0,5,0,5"/>
    <Rectangle/>
</DockPanel>
```

Grid

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
    <TextBox Grid.Row="0" Name="textBox" Width="100" Height="30"
                                                    HorizontalAlignment="Left" />
    <Button Grid.Row="1" Name="button" Content="Start"
                            Width="100" Height="30" HorizontalAlignment="Left" />
    <Label Grid.Row="2" Name="label" Content="Label" Width="100" Height="30"
                                                    HorizontalAlignment="Left" />
</Grid>
```

Styles

Styles facilitate applying standard formatting to a set of controls. They can be defined for a window or the whole application.

```
<Window.Resources>
    <Style TargetType="Button">
        <Setter Property="HorizontalAlignment" Value="Left" />
        <Setter Property="Width" Value="100" />
    </Style>
</Window.Resources>
```

Asynchronous and concurrent methods

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        button.Click += Button_Click;
    }

    private async void Button_Click(object sender, RoutedEventArgs e)
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();
        int result = await CalculatePrimesAsync(limit);
        sw.Stop();
        label.Content = $"{result} prime numbers. Calculated in {
                        Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
    }

    public async Task<int> CalculatePrimesAsync(int limit)
    {
        Func<int> func = () => (
           from n in Enumerable.Range(2, limit).AsParallel()
           where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
```

```
            select n).Count();
        int result = await Task.Run(func); //returns Task<int>
        return result;
    }
```

Task.Run queues the specified work to run on the thread pool and returns a proxy for the task returned by function.

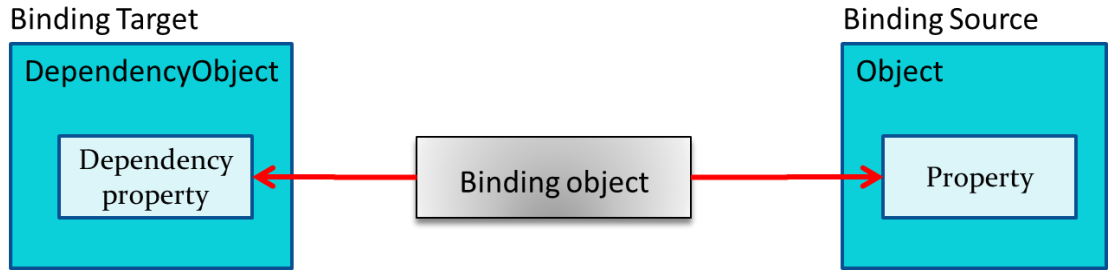
The await operator is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes. Methods using await must be modified by the async keyword.

Products

Add a new window with a Label and a TextBox. Double click the TextBox to generate the event handler.

```
<Window x:Class="WpfApplication1.ProductList"
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Label Content="Search" />
            <TextBox Name="SearchBox" Width="100"
                     TextChanged="SearchBox_TextChanged" />
        </StackPanel>
        <DataGrid>
        </DataGrid>
    </DockPanel>
</Window>
```

Data Binding




```
{Binding Path=PropertyName}
```

A binding object connects the properties of binding target object to a data source (for example, a database, an XML file, or any object that contains data). The above XAML markup extension syntax can be used to build a Binding object and set its Path property.

```
        <DataGrid Grid.Row="1" AutoGenerateColumns="False"
                  GridLinesVisibility="None" ItemsSource="{Binding}" >
            <DataGrid.Columns>
                <DataGridTextColumn Binding="{Binding Name}" Header="Name"/>
            </DataGrid.Columns>
        </DataGrid>
```

Set the DataGrid's DataContext in the code behind class

```
namespace WpfApplication1
{
    public partial class ProductList : Window
    {
        private WebApiProductModel model = new WebApiProductModel();
        public ProductList()
        {
            InitializeComponent();
            SearchBox_TextChanged(null, null);
            SearchBox.Focus();
        }

        private void SearchBox_TextChanged(object sender, TextChangedEventArgs e)
        {
            ICollection<Product> products = model.SelectByName(SearchBox.Text);
            DataContext = products;
        }
```

Asynchronous methods

Integration tests for data access methods

```
namespace Core.IntegrationTests.WebApi
{
    //test classes in the same collection don't run in parallel
    [Collection("Collection 1")]
    public class WebApiProductModelTest
    {
        //set up database
        public WebApiProductModelTest()
        {
        }

        [Fact]
        [Trait("Core.WebApi", "Integration Test")]
        public async void SelectByNameAsync_ReturnsProducts()
        {
            //arrange
            WebApiProductModel sut = new WebApiProductModel();
            //act
            ICollection<Product> products = await sut.SelectByNameAsync("");
            //assert
            Assert.NotNull(products);
            Assert.True(products.Count > 0);
        }

        [Fact]
        [Trait("Core.WebApi", "Integration Test")]
        public async void CreateAsync_AddsProductToDatabase()
        {
            //arrange
            WebApiProductModel sut = new WebApiProductModel();
            Product product = new Product("p14", "Hammer", 2.20, 4.40);
            //act
            bool created = await sut.CreateAsync(product);
            //assert
            Assert.True(created);
        }
    }
}
```

Data access methods

```
namespace Core.DataAccess.WebApi
{
    public class WebApiProductModel
    {
        private string uri = "http://localhost:55301/api/productService/";

        public async Task<ICollection<Product>> SelectByNameAsync(string name)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.GetAsync(uri + name);
                IEnumerable<Product> products =
                    await response.Content.ReadAsAsync<IEnumerable<Product>>();
                return products.ToList();
            }
        }

        public async Task<bool> CreateAsync(Product product)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.PostAsJsonAsync(uri,
                                                                            product);
                return response.StatusCode == HttpStatusCode.Created;
            }
        }
    }
}
```

Calling async methods

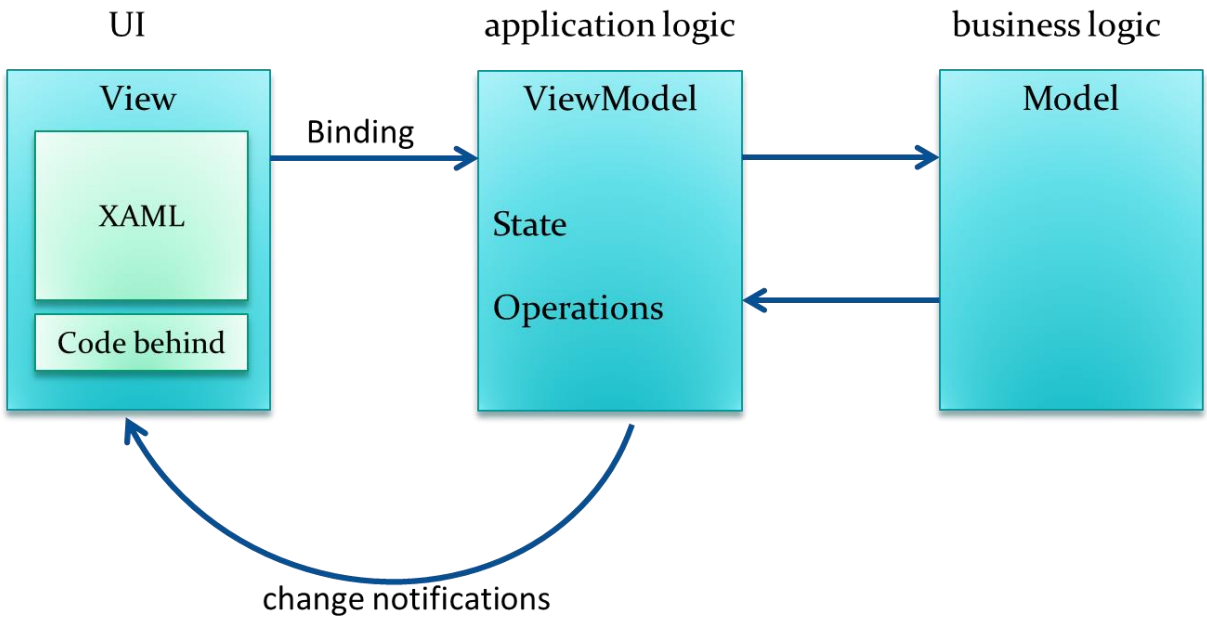
```
namespace WpfApplication1
{
    public partial class ProductList : Window
    {
        private WebApiProductModel model = new WebApiProductModel();
        public ProductList()
        {
            InitializeComponent();
            SearchBox_TextChangedAsync(null, null);
            SearchBox.Focus();
        }

