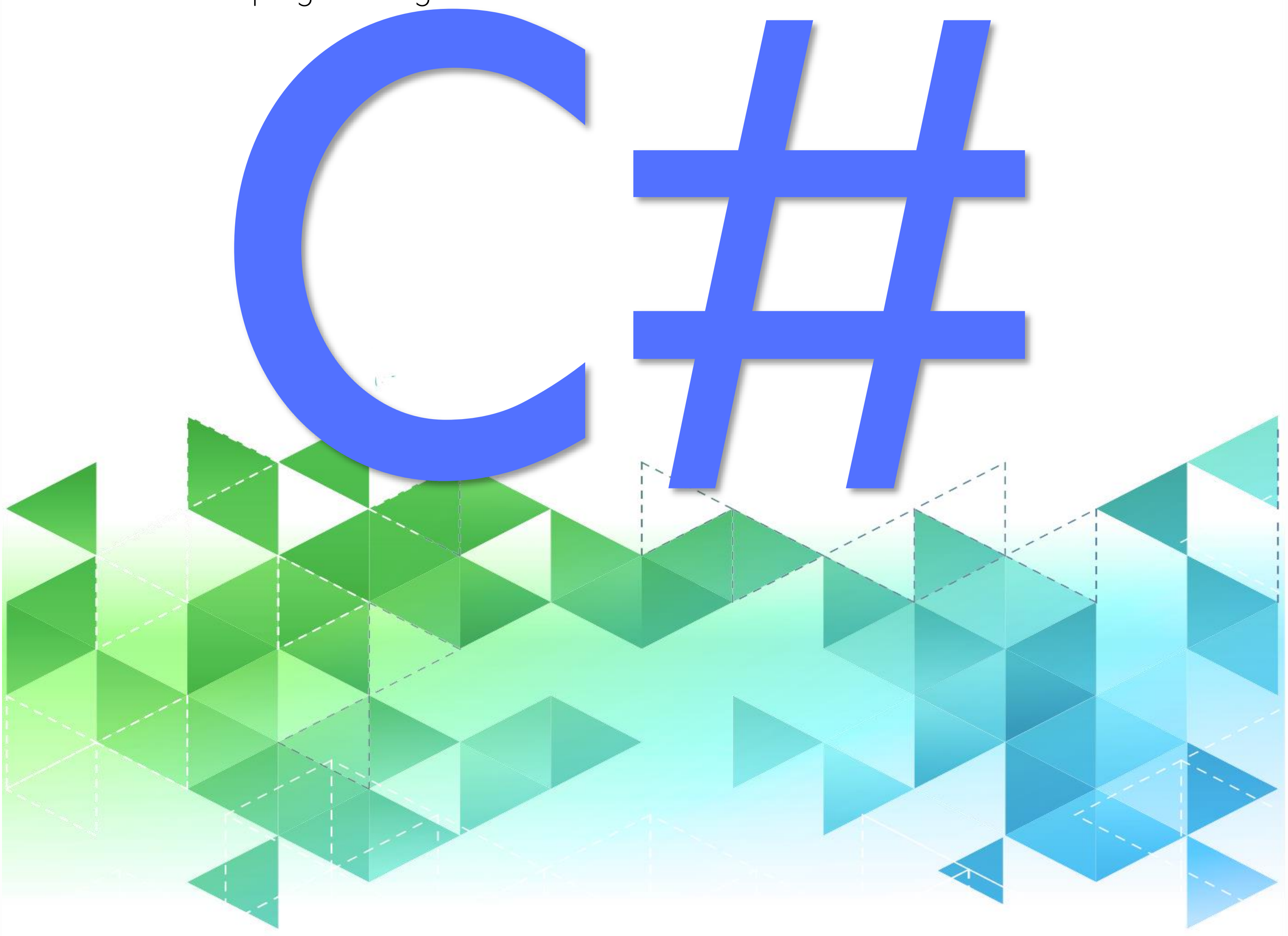
# Functional Programming In C#

{with: categories}

Gain advanced understanding of the mathematics behind modern functional programming.



dimitris papadimitriou v 2.2.1

# FUNCTIONAL PROGRAMMING IN C#  WITH CATEGORIES

Gain advanced understanding of the mathematics
behind modern functional programming

Dimitris Papadimitriou

**Version : 2.2.1** last updated 20-Apr-2021

Feel free to contact me at:

https://www.linkedin.com/in/dimitrispapadimitriou/

https://leanpub.com/u/dimitrispapadim

https://github.com/dimitris-papadimitriou-chr

dimitrispapadim@live.com

## Resources

**Source Code**: https://github.com/dimitris-papadimitriou-chr/FunctionalCSharpWithCategories

Also contains https://github.com/dimitris-papadimitriou-chr/Practical-Functional-CSharp

# Contents

# Purpose

The tools we use have a profound (and devious!) influence on
our thinking habits, and, therefore, on our thinking abilities.
— Edsger Dijkstra

OO makes code understandable by encapsulating moving parts.
FP makes code understandable by minimizing moving parts.
—Michael Feathers (Twitter)

One of the main reasons for this book is to transfer in the community of object-oriented developers some of the ideas and advancements happening to the functional side. This book wants to be pragmatic. Unfortunately, some great ideas are coming from academia that cannot be used by the average developer in his day to day coding.