        private async void SearchBox_TextChangedAsync(object sender,
                                                      TextChangedEventArgs e)
        {
            ICollection<Product> products =
                                    await model.SelectByNameAsync(SearchBox.Text);
            DataContext = products;
        }
    }
}
```

WPF MVVM

Products

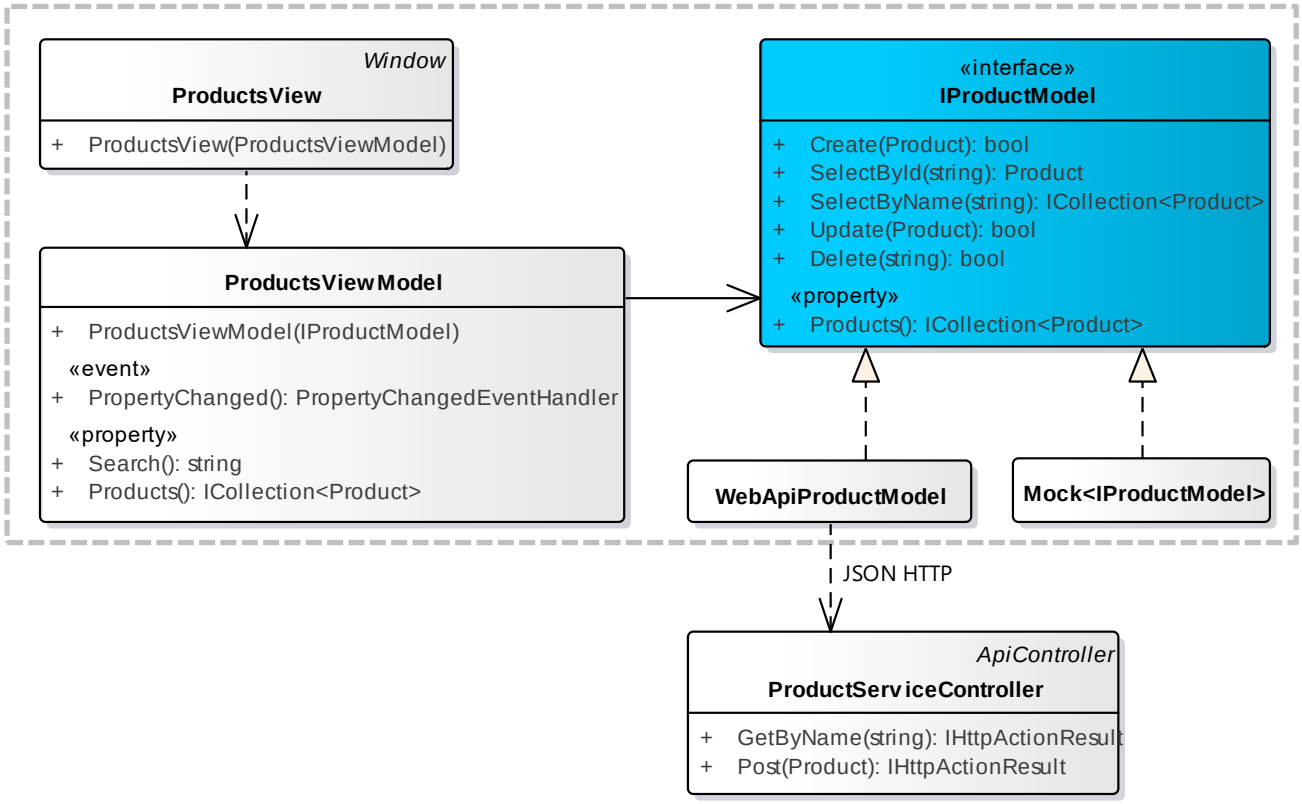
overview

The Model-View-ViewModel (MVVM) pattern facilitates the separation of an application's business and presentation logic from its user interface, making the application easier to test and maintain.



The ViewModel is an abstraction of the View. UI components are bound to properties in the ViewModel. By implementing the INotifyPropertyChanged interface, the ViewModel notifies to the View about changes to its properties. The ViewModel can also contain operations, handling component events.

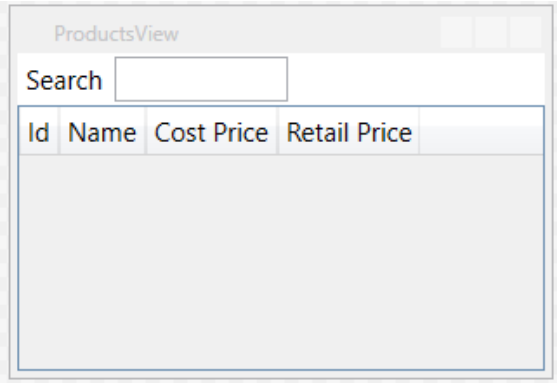
UML



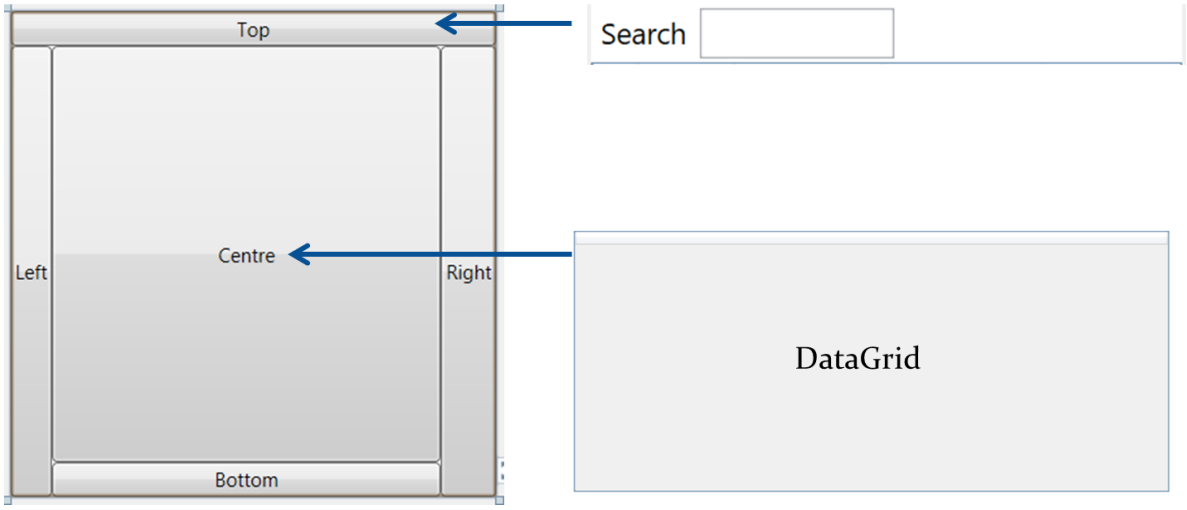
View

Components can be arranged in the window using layout controls, which can be nested.

Canvas	position defined relative to top left of canvas
DockPanel	align content to top, bottom, left right
Grid	arranges content within a grid
StackPanel	content placed in a single row or column
WrapPanel	wraps content to next row or column
TabControl	multiple items share same space on the screen



For example, to build this UI, a DataGrid could be placed in the centre region of a DockPanel, and a StackPanel containing TextBlock and TextBox components placed in the north region.




```
<Window x:Class="WinClient.Products.View.ProductsView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <DockPanel>
        <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
            <Label Content="Search" />
            <TextBox Name="SearchBox" />
        </StackPanel>
        <DataGrid >
        </DataGrid>
    </DockPanel>
</Window>
```

Binding

The DataContext for the window will be the ViewModel. The TextBox is bound to the ViewModel's Search property. The Mode and UpdateSourceTrigger properties specify that the TextBox will set the ViewModel property as text is typed into it.

```
<StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
    <Label Content="Search" />
    <TextBox Name="SearchBox" Text="{Binding Search, Mode=TwoWay,
                                     UpdateSourceTrigger=PropertyChanged}" />
</StackPanel>
```

The DataGrid's ItemsSource property is bound to the ViewModel's Products property, which could be a collection of Product objects. This contains DataGridTextColumns, which are bound to properties of the Product.

```
<DataGrid ItemsSource="{Binding Products}"
    GridLinesVisibility="None"
    AutoGenerateColumns="False" >
    <DataGrid.Columns>
       <!-- Binding Property sets the binding that associates
       the column with a property in the data source   -->
       <DataGridTextColumn Binding="{Binding Id}" Header="Id"/>
       <DataGridTextColumn Binding="{Binding Name}" Header="Name"/>
```

```
        <DataGridTextColumn Binding="{Binding CostPrice}" Header="Cost Price"/>
        <DataGridTextColumn Binding="{Binding RetailPrice}" Header="Retail Price"/>
    </DataGrid.Columns>
</DataGrid>
```

View code behind

This sets the DataContext for the window as the ViewModel

```
public partial class ProductsView : Window
{
    public ProductsView(ProductsViewModel viewModel)
    {
        InitializeComponent();
        DataContext = viewModel;
        SearchBox.Focus();
    }
}
```

ViewModel

By implementing INotifyPropertyChanged, subscribers can be notified when the properties change

```
public class ProductsViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    private WebApiProductModel productModel;

    private string search;
    private ICollection<Product> products;

    public ProductsViewModel(WebApiProductModel productModel)
    {
        this.productModel = productModel;
        Products = productModel.SelectByName(Search);
    }

    public string Search
    {
        get { return search; }
        set
        {
            search = value;
            Products = productModel.SelectByName(Search);
        }
    }

    public ICollection<Product> Products  //Bound to DataGrid ItemsSource
    {
        get { return products; }
        set
        {
            products = value;
            // If DataContext not set, there will be no subscribers to the event
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs("Products"));
        }
    }
}
```

Startup

Instead of the StartupUri property in App.xaml, add an OnStartup event handler to App.xaml.cs

```
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
            WebApiProductModel model = new WebApiProductModel();
            ProductsViewModel viewModel = new ProductsViewModel(model);
            ProductsView view = new ProductsView(viewModel);
            view.Show();
        }
    }
```

Call Asynchronous methods

```
namespace WinClient.Products.ViewModel
{
    public class ProductsViewModel : INotifyPropertyChanged
    {
        private WebApiProductModel productModel;
        public ProductsViewModel(WebApiProductModel productModel)
        {
            this.productModel = productModel;
            LoadData();
        }