**This is not an Introductory book to functional programming,** even if I restate some of the most basic concepts in the light of category theory and the object-oriented paradigm. This book covers the middle ground. Nonetheless, it is written in a way that if someone pays attention and goes through the examples, he or she could understand the concepts. An introductory book will follow in Leanpub, covering more basic ideas, along with the extended version of this book covering more advanced topics. If you think the content of this book is more difficult than what you expected, please contact me to give you a free copy of the book "The Simplified Functional Programming in C#, with categories" when it is finished in Leanpub.

**In this book, I will try to simplify the mathematical concepts in a way that can be displayed with code**. If something cannot be easily displayed with code probably will not be something that can be readily available to a developer's arsenal of techniques and patterns.

**If you think a section is boring, then skip it and maybe finish it later.**

One thing I admire in Software Engineers is their ability to infer things. The ability to connect the dots is what separates the exceptional developer from the good one. In some parts of the book, I might indeed have gaps or even assume things that you are not familiar with. Nonetheless, I am sure that even an inexperienced developer will connect the dots and assign meaning, as I usually do when left with unfinished

# About this book coding conventions

Let's be pragmatic here, this style of coding that promotes purity should **not be an end in itself**, and we should always be able to convince ourselves to go back into an object-oriented style of coding even if it is not that pure. The important thing is to write **functional code that provides business value** to the organization and to the end-user that will eventually use it. We must be pragmatic and utilitarian when we code and abstractionists when we think about code.

This book is aiming to present the basics of functional programming using the **language-ext** library. We will try to exhibit the usage of the basic Functional types: Option, Either, Task and Validation within an ASP.NET MVC web paradigm.

There is a **WebApplicationExample** solution that has multiple simpler examples of language-ext in a .net core web application. The intention is to build up an understanding behind the architecture of the **Sample Contoso** web Application in the **language-ext** GitHub project which is examined in the last chapter of this book.

## The language-ext C# library

This book is all about the **specialization** of the concepts of functional programming with the language-ext C# library. There are some functional libraries out there for C#, but currently the only complete library that could be used in a commercial project is the language-ext. In this section we are going to see a couple of the basic crosscutting ideas in the library and then in the following Chapters we will introduce all the different structures.

In this book we are going only to **use the Nu-get package LanguageExt.Core.**

# C# Functional Features

An overview

"1996 - James Gosling invents Java. Java is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Sun loudly heralds Java's novelty.

2001 - Anders Hejlsberg invents C#. C# is a relatively verbose, garbage collected, class based, statically typed, single dispatch, object oriented language with single implementation inheritance and multiple interface inheritance. Microsoft loudly heralds C#'s novelty."
-*Brief, Incomplete and Mostly Wrong History of Programming Languages*,

# 1 Language Functional support

## 1.1 Built-in .NET Delegates

.NET contains a set of delegates designed for commonly occurring use cases, so we do not have to create our own delegate for every single occasion. Those are residing in the System namespace. Here are the most important and most used :

---

`Func<T,R>`

The most common delegate by far will be the `Func<T, R>`, which represents functions and methods that look like this `R f(T t)` taking one argument and returning one result.

Visually this could be represented by just an arrow: $T \xrightarrow{f} R$

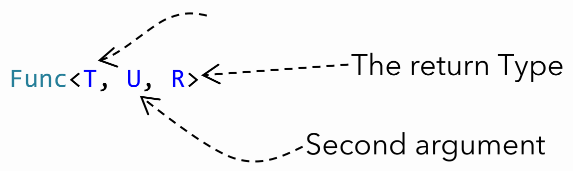
---

`Func<T,U,R>`

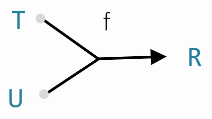
represents a function Type that has the following signature `R f(T t1, U t2)`.

The `Func<double, double, double>` represents a Delegate Type that has the following signature $(\mathbf{T}, \mathbf{U}) \rightarrow \mathbf{R}$

First argument



If you wanted to visualize this could be a join:



## Action<T>

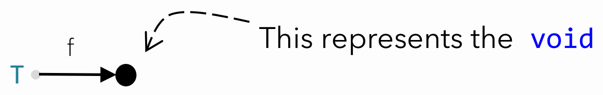
The `Action<T>` Functional interface represents functions that have this signature `void f(T t)`. In functional programming we usually do not like the `void` because it is not a Type like all the others, and this forces us to treat it differently. `Action<T>` is just `Function<T, void>` but because void is not a Type .NET had to invent `Action<T>`. As an example of usage look at the following, in which we "consume" a string returning nothing:

```
Action<string> print = (string x) => Console.WriteLine(x);
```

**When you return void is an indication that you create a side effect of some sort**. We can shorten the declaration, using the point free notation :

```
Action<string> print1 = Console.Write;
```

A visualization for `Action<T> could be:`

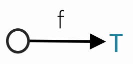


## Func<T>

The `Func<T>` represents functions that have this signature `T f()` that don't take any arguments. A simple example would be:

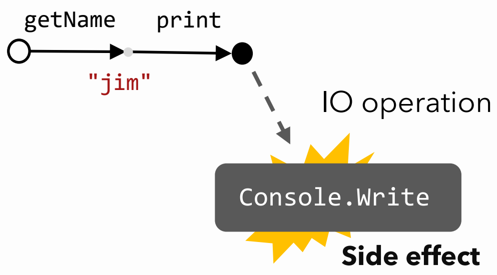
```
Func<string> getName =()=>"jim";
```

visualize this as a value coming out of nothing:



Func<T> is the dual of Action<T> and in this way we can compose them and annihilate them into a void.

```
Action<string> print = Console.Write;
Func<string> getName =()=>"jim";

print(getName());            // prints Jim as side effect with return type void
```



## Predicate<T>

The Predicate<T> Functional interface represents functions that have this signature `bool f(T t1)` that returns a bool. This is equivalent to a Func<T,bool>.

## Summary Table

| | |
|---|---|
| `Action<T1>` | `void f(T1 a){…}` |
| `Action<T1,T2>` | `void f(T1 arg1, T2 arg2){…}` |
| `Func<TResult>` | `TResult f(){…}` |
| `Func<T1,TResult>` | `TResult f(T1 arg1){…}` |
| `Func<T1,T2,TResult>` | `TResult f(T1 arg1, T2 arg2){…}` |
| `Func<T1,T2,…T16,TResult>` | `TResult f(T1 arg1, T2 arg2,…,T16 arg16){…}` |
| `Predicate<T>` | `bool f(T arg1){…}` |

**Remember** :

1. The Action delegates only have input arguments.
2. The Func<_,,,_> is the general case.
3. For Any Func<_,,,TResult> the Result is always at the Last position.
4. If you only want to return a result ()=>… you must use a Func<TResult>.

.NET Conventions

?.Invoke(…)

Since we can declare `Func` delegates as variables, it is possible that the delegate variable has not been assigned at the time of Invocation:

```
Func<string> getName = null;
var name = getName();        // we get a Null Reference Exception
```

in order to avoid this situation we usually use the ?.`Invoke()` method on the Delegates

```
getName?.Invoke();          //because of the ? operator we dont have an Exception.
```

## 1.2 Higher-order functions

A **Higher-Order function (Hof) is a function that receives a function as an argument or returns a function as output**. But I assume that you are already familiar with this definition.

The following is a Hof because it takes a `Func<T, U>` delegate as an argument.

Function as input

```
R F<T, U, R>(Func<T, U> g, T y)
{
        ...
}
```

The type of this function is **F:** ((**T**→**U**) ×**T**) →**R.** While the following is a Hof because it returns a `Func<T, U>`

Function as returned type.

```
Func<T, U> G<T, U>(T y)
{
        ...
}
```

With a type of **G: T**→(**T**→**U**) , we can also use the local function form inside other methods or functions.

In the form of `Func` delegates the previous functions now should be declared like this:

```
Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...

Func<T, Func<T, U>> G = (T y) => ...
```

unfortunately, C# does not allow for Generic definitions at the level of the delegate. We cannot write for example :

```
Func<Func<T, U>, T, R> F = <T, U, R> (Func<T, U> g, T y) => ...

Func<T, Func<T, U>> G = <T, U>(T y) => ...
```

For this reason, we must place them in a Function or Class, and include the generic constrains there:

We must place the generics on the scope of the function that contain the lambdas.

```
public static void Foo<T, U, R>()
{
        Func<Func<T, U>, T, R> F = (Func<T, U> g, T y) => ...

        Func<T, Func<T, U>> G = (T y) => ...
}
```

Again, here the type of `Func<Func<T, U>, T, R>` corresponds to ((**T**→**U**) × **T**) →**R** and the type of `Func<T, Func<T, U>>` corresponds to **T**→(**T**→**U**)

Obviously, we can have as many levels of Higher order Functions by allowing the Functions passed as arguments or return as results being themselves of Higher type. For example:

This is a Hof passed as an argument.

```
R F<T, U, R>(Func<Func<T, U>,U> g, T y)
{
        ...
}
```

With type **F:** (((**T**→**U**) →**U**) × **T**) →**R**. In practice this type of Higher-Higher-order-functions becomes too convoluted too fast, and it is not recommended. In a delegate variable declaration this already looks way to complex.

```
Func<Func<Func<T, U>, U>, T, R> F = (Func<Func<T, U>, U> g, T y) => ...
```

What makes it worse is that we cannot replace the variable type with the `var` keyword.

## 1.3 Extension Methods

we are going to use Extension methods a lot. Extension methods might diverge from the traditional Object-oriented model of programming but allow us to provide custom functionality to .NET native types or extend our classes in ways that are not possible (or very difficult) through the standard language capabilities.

For example, we can give useful methods to native types to simplify our code and provide method chaining. Take for example this extension of Tuples that we are going to use. By adding the Following static class that contains an Extension method on the `ValueTuple`

```
public static class FunctionalExt                          ③
{
 ②        public static List<T> ToList<T>(this ValueTuple<T, T> @this) =>
        ①        new List<T> { @this.Item1, @this.Item2 };
}
```

[Extension methods should be static (1) inside a static class (2) and you must use the `this` keyword (3) before the argument, which acts as a reference to the type you want to attach the method.]

We can now go ahead and use the `ToList<T>` method on any 2-item `ValueTuple`

```
var list = (4, 5).ToList();
```

since `ValueTuple<T,T>` is equivalent to a list `List<T>` with 2 items.