        private async void LoadData()
        {
            Products = await productModel.SelectByNameAsync(Search);
        }
    }
}
```

Dependency injection

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is the passing of a dependency to a dependent. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Benefits of DI include

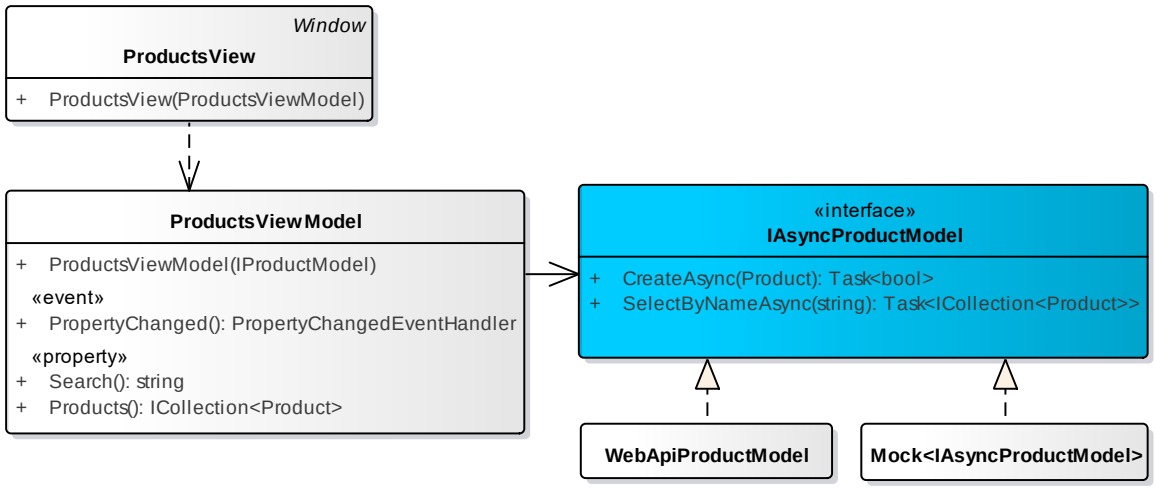
- Maintainability
- Testability (using mock implementations of injected interfaces)
- Flexibility and Extensibility
- Loose Coupling (reducing the number of dependencies between the application's components)

Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. Other popular containers include Ninject and Autofac.

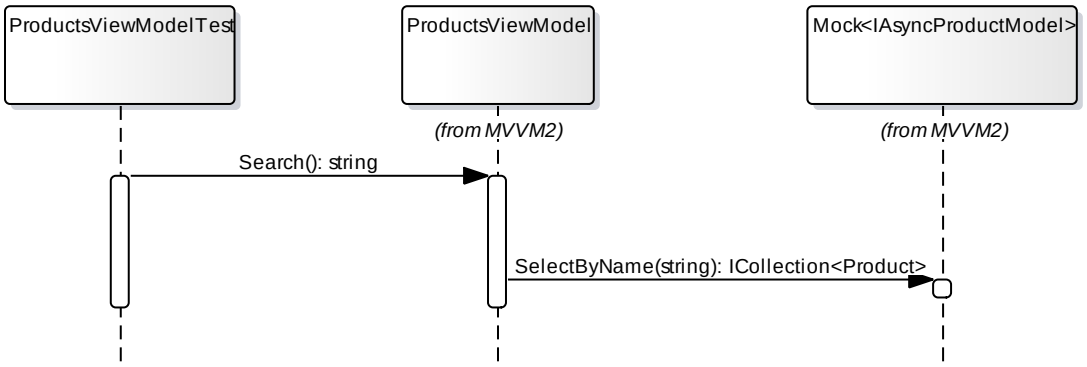
Unity's RegisterType method associates an implementing class with an interface. The Resolve method uses constructor injection to instantiate the dependencies.

```
public partial class App : Application
{
    protected override void OnStartup(StartupEventArgs e)
    {
        base.OnStartup(e);
        IUnityContainer container = new UnityContainer();
        container.Resolve<ProductsView>().Show();
    }
}
```

Unit tests



Verify that the DOC's SelectByName method is called when the SUT's Search property is set.




```
namespace WinClient.Tests.ViewModel
{
    public class ProductsViewModelTest
    {
        private Mock<IAsyncProductModel> doc = new Mock<IAsyncProductModel>();
        private ProductsViewModel sut;

        public ProductsViewModelTest()
        {
            sut = new ProductsViewModel(doc.Object);
        }

        [Fact]
        [Trait("WinClient", "Unit Test")]
        public void SearchPropertySetter_ShouldCallSelectByNameAsync()
        {
            // Act
            sut.Search = "something";
            // Assert
            doc.Verify(m => m.SelectByNameAsync("something"));
        }
```

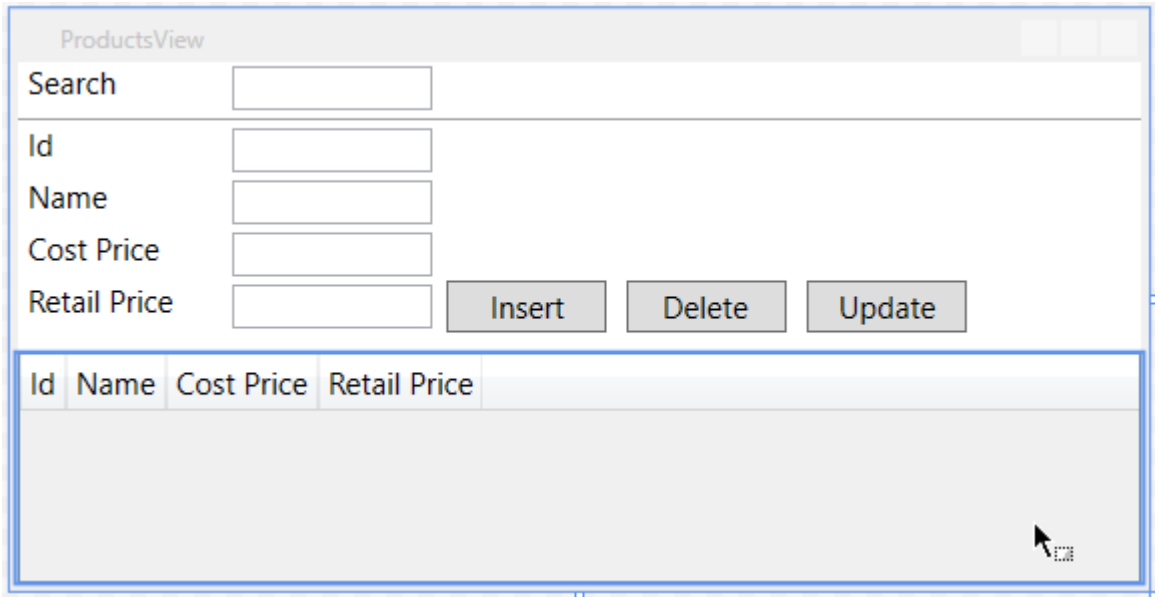
Extract the IAsyncProductModel interface from the WebApiProductModel class. Change the ProductsViewModel constructor to take an interface argument. If using Unity DI, associate this interface with a class in App.xaml.cs

```
container.RegisterType<IAsyncProductModel, WebApiProductModel>();
```

Products MVVM continued

UI

The application could be embellished by enabling inserts, updates and deletes to the database via the web service



View

The additional TextBoxes and Buttons are bound to properties in the ViewModel.

```
            <Separator/>
            <StackPanel Orientation="Horizontal">
                <Label Content="Id" />
                <TextBox Text="{Binding Id}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Content="Name" />
                <TextBox Text="{Binding Name}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal">
                <Label Content="Cost Price" />
                <TextBox Text="{Binding CostPrice}" />
            </StackPanel>
            <StackPanel Orientation="Horizontal" Margin="0,0,0,10">
                <Label Content="Retail Price" />
                <TextBox Text="{Binding RetailPrice}" />
                <Button Command="{Binding AddProductCommand}" Content="Insert" />
                <Button Command="{Binding DeleteProductCommand}" Content="Delete" />
                <Button Command="{Binding UpdateProductCommand}" Content="Update" />
            </StackPanel>
```

Styles

Much like CSS styles for an HTML page, the properties of XAML controls can be configured such that all the components of a specified type are rendered in the same way. The following style elements set the FontSize, MinWidth and Margin properties for all the Label, TextBox, DataGridCell and DataGridColumnHeader controls in the Window.

```
<Window x:Class="WinClient.Products.View.ProductsView1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Window.Resources>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="15" />
        </Style>
        <Style TargetType="TextBox">
            <Setter Property="FontSize" Value="15" />
            <Setter Property="MinWidth" Value="100" />
            <Setter Property="Margin" Value="2" />
        </Style>
        <Style TargetType="DataGridCell">
            <Setter Property="FontSize" Value="15" />
        </Style>
        <Style TargetType="DataGridColumnHeader">
            <Setter Property="FontSize" Value="15" />
        </Style>
    </Window.Resources>
```

ViewModel

```
namespace WinClient.Products.ViewModel
{
    public class ProductsViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private IAsyncProductModel productModel;

        //Backing fields for properties bound to TextBoxes
        private string id;
        private string name;
        private double? costPrice;
        private double? retailPrice;
        private string search;
        //Backing fields for properties bound to DataGrid
        private ICollection<Product> products;
        private Product selectedProduct;

        public ProductsViewModel(IAsyncProductModel productModel)
        {
            this.productModel = productModel;
            LoadCommands();
            LoadData();
        }

        private void LoadCommands()
        {
            AddProductCommand = new CustomCommand(AddProduct, CanAddProduct);
            DeleteProductCommand = new CustomCommand(DeleteProduct, CanDeleteProduct);
            UpdateProductCommand = new CustomCommand(UpdateProduct, CanUpdateProduct);
        }

        private async void LoadData()
        {
            Products = await productModel.SelectByNameAsync(Search);
        }

        #region Bound Properties
        public string Search
        {
            get { return search; }
            set
            {
                search = value;
                LoadData();
            }
        }
        public Product SelectedProduct //Bound to DataGrid SelectedItem
        {
            get { return selectedProduct; }
            set
            {
                selectedProduct = value;
            }
        }