To take this one step further we will rewrite our Map definition on `ValueTuple<T,T>` from the Higher order Functions section :

```
ValueTuple<T1, T1> Map<T, T1>(ValueTuple<T, T> @this, Func<T, T1> f) =>
    new ValueTuple<T1, T1>(f(@this.Item1), f(@this.Item2));
```

and instead create an extension method on the `ValueTuple<T,T>`

```
public static class FunctionalExt
{
  public static ValueTuple<T1, T1> Map<T, T1>(
       this ValueTuple<T,T> @this,
       Func<T,T1> f) => new ValueTuple<T1, T1>(f(@this.Item1), f(@this.Item2));
}
```

The above declaration now will allow us to write:

```
var result  = (1,3).Map(x=>x*x);
```

instead of this:

```
var result = Map((1, 3), x => x * x);
```

in this way we obtain a certain control over the Native types and force our functional style of writing.

# 2 Algebraic Data Types

“Lists come up often when discussing monoids,
and this is no accident: lists are the “most fundamental” Monoid instance.”
Monoids: Theme and Variations (Functional Pearl)

## 2.1 Recompose the List

This is the end of this chapter and what for the rest of the section we are going to see various Recursive definition on the List. All the examples have two cases:

1. The base case for the empty List **[]**
2. And a recursive case where we break up the List **[v , Rest]**, use the first element (head) **v** of the list, and then Repeat the recursive operation for the rest of the list **Rest**.

Let us first start with something simple. We will break down the list and reassemble it. We define symbolically a function **Rebuild** that just rebuilds the list without modifying it:

**Rebuild** ([ ]) = [ ]

**Rebuild** ([v , rest]) = [v, **Rebuild** (rest) ]

An implementation would be:

**Rebuild** ([ ]) = [ ]

```
List<int> Rebuild(List<int> @list)
{
     return @list.MatchWith((
                    Empty: () => new List<int> { },
                    Cons: (v, rest) => (v, Rebuild(rest)).AsList()
                   ));
}
```

[v, **Rebuild** (rest) ]

Here the `.AsList()` is a custom extension on types:

```
public static List<T> AsList<T>(this ValueTuple<T, List<T>> @this) =>
    @this.Item1.AsList().Concat(@this.Item2);
```

Let us make an example “execution” of this simple Term Rewriting system with its two rules. Let’s start with an initial state [1, 4,6] on which we will apply the reduction sequentially.

**Rebuild** ( [1,4,6] ) →

[1, **Rebuild** ( [4,6] ) ] →

```
[1,4, Rebuild ( [6] )]          →
…                                       //at some point we empty the list
[1,4,6, Rebuild ( [] )]         →       //we reached the base case and the recursion stops
[1,4,6]
```

As you can see what we did is just traverse the list one element at a time by recursion and used the list monoid (**[],***) to rebuild it.

Using the `Aggregate` Linq method on the `List<T>`

```
List<T> Rebuild<T>(List<T> @list) => list.Aggregate(
                    new List<T> { },
                    (acc, current) => (acc, current).AsList()
                );
```

Unfortunately, this reverses the List ( this will happen even if you reverse the order of concatenation `(current, acc).AsList()`). The `Aggregate` is weaker than the recursion since we lose the order of traversal. `Aggregate` is great for the cases where we want to fold to things into less information e.g. Fold a list into a bool or an integer.

## 2.2 Filtering

Here we will add another `Func<T, bool> @predicate` parameter to the recursive Function, we are gong to use this `@predicate` to filter the Array using recursion. Symbolically, the **Filter** function:

```
Filter ( [ ], predicate) = [ ]

Filter ( [v ,  rest], predicate) =  if( predicate (v) )
                                        [v ,  Filter (rest , predicate) ]
                                    else
                                        Filter (rest , predicate)
```

This says:

1. If you have an Empty array just Return an Empty array.
2. If you have any other array:
    a. Check if the head of the array satisfy the predicate, if it does then then keep the head and append the Remaining Filtered array:
    [v , **Filter** (rest , predicate) ]

    b. If it does not then just return the Remaining Filtered array and discard the element :
    **Filter** (rest , predicate)

```
List<T> Filter<T>(List<T> @array, Func<T, bool> @predicate)
    where T : IEquatable<T> =>
            @array.MatchWith((
                                    Empty: () => new List<T> { },
                                    Cons: (v, rest) =>  @predicate(v) ?
                                          Filter(rest, @predicate) :
                                          (v, Filter(rest, @predicate)).AsList()
                                    ));
```

**Filter** ( [ ], predicate) = [ ]

**Filter** (rest , predicate)

[v ,  **Filter** (rest , predicate) ]

Run This:  .Net Fiddle

# Functors

“It should be observed first that the whole concept of a category is essentially an auxiliary one; Our basic concepts are essentially those of a functor and of a natural transformation.”
S. Eilenberg and S. MacLane

## 2.3 Maybe Functor aka Option<T>

### 2.3.1 Dealing with null

"…The absence of Sum types is what Tony Hoare called the **Billion Dollar Mistake** … all the superstitious nonsense about null is all about the failure of language design, because they don’t have Sum types. "

– Robert Harper
Oregon Programming Languages Summer School

The problem of null or undefined is one of the biggest problems in computation theory. If we want to write meaningful programs, we must accept the fact that some computations might yield no result. The usual way object-oriented languages deal with this is by checking the types for not being undefined in order to use them. Any developer knows the number of bugs that have as source the problem of undefined.

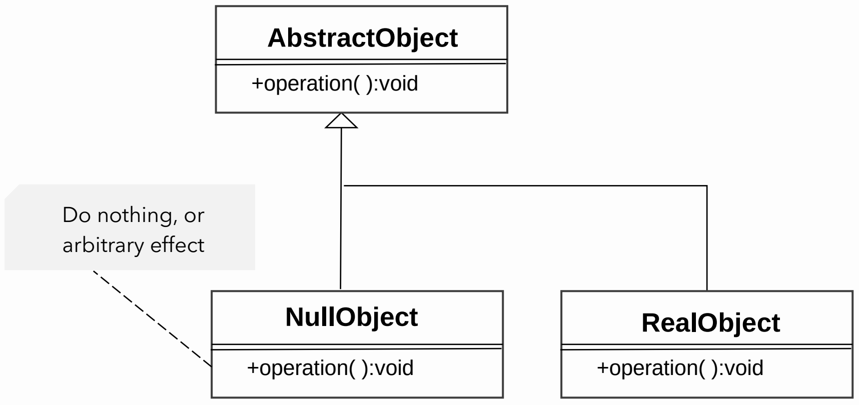
The classical solution is using conditionals everywhere to check for null.

```
if (result) {

   result.operation();

} else {

   //do nothing

}
```

This reduces the cohesion of the codebase, because we must tightly couple to code relating to the cross-cutting concern of null checking.

## 2.3.2 The Null Object Design pattern

A more elegant solution to the problem of Null is the "Null Object" design pattern. The null object pattern is a special case of the strategy design pattern. The idea here is to replace conditional with strategy]. Instead of setting something as null we can set it as `NullObject`. Were the `NullObject` methods are empty, or they do not do anything when called. `NullObject` is in fact designed in a way that if it is fed to any method that waits for a `RealObject`, there should not be any unexpected side-effect, or any exceptions thrown.



A simple implementation of the `NullObject` design pattern with C# could be something like this :

```
public abstract class AbstractObject<T>
{
        public abstract void Operation();
}

public class NullObject<T> : AbstractObject<T>
{
        public NullObject() { }
        public override void Operation() {//Do nothing here }
}

public class RealObject<T> : AbstractObject<T>
{
        private readonly T value;
        public RealObject(T value) => this.value = value;//instructor injection

        public override void Operation()
         {   // do something with value
             Console.WriteLine($"{value}");
         }
}
```

Run This: .Net Fiddle

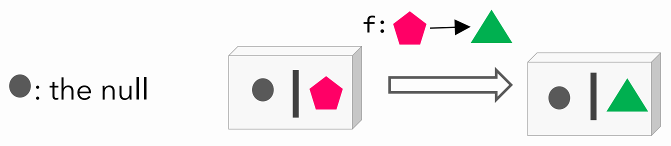
This implementation is unobtrusive. It should not affect the rest of the code and should remove the need to check for null.

The major problem with this pattern is that for each object, we need to create a nullObject with the operations that we are going to use inside the rest of the code. **This will force us to duplicate all the domain objects in our model. It is a big trade-off**. There is no easy way to abstract the mechanics of this implementation in an object-oriented world, mainly because we cannot have specific information about the operations that we want to isolate and create a nullObject that provides those operations.

### 2.3.3 The Functional equivalent - Maybe as Functor

In Functional programming, we use the **Maybe Type** [sometimes also called Option] in order to represent the possibility of something being null. The Maybe solves the null problem by moving the mechanics of null checking inside a functor `map`. **Thus, instead of applying the objects on functions, we reverse the flow and apply the functions onto objects**. Now, we can isolate the effect inside a single point; the `map`.

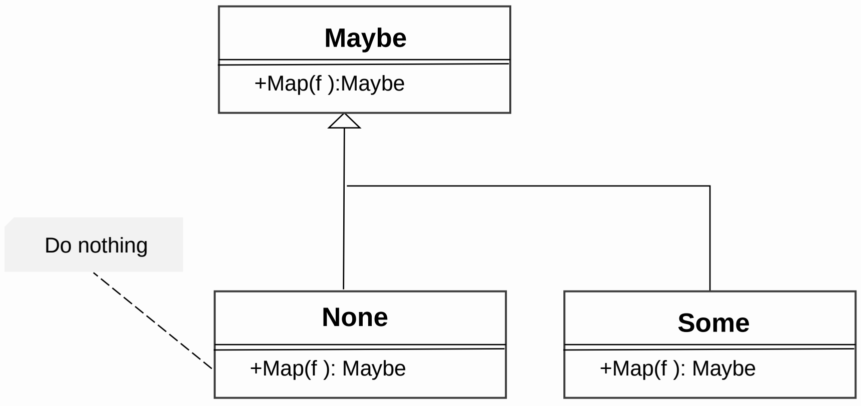
The standard implementation of the maybe/option Type is a hierarchy with a base class and two sub-types: the `Some` and the `None`. **The None represents the case of null**



After all this discussion, hopefully, the implementation of maybe functor would be apparent

```
public abstract class Maybe<T>
{
     public abstract Maybe<T1> Map<T1>(Func<T, T1> f);
     public abstract TResult MatchWith<TResult>(
                         (Func<TResult> None, Func<T, TResult> Some) pattern);
}

public class None<T>: Maybe<T>
{
     public override TResult MatchWith<TResult>(
          (Func<TResult> None, Func<T, TResult> Some) pattern) => pattern.None();

     public override Maybe<T1> Map<T1>(Func<T, T1> f) => new None<T1>();
}

public class Some<T> : Maybe<T>
{
    private readonly T value;
    public Some(T value) => this.value = value;

    public override TResult MatchWith<TResult>(
(Func<TResult> None, Func<T, TResult> Some) pattern) => pattern.Some(value);

    public override Maybe<T1> Map<T1>(Func<T, T1> f) => new Some<T1>(f(value));
}
```

Run This: .Net Fiddle



Maybe MatchWith implementation for Maybe is straightforward. Since it is a hierarchy, we have to implement for each subtype the MatchWith, and we are going to call different callbacks, so we can distinguish (pattern match) between Some and None.