        //Operations
        public CustomCommand AddProductCommand { get; private set; }
```

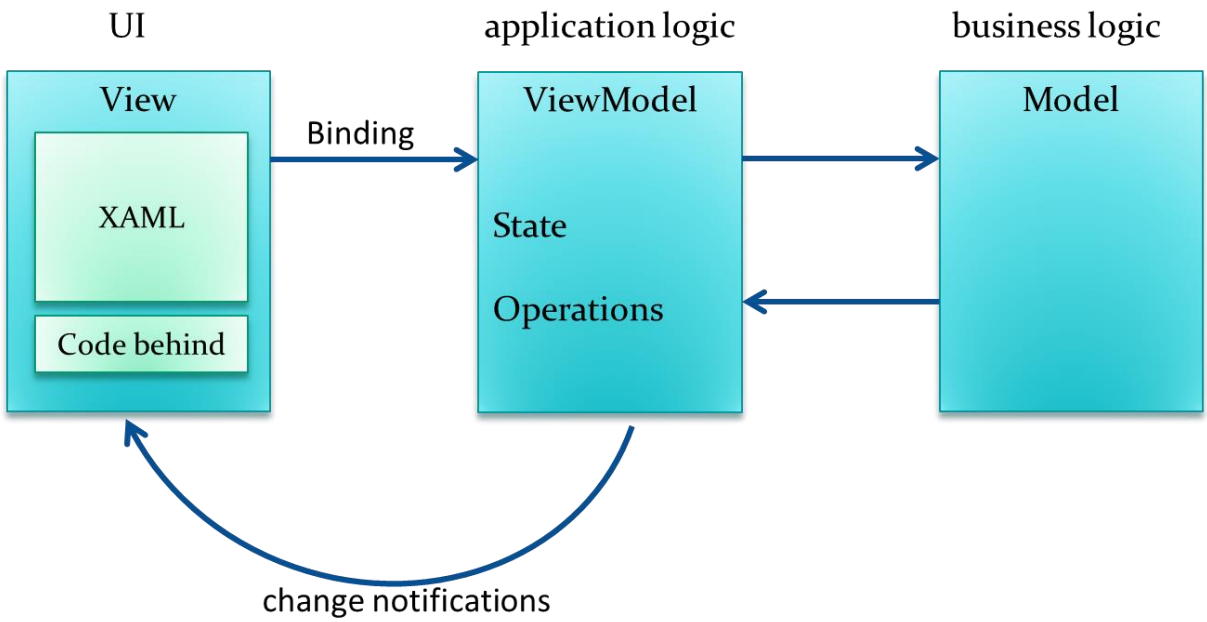
```
public CustomCommand DeleteProductCommand { get; private set; }
public CustomCommand UpdateProductCommand { get; private set; }

private async void AddProduct(object obj)
{
    Product product = new Product(Id, Name, CostPrice.GetValueOrDefault(),
                                          RetailPrice.GetValueOrDefault());
    await productModel.CreateAsync(product);
    //continues when awaitable Task returns
    LoadData();
}

private bool CanAddProduct(object obj)
{
    //determines whether button is enabled
    return Id != null && Name != null && CostPrice.HasValue
                                          && RetailPrice.HasValue;
}

private async void UpdateProduct(object obj)
{
    //Pass the SelectedProduct to the IProductModel's
    //UpdateAsync method
}


//the other properties bound to components in the View aren't shown
```

Model

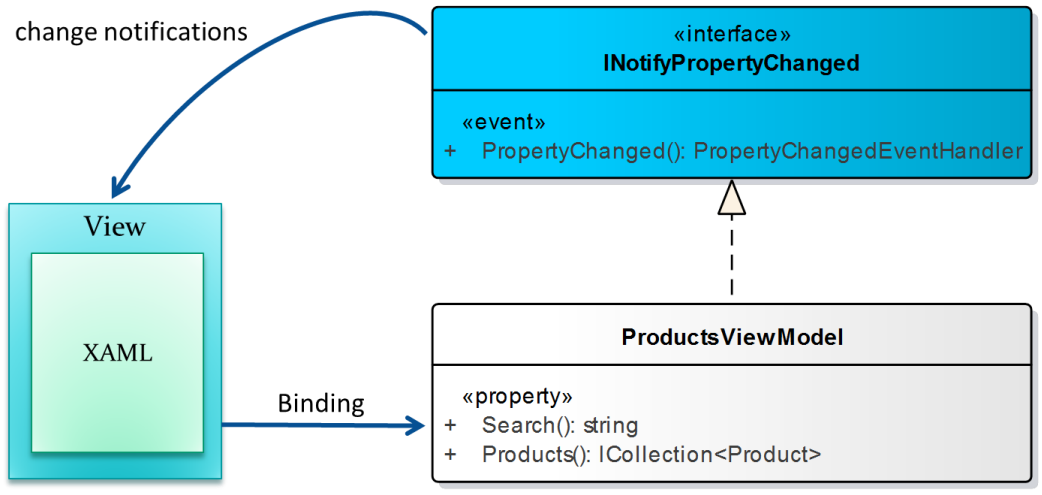
```
namespace Core.DataAccess.WebApi
{
    public interface IAsyncProductModel
    {
        Task<ICollection<Product>> SelectByNameAsync(string name);
        Task<bool> CreateAsync(Product product);
        Task<Product> SelectByIdAsync(string id);
        Task<bool> UpdateAsync(Product product);
        Task<bool> DeleteAsync(string id)
    }

    public class WebApiProductModel : IProductModel, IAsyncProductModel
    {
        //private string uri = "http://www.sdineen.uk/api/productService/";
        private string uri = "http://localhost:55301/api/productService/";

        public async Task<Product> SelectByIdAsync(string id)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.GetAsync(uri + "id" + id);
                return await response.Content.ReadAsAsync<Product>();
            }
        }

        public async Task<bool> UpdateAsync(Product product)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response =
                            await client.PutAsJsonAsync(uri + product.Id, product);
                return response.StatusCode == HttpStatusCode.OK;
            }
        }

        public async Task<bool> DeleteAsync(string id)
        {
            using (HttpClient client = new HttpClient())
            {
                HttpResponseMessage response = await client.DeleteAsync(uri + id);
                return response.StatusCode == HttpStatusCode.OK;
            }
        }
    }
}
```

Primes MVVM

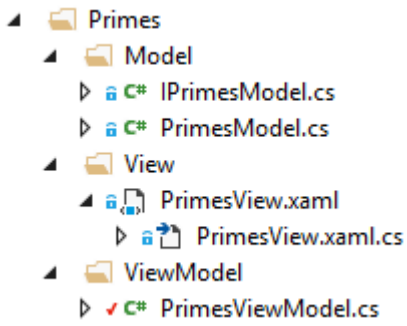
The Model-View-ViewModel (MVVM) pattern facilitates the separation of an application's business and presentation logic from its user interface, making the application easier to test and maintain.



The ViewModel is an abstraction of the View. UI components are bound to properties in the ViewModel. By implementing the INotifyPropertyChanged interface, the ViewModel notifies to the View about changes to its properties. The ViewModel can also contain operations, handling component events.



Folders



Model

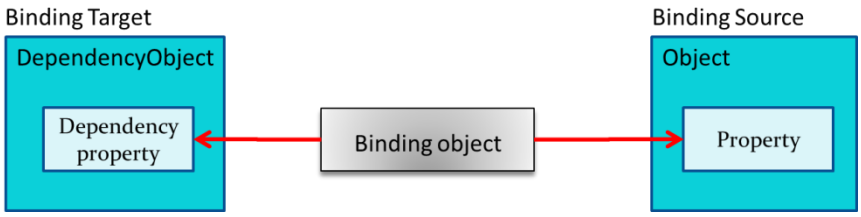
```
namespace WinClient.MVVM.Model.Primes
{
    public class PrimesModel
    {
        public int Count(int max)
        {
            return (from n in Enumerable.Range(2, max)
                    where Enumerable.Range(2, (int)Math.Sqrt(n)-1).All(i => n % i > 0)
                    select n).Count();
        }

        public async Task<int> CountAsync(int max)
        {
            Func<int> func = () =>
                 (from n in Enumerable.Range(2, max)//.AsParallel()
                  where Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)
                  select n).Count();
            int result = await Task.Run(func);
            return result;
        }
    }
}
```

Task.Run queues the specified work to run on the thread pool and returns a proxy for the task returned by function.

The await operator is applied to a task in an asynchronous method to suspend the execution of the method until the awaited task completes. Methods using await must be modified by the async keyword.

Extract the interface for the above class.

View



A binding object connects the properties of binding target object to a data source (for example, a database, an XML file, or any object that contains data). The XAML markup extension syntax enclosed in braces:

```
<TextBlock Text="{Binding Result}"
```

is used to build a Binding object. The default source of bindings is the DataContext property, which can be set in the code-behind class. The constructor takes a PrimesViewModel argument, which can be generated.

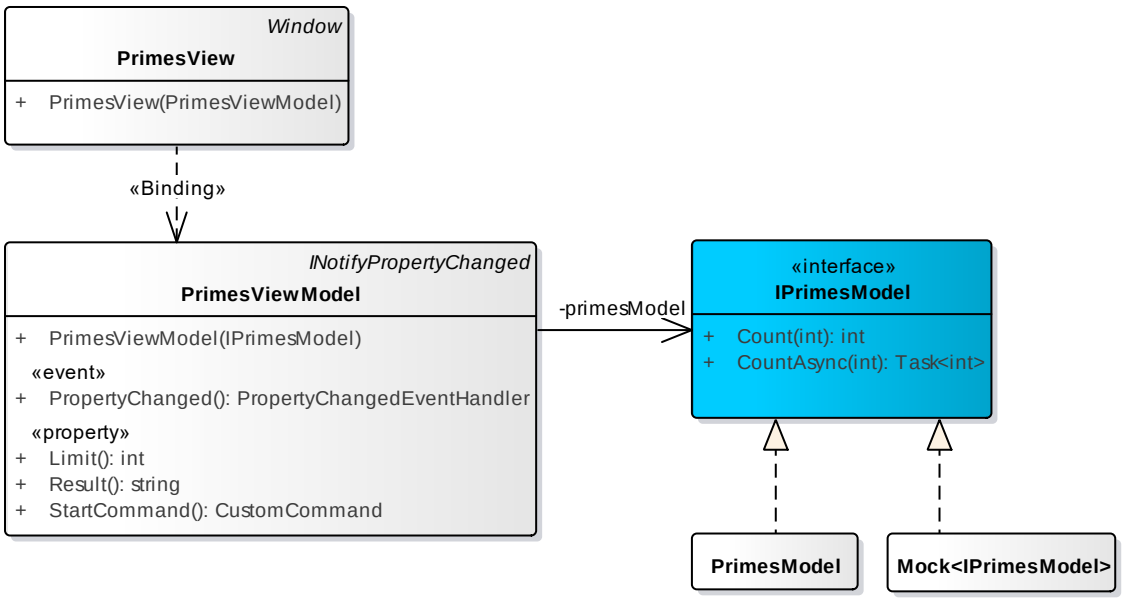
```
namespace WinClient.MVVM.View
{
    public partial class PrimesView : Window
    {
        public PrimesView(PrimesViewModel viewModel)
        {
            InitializeComponent();
            DataContext = viewModel;
        }
    }
}
```

The TextBox.Text property has a default UpdateSourceTrigger value of LostFocus. Changing this to PropertyChanged will cause the source to be updated as text is typed into the TextBox.