And we can use it like this:

```
 var result = new None<int>( )
                .Map(v=> $"number is  : {v}")
                .MatchWith<string>(pattern: (
                    None: () => "Not found",
                    Some: v => $"Answer - {v}"
            ));
```

Run This: .Net Fiddle

### 2.3.3.1 Using C# 8. pattern matching

We will display both MatchWith and switch pattern matching just for completeness. We can skip the custom definition by using the native C# 8.0 pattern matching by defining

```
   public abstract class Maybe<T>
   {
        public Maybe<T1> Map<T1>(Func<T, T1> f) =>
         this switch
         {
             None<T> { } => new None<T1>(),
             Some<T> { Value: var v } => new Some<T1>(f(v)),
         };
   }

    public class None<T> : Maybe<T>
    {
        public None() { }
    }

    public class Some<T> : Maybe<T>
    {
        public readonly T Value;
        public Some(T value) => this.Value = value;
    }
```

If we want to hide the value `public readonly T Value` and make it into `private` then we can use the `Deconstruct` to gain access for the pattern match :

```
public abstract class Maybe<T>
{
        public Maybe<T1> Map<T1>(Func<T, T1> f) =>
         this switch
         {
             None<T>() => new None<T1>(),
             Some<T>(var v) => new Some<T1>(f(v)),
             _ => throw new NotImplementedException(),
         };
}


public class None<T> : Maybe<T>
{
     public None() { }
     public void Deconstruct() { }
}

public class Some<T> : Maybe<T>
{
     private readonly T value;
     public Some(T value) => this.value = value;
     public void Deconstruct(out T value) => value = this.value;
}
```

Run This: Fiddle

With this formalism we can now write as an example :

```
 string result = new None<int>()
              .Map(x => x + 1)
              switch
              {
                None<int>() => "nothing",
                Some<int>(var x) => x.ToString(),
                _ => throw new NotImplementedException(),
            };
```

Run This: Fiddle

### 2.3.4 Maybe Functor Example

Now let us get a client asynchronously with a certain Id from a repository.

```
public class MockClientRepository
{
    List<Client> clients = new List<Client>{
                    new  Client{Id=1, Name="Jim"},
                    new  Client{Id=2, Name="John"}
                };

        public Maybe<Client> GetById(int id) =>
                    clients.FirstOrNone(x => x.Id == id);

}
```

Run This: .Net Fiddle

Here we have replaced the result of the array filtering:

```
.FirstOrDefault(x => x.Id == id)
```

with something that will return a `Maybe<Client>`. If there is no client for a specific id, we should return `None()`. If there is a client, we will return `Some(client)`.

```
public static partial class FunctionalExtensions
{

public static Maybe<T> FirstOrNone<T>(this List<T> @source,Func<T,bool> predicate)
{
            var firstOrDefault = @source.FirstOrDefault(predicate);
            if (firstOrDefault != null)
                return new Some<T>(firstOrDefault);
            else
                return new None<T>();
 }
}
```

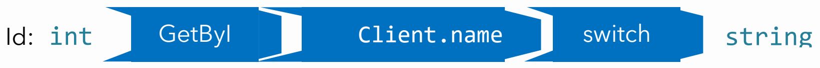
Now we can write:

```
var repository = new MockClientRepository();

var result = repository.
                GetById(1)
                  .MatchWith(pattern:(
                        none: () => "Not Found",
                        some: (client) => client.Name
                    ));
```

Or by using native pattern matching

```
var repository = new MockClientRepository();
var result = repository.
                GetById(1) switch
                {
                    None<Client>() => "Not Found",
                    Some<Client>(var client) => client.Name,
                    _ => throw new NotImplementedException(),
                };
```

Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle



**SideNote**:

You can run a simple ASP.NET MVC Maybe Functor Example .NET Fiddle. Also you can download the Practical-Functional-CSharp project from Github and run WebApplicationExample.sln the from the **WebApplicationExample** folder**.**

## 2.3.5 Maybe in Language-ext / Option

The usual name for this functor F=1+X (disjoint union with a base point) in most functional languages is called **Maybe** but in Language-ext is called **Option** which is another common naming for Maybe that comes closer to the OOP paradigm. Lets say that **Maybe** is the name of the concept and **Option** the implementation. It is the same thing. For the rest of this book **we will use language-ext Option** monad in our code examples.