```
    <Grid>
        <TextBox Text="{Binding Limit, UpdateSourceTrigger=PropertyChanged}"
        <Button Command="{Binding StartCommand}"
        <TextBlock Text="{Binding Result}"
    </Grid>
```

ViewModel

The View binds to properties in the ViewModel named Limit, Result and StartCommand


Window
PrimesView
+ PrimesView(PrimesViewModel)
«Binding»
INotifyPropertyChanged
PrimesViewModel
+ PrimesViewModel(IPrimesModel)
«event»
+ PropertyChanged(): PropertyChangedEventHandler
«property»
+ Limit(): int
+ Result(): string
+ StartCommand(): CustomCommand
-primesModel
«interface»
IPrimesModel
+ Count(int): int
+ CountAsync(int): Task<int>
PrimesModel
Mock<IPrimesModel>

```
namespace WinClient.MVVM.ViewModel
{
    public class PrimesViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        private IPrimesModel primesModel;

        private int limit;
        private string result;

        public PrimesViewModel(IPrimesModel primesModel)
        {
            this.primesModel = primesModel;
        }

        public int Limit
        {
            get { return limit; }
            set
            {
                limit = value;
                LoadData();
            }
        }

        public string Result
        {
            get { return result; }
            set
            {
                result = value;
                // Notify listeners <TextBlock Text="{Binding Result}"
                PropertyChanged(this, new PropertyChangedEventArgs("Result"));
            }
        }

        private async void LoadData()
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            int count = await primesModel.CountAsync(Limit);
            sw.Stop();
            Result = $"{result} prime numbers. Calculated in
                            {Math.Round(sw.Elapsed.TotalSeconds, 2)} seconds";
        }
    }
}
```

Startup

Since dependencies need to be passed into the View and ViewModel's constructors, the View is instantiated in the App class. Take out the StartupUri property in App.xaml.

```
namespace WinClient
{
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            PrimesModel model = new PrimesModel();
            PrimesViewModel viewModel = new PrimesViewModel(model);
            PrimesView view = new PrimesView(viewModel);
            view.Show();
        }
    }
}
```

Dependency injection

Dependency injection is a software design pattern that implements inversion of control for resolving dependencies. An injection is the passing of a dependency to a dependent. Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Benefits of DI include

- Maintainability
- Testability (using mock implementations of injected interfaces)
- Flexibility and Extensibility
- Loose Coupling (reducing the number of dependencies between the application's components)

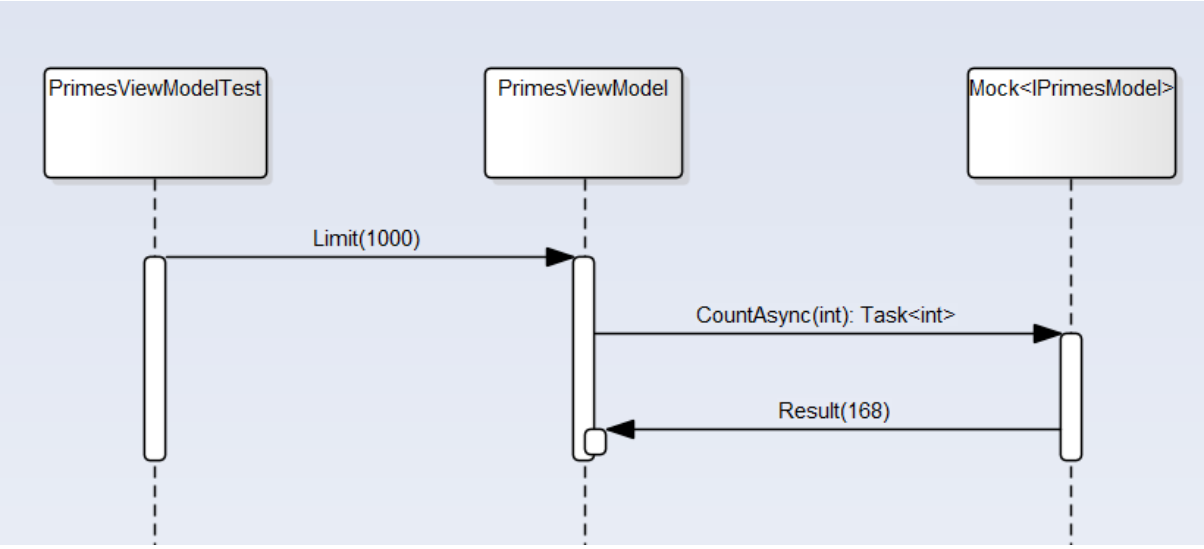
Unity is a lightweight, extensible dependency injection container that supports interception, constructor injection, property injection, and method call injection. Other popular containers include Ninject and Autofac.

Unity's RegisterType method associates an implementing class with an interface. The Resolve method uses constructor injection to instantiate the dependencies.

```
namespace WinClient
{
    public partial class App : Application
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);

            IUnityContainer container = new UnityContainer();
            container.RegisterType<IPrimesModel, PrimesModel>();
            container.Resolve<PrimesView>().Show();        }
    }
}
```

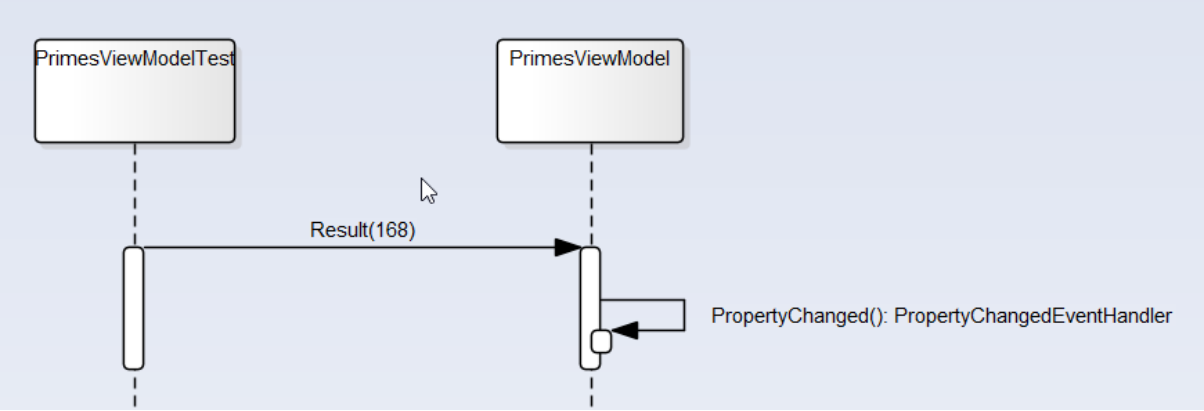
Unit tests

A Mock instance of IPrimesModel can be passed into the PrimeViewModel's constructor, enabling verification of interactions between the system under test and its dependency. For example, when the ViewModel's Limit property is set, the Model's CountAsync method should be called, and the return value assigned to the ViewModel's Result Property.




```
namespace WinClient.Tests
{
    public class PrimesViewModelTest
    {
        private Mock<IPrimesModel> primesModelMock = new Mock<IPrimesModel>();
        private PrimesViewModel viewModel;

        private bool eventRaised;
        private PropertyChangedEventArgs eventArgs;

        //called before each test method
        public PrimesViewModelTest()
        {
            primesModelMock.Setup(pm => pm.CountAsync(1000)).ReturnsAsync(168);
            primesModelMock.Setup(pm => pm.CountAsync(1000000)).ReturnsAsync(78498);
            viewModel = new PrimesViewModel(primesModelMock.Object);

            //add a delegate instance to the ViewModel's PropertyChanged event to
            //enable tracking
            ((INotifyPropertyChanged)viewModel).PropertyChanged +=
                (object sender, PropertyChangedEventArgs e) =>
                                        { eventRaised = true; eventArgs = e; };
        }
```

The following test asserts that when the ViewModel's Result property is set, a PropertyChanged event is raised. The ViewModel implements INotifyPropertyChanged



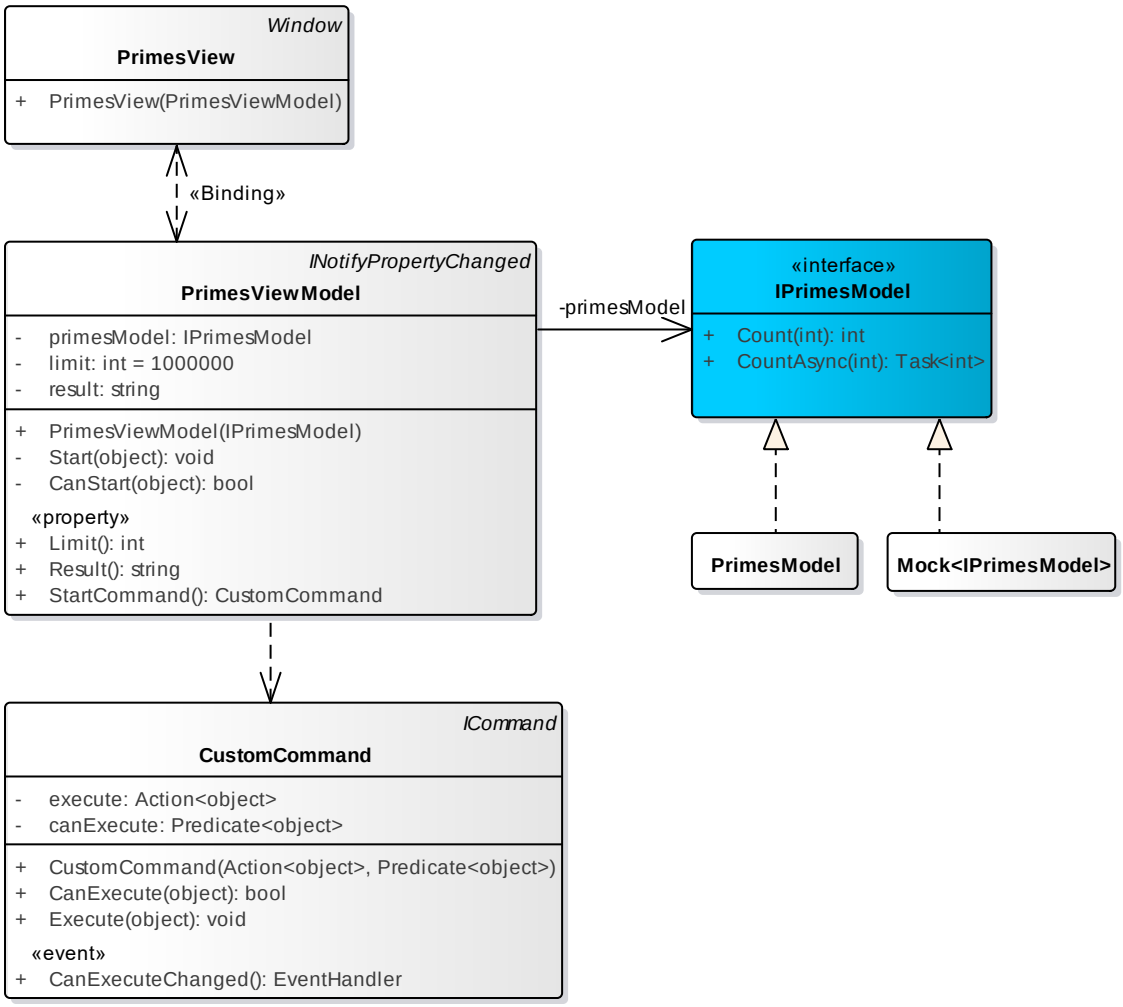

```
[Fact]
[Trait("Category", "Unit Test - Interactions")]
public void ResultProperty_ShouldRaiseEventNamedResult()
{
    // Act
    viewModel.Result = "168";

    // Assert
    Assert.True(eventRaised);
    Assert.Equal("Result", eventArgs.PropertyName);
}
```

Commands

The ViewModel can contain operations as well as state.

```
<Button Command="{Binding StartCommand}"
```

The Command property of the Button is bound to a property that implements ICommand. StartCommand is a property of type CustomCommand. This is a sample implementation of ICommand whose constructor takes two delegate arguments; an Action that determines what the command does and a predicate that determines whether the component is enabled. The Start and CanStart private methods match the signatures of these delegates.




```
namespace WinClient.MVVM.Utility
{
    public class CustomCommand : ICommand
    {
        private Action<object> execute;
        private Predicate<object> canExecute;

        public CustomCommand(Action<object> execute, Predicate<object> canExecute)
        {
            this.execute = execute;
```

```
            this.canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return canExecute == null ? true : canExecute(parameter);
        }

        //executes code in ViewModel
        public void Execute(object parameter)
        {
            execute(parameter);
        }

        public event EventHandler CanExecuteChanged
        {
            add
            {
                CommandManager.RequerySuggested += value;
            }
            remove
            {
                CommandManager.RequerySuggested -= value;
            }
        }
    }
}
```

```
namespace WinClient.MVVM.ViewModel
{
    public class PrimesViewModel : INotifyPropertyChanged
    {
        private IPrimesModel primesModel;

        public PrimesViewModel(IPrimesModel primesModel)
        {
            this.primesModel = primesModel;
            LoadCommands();
            LoadData();
        }

        public CustomCommand StartCommand { get; private set; }

        private void LoadCommands()
        {
            StartCommand = new CustomCommand(Start, CanStart);
        }

        private void Start(object obj)
        {
            LoadData();
        }

        private bool CanStart(object obj)
        {
            return true;
        }
```

Design Principles and patterns

10 Design Principles

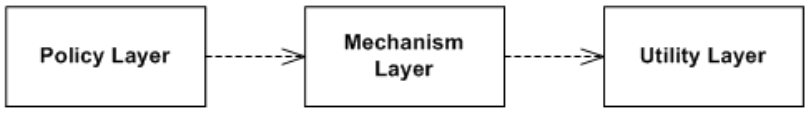
- Don't repeat yourself
  - but avoid using common code for unrelated functionality
- Encapsulate what changes
  - make variables and methods private by default and increase access step by step, from private to protected
- Program to an interface, not to an implementation
  - variables, method arguments and return types
- Favour composition over inheritance
  - Enables changing behaviour at runtime
- Principle of least knowledge
  - Only talk to your immediate friends (use only one dot)
  - advantage - software is more maintainable and adaptable since objects are less dependent on the internal structure of other objects
  - disadvantage - may result in having to write many wrapper methods to propagate calls to components
- S - Single responsibility principle
  - A class should have responsibility over a single part of the functionality
- O - Classes open for extension but closed for modification
  - new functionality should be added with minimum changes in the existing code
  - see Decorator pattern
- L - Liskov Substitution Principle
  - Base class can be replaced with a derived class without affecting functionality
  - an extension of the Open Close Principle: ensure that new derived classes don't change behaviour of base class

```
class Square : Rectangle
{
    public override int Width
    {
        set
        {
            width = value;
            height = value;
        }
    }
```

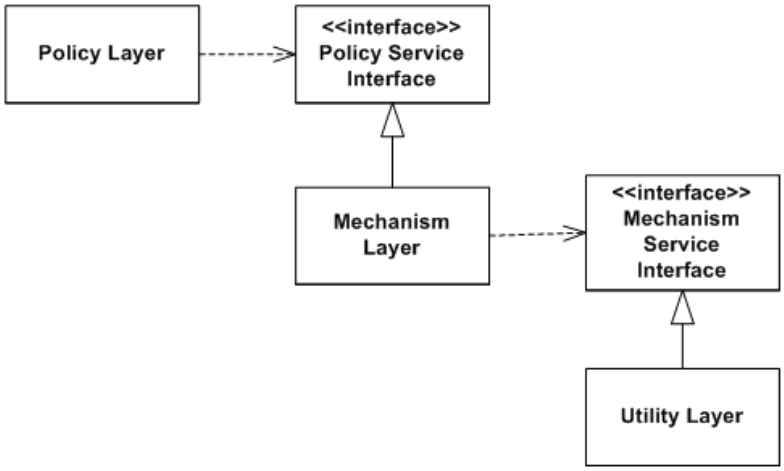
- I - Interface Segregation principle
  - client should not be forced to depend on methods it does not use
  - split one large interface into several smaller interfaces

- D - Dependency inversion
    - High-level modules should not depend on low-level modules. Both should depend on abstractions.
    - Abstractions should not depend on details. Details should depend on abstractions.
        - if the details change they should not affect the abstraction
    - inversion of control is a way of implementing the dependency inversion principle

      Without dependency inversion, higher-level components depend on lower-level components



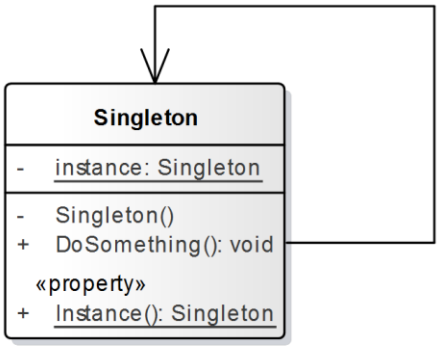
With DI, abstractions are owned by the higher-level components



Design patterns

Singleton

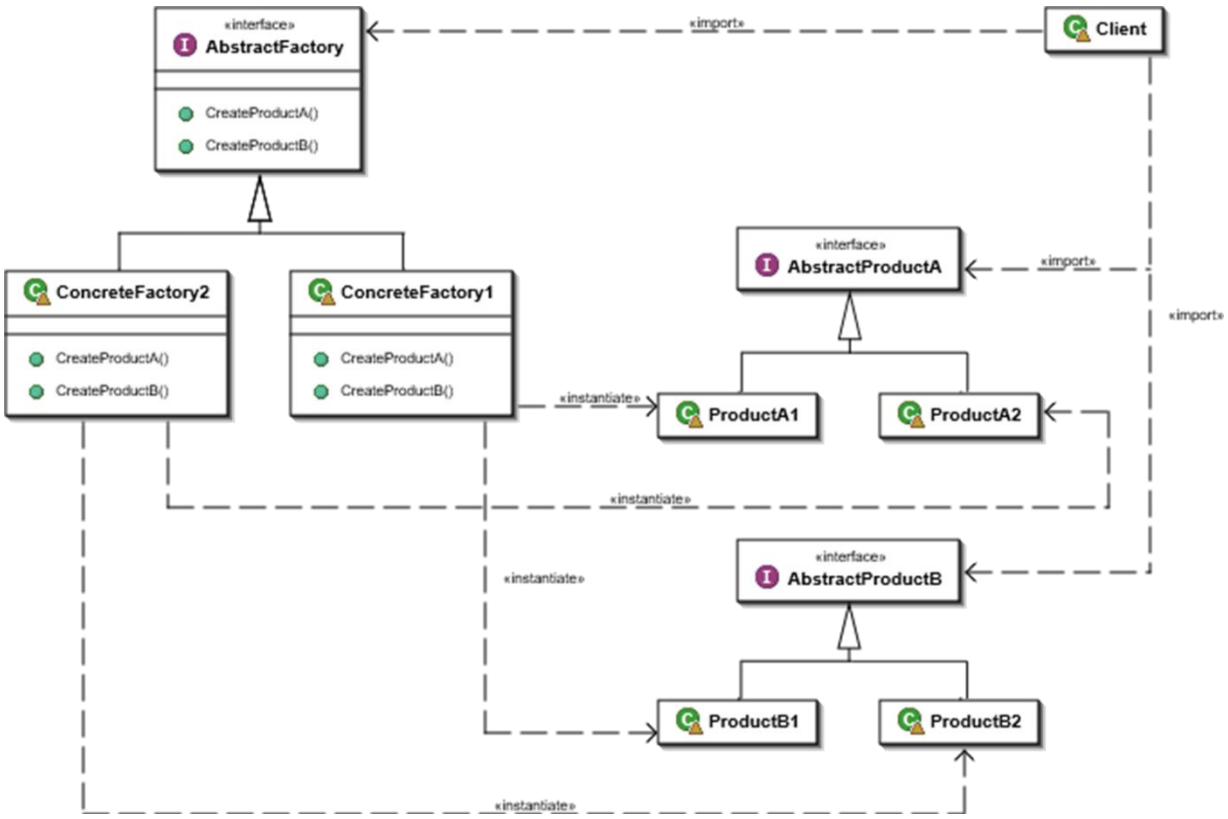
ensure that only one instance of a class is created

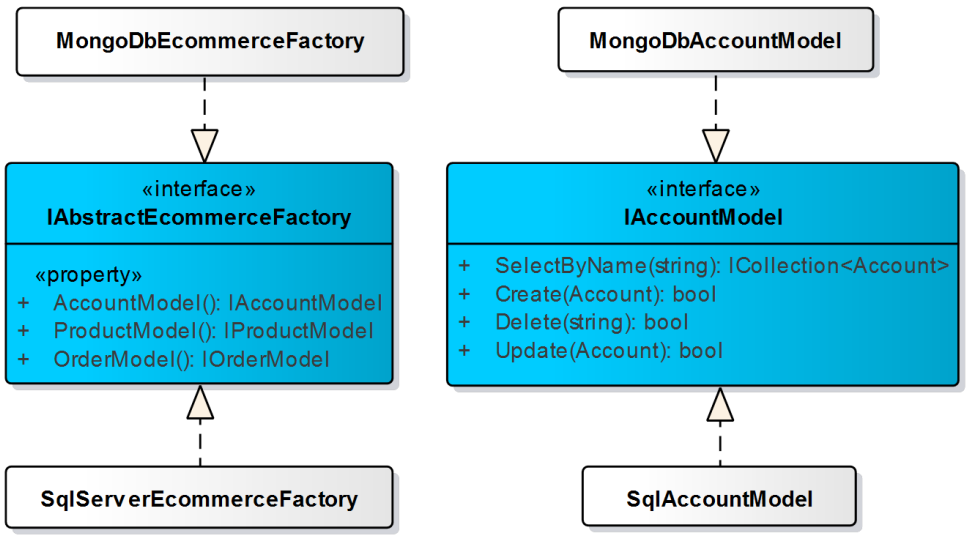