### 2.3.6 Maybe Functor Example With language-ext Option

Fortunately for us the language-ext have defined **an implicit operator on the Option that does the conversion of null to Option immediately**:

```
public static implicit operator Option<A>(A a);
```

this allows us to write something like the following:

```
public Option<Client> GetById(int id) => clients.SingleOrDefault(x => x.Id == id);
```

We now return an `Option<Client>`

the compiler expects an `Option<Client>` so it uses the operator `Option<A>(A a)` in order to do the conversion on the spot. Now we can write:

```
public class MockClientRepository
 {
        List<Client> clients = new List<Client>{
                    new  Client{Id=1, Name="Jim", },
                    new  Client{Id=2, Name="John",}
                };

      public Option<Client> GetById(int id) =>
                          clients.SingleOrDefault(x => x.Id == id);
    }
```

Run This: .Net Fiddle

We can write again:

```
public string GetNameById(int clientId) =>
       clients
        .GetById(clientId)
        .Map(client => client.Name)
        .Match(Some: (name) => name,
               None: () => "no client");
```

Run This: .Net Fiddle

### 2.3.7 C# 8. pattern matching Support for Option

Because of the structure of the Option implementation in language-ext library, it's not possible to use `switch` directly on the `Option<>` but fortunately it exposes an `Option<>.Case` property that allow us to use `switch:`

```
public string GetNameById1(int clientId) =>
        clients
        .GetById(clientId)
        .Map(client => client.Name)
         .Case switch
        {
            SomeCase<string>(var name) => name,
            NoneCase<string> { } => "no client",
            _ => throw new NotImplementedException()
        };
```

The Option exposes a `Case` property.

Run This: .Net Fiddle

The two subtypes that we must match on, are the `SomeCase<>` and `NoneCase<>` and we are going to use this form instead of the `Match` a lot. Both provide the same behaviour, and it comes down to preference.

The only drawback is that **you must write explicitly the Types** of the `SomeCase<>` and `NoneCase<>` because the compiler cannot infer those.

### 2.3.8 Folding Maybe

For the maybe Fold is extremely easy to implement we start with the accumulator and if we have a None then we just return itself or in the case of the Some we apply the reducer to the `accumulator` and the value in order to merge the values:

```
public S Fold<S>(S accumulator, Func<S,T,S> reducer) =>
          this.MatchWith((
                   none: () => accumulator,
                   some: (v) => reducer(accumulator,v)
              ));
```

Most of the times (almost all) we will prefer the MatchWith to get the value out of the Maybe. We can use Fold like this:

```
 public string GetNameById(int clientId) =>
        clients
        .GetById(clientId)
        .Map(client => client.Name)
         .Fold("", (a, name) =>
         {
             return name;
         });
```

Run This: .Net Fiddle

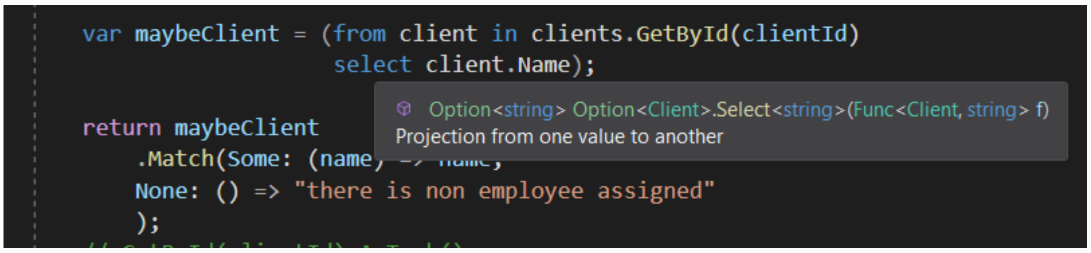
### 2.3.9 Using the Linq syntax

The language-ext library provides a Select method that allows us to use the Linq syntax with the Option:

```
public string GetNameById (int clientId) =>
        (from client in clients.GetById(clientId)
         select client.Name)  .Case switch
        {
            SomeCase<string>(var name) => name,
            NoneCase<string> { } => "no client",
            _ => throw new NotImplementedException()
        };
```

Run This: .Net Fiddle
Run This: ASP.NET MVC Fiddle

you can see the signature of the select if you place the cursor over the select Keyword.



```
var maybeClient = (from client in clients.GetById(clientId)
                   select client.Name);

return maybeClient
    .Match(Some: (name) => name,
    None: () => "there is non employee assigned"
    );
```

Option<string> Option<Client>.Select<string>(Func<Client, string> f)
Projection from one value to another