```
public class Singleton
{
    private static Singleton instance;
    private Singleton()
    {
    }
    public static Singleton Instance
    {
        get
        {
            if (instance == null)
                instance = new Singleton();
            return instance;
        }
    }
    public void DoSomething()
    {
    }
}
```

Abstract Factory

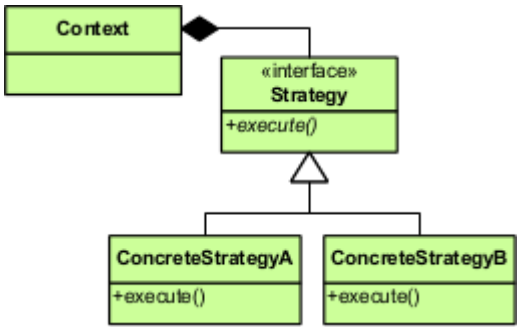
Provides an interface for creating families of related objects



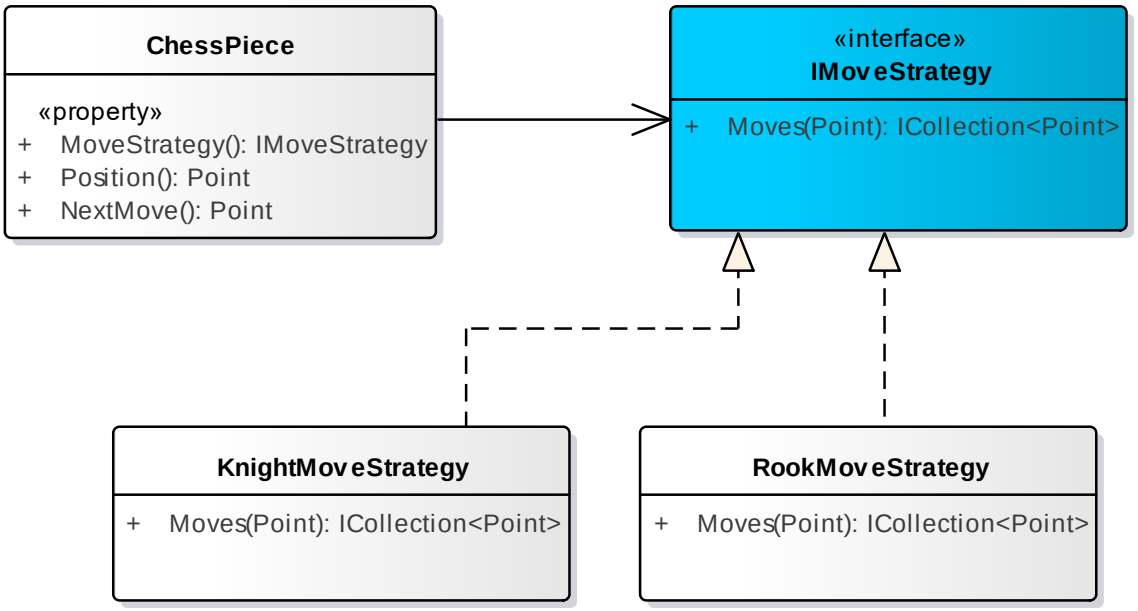



```
IAbstractEcommerceFactory factory = new SqlServerEcommerceFactory();

IAccountModel accountModel = factory.AccountModel;
accountModel.Create(new Account());
IProductModel productModel = factory.ProductModel;
productModel.Create(new Product());
IOrderModel orderModel = factory.OrderModel;
orderModel.Create(new Order());
```

Strategy

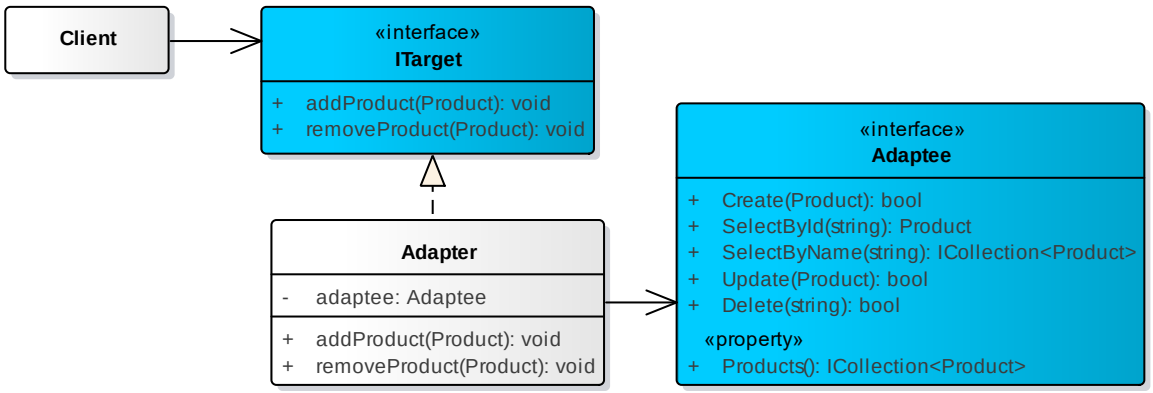


When classes differ only in their behaviour, isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.



Adapter

converts the interface of a class into another interface that clients expect.

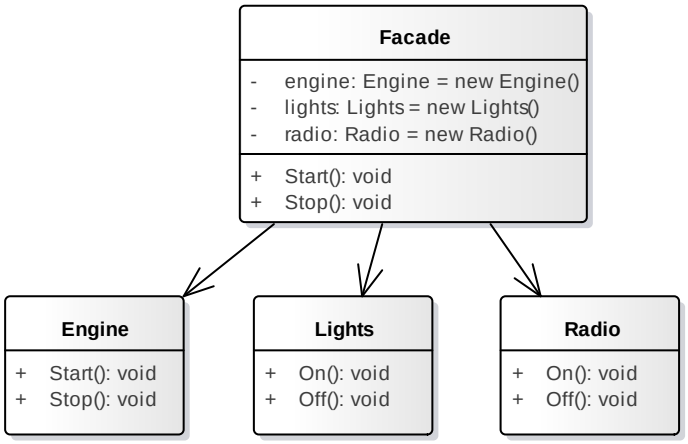


- Target - defines the domain-specific interface that Client uses.
- Adapter - adapts the interface Adaptee to the Target interface.
- Adaptee - defines an existing interface that needs adapting.
- Client - collaborates with objects conforming to the Target interface.

```
public class Adapter : ITarget
{
    private Adaptee adaptee;

    public void addProduct(Product p)
    {
        adaptee.Create(p);
    }

    public void removeProduct(Product p)
    {
        adaptee.Delete(p.Id);
    }
}
```

Façade

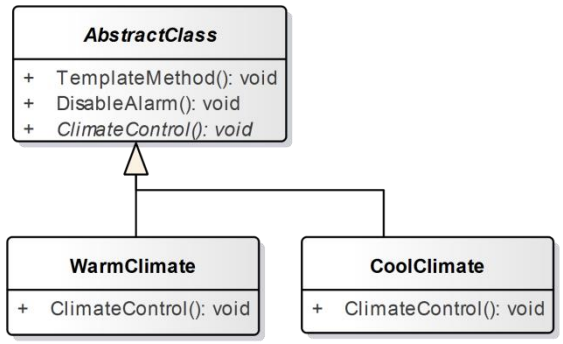
provides a unified interface to a set of interfaces in a subsystem.




```
public class Facade
{
    private Engine engine = new Engine();
    private Lights lights = new Lights();
    private Radio radio = new Radio();
    public void Start()
    {
        engine.Start();
        lights.On();
        radio.On();
    }
    public void Stop()
    {
        engine.Stop();
        lights.Off();
        radio.Off();
    }
}
```

Template Method

defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behaviour.



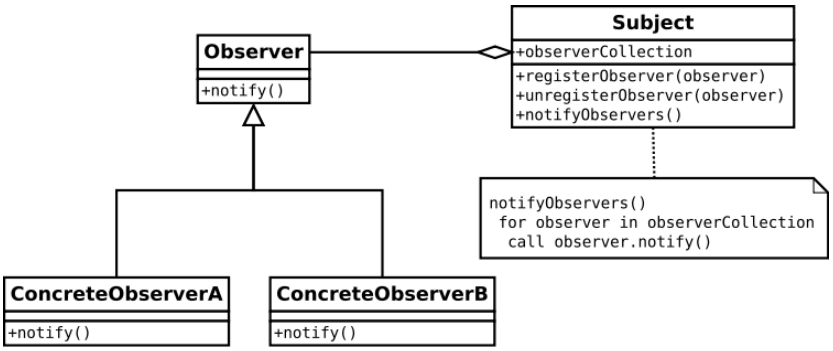

```
public abstract class AbstractClass
{
    public void TemplateMethod()
    {
        DisableAlarm();
        ClimateControl();
    }
    public void DisableAlarm()
    {
        Console.WriteLine("Alarm disabled");
    }
    public abstract void ClimateControl();
}
```

algorithm

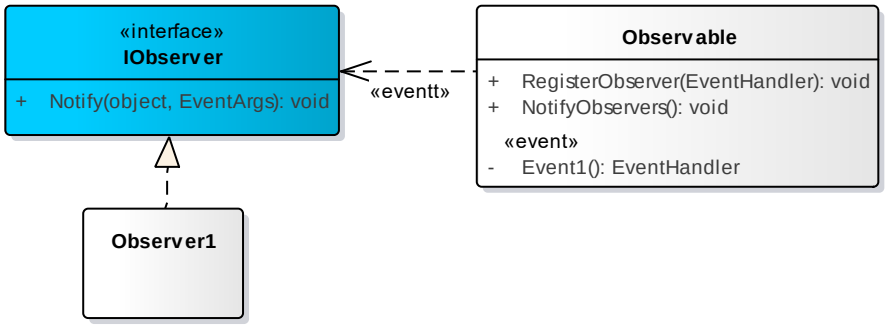
```
public class WarmClimate : AbstractClass
{
    public override void ClimateControl()
    {
        Console.WriteLine("Air conditioning");
    }
}
```

Observer

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



- Observable – defines the operations for attaching and de-attaching observers to the client. Notifies the attached Observers when the state changes
- IObserver - interface defining the operations to be used to notify this object.
- Observer1 - IObserver implementation

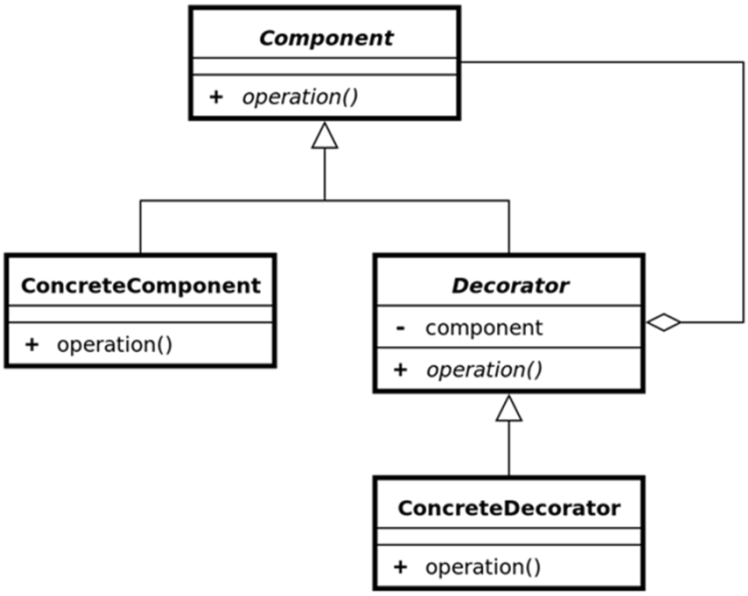


An implementation of the pattern using .NET events

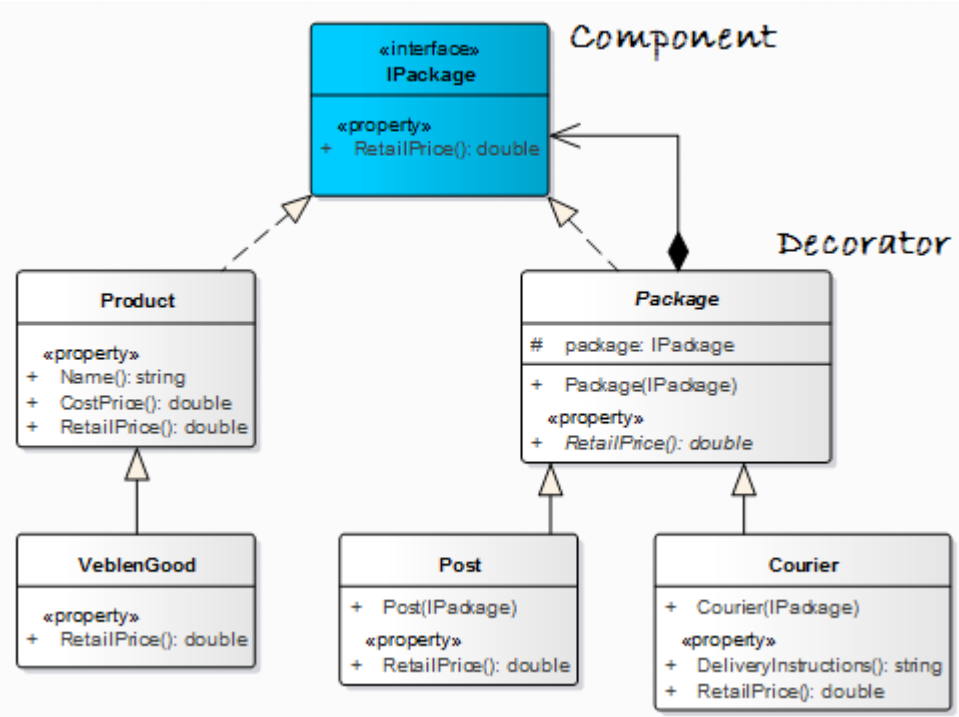
```
public class Observable
{
    private event EventHandler Event1;
    public void RegisterObserver(EventHandler handler)
    {
        Event1 += handler;
    }
    public void NotifyObservers()
    {
        Event1(this, new EventArgs());
    }
}
class Class1
{
    public static void Main(string[] args)
    {
        IObserver observer = new Observer1();
        Observable observable = new Observable();
        observable.RegisterObserver(observer.Notify);
        observable.NotifyObservers();
        Console.ReadKey();
    }
```

Decorator

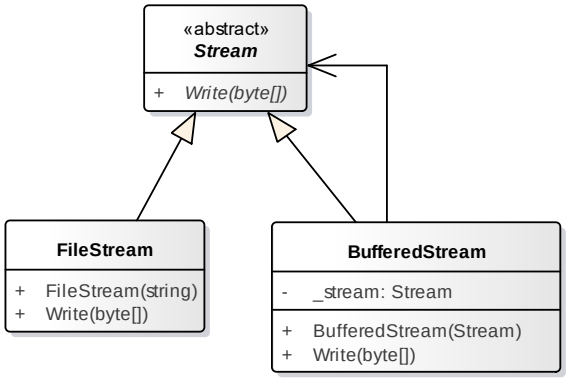
add additional responsibilities dynamically to an object



- Component - interface for objects that can have responsibilities added to them
- ConcreteComponent - object to which additional responsibilities can be added
- Decorator - Maintains a reference to a Component object and implements the Component's interface
- Concrete Decorators - extend functionality of the component by adding state or behaviour



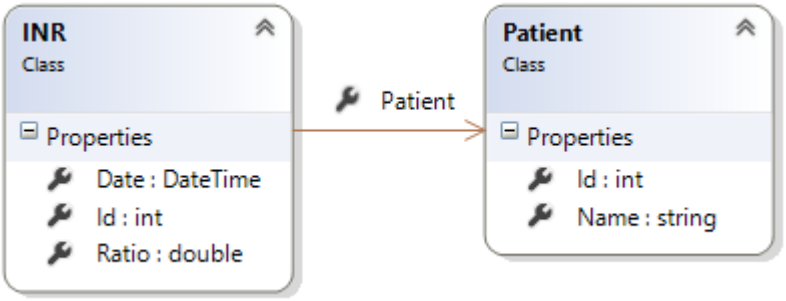
The System.IO namespace uses the Decorator design pattern




```
FileStream fileStream = new FileStream(@"C:\Users\User\Documents\file1.bin",
                                                                FileMode.Create);
BufferedStream bufferedStream = new BufferedStream(fileStream);
byte[] bytes = {10,20,127};
bufferedStream.Write(bytes, 0, 3);
bufferedStream.Close();
fileStream.Close();
```

Exercise

The INR is the ratio of a patient's blood clotting time to a normal sample. Build a web application that records INRs for Patients.




```
public class PatientController : Controller
{
    public ActionResult AddINR(int patientid)
    {
        //pass the selected patient id to the view
        //in a ViewBag property named patientid
        ViewBag.patientid = patientid;
        return View();
    }

    [HttpPost]
    public ActionResult AddINR(INR inr, int patientid)
    {
        inr.Patient = patientModel[patientid];
        inrModel.Create(inr);
        return RedirectToAction("List");
    }
```

```
@Html.Hidden("patientid", ViewBag.patientid as object)
```