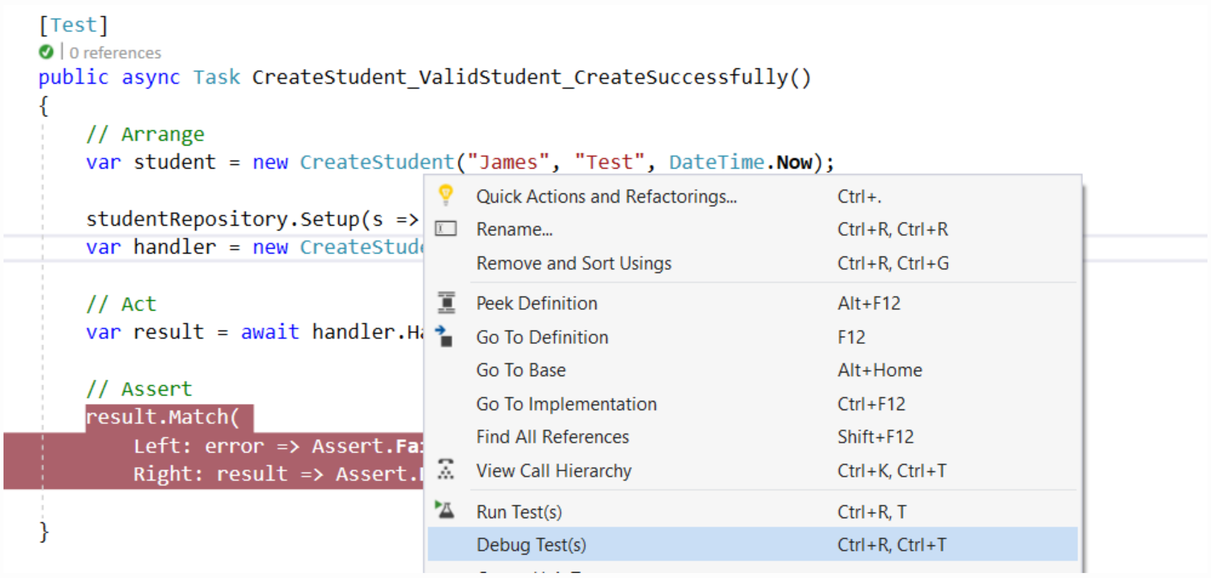
# Clean Architecture in .NET

In this Section we are going to take a brief look at the possibility of **Functional Software Architecture** and how this can be integrated in the latest trends in **modern Object-oriented software architectures**. This is not a deep dive in Architectural Design but a discussion behind the architecture of the **sample Contoso** web Application in the **language-ext** GitHub project.

## 3 A Clean Functional Architecture Example

### 3.1 Download and Setup the Project

The project is ***headless*** and does not have any **Views** implemented. You can mock Http actions using postman or any other tool, or you might consider building your own front end UI. You can also Debug over the Tests accompanying the App as an Entry point.

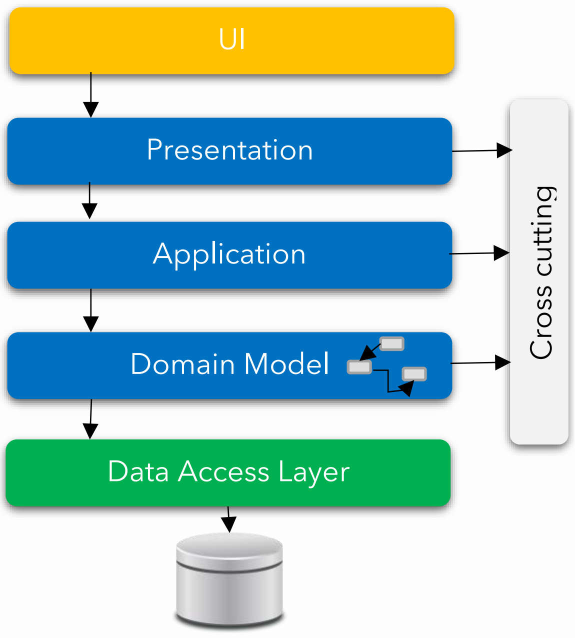


```
[Test]
0 references
public async Task CreateStudent_ValidStudent_CreateSuccessfully()
{
    // Arrange
    var student = new CreateStudent("James", "Test", DateTime.Now);

    studentRepository.Setup(s =>
    var handler = new CreateStud

    // Act
    var result = await handler.H

    // Assert
    result.Match(
        Left: error => Assert.Fa
        Right: result => Assert.
}
```

Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Shift+F12
View Call Hierarchy Ctrl+K, Ctrl+T
Run Test(s) Ctrl+R, T
Debug Test(s) Ctrl+R, Ctrl+T

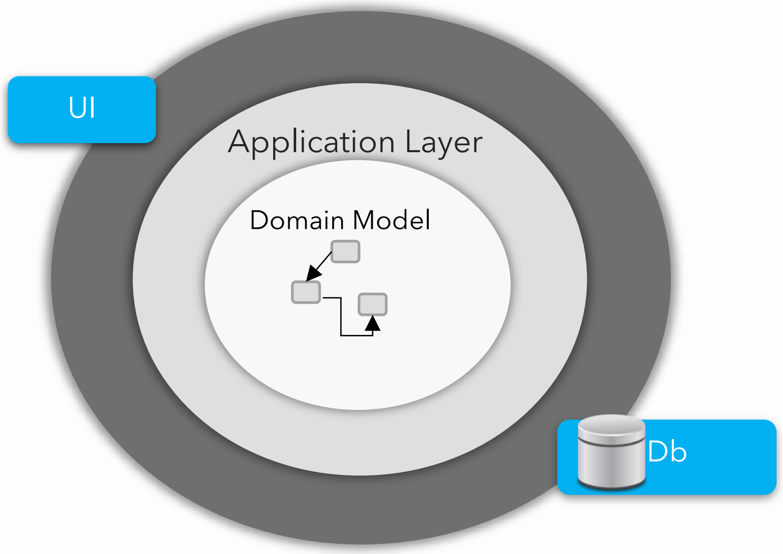
### 3.1.1 Clean Architecture with .NET core

The architecture of web applications depends on the scale and scope of the application. The Previous decade the dominant architecture for small and medium web applications was the **Layered Architecture.** A Layered Architecture, organizationed the project structure into four main categories: **presentation, application, domain, and infrastructure**.



Under the weight of ideas such as Domain -Driven Design from Eric Evans and Clean Code from "Uncle Bob" - Robert C. Martin the focus of the Architecture became the separation of concerns of the **Domain** from all the technicalities such as technologies, tools and implementation details.

**The domain and its Use Cases** now were placed in the **centre** of the architecture design with minimal dependencies. This led to a series of evolutionary transformations of the Layered structure ( Hexagonal Architecture, Pipes and adapters , DDD  etc) leading at to the concept of Clean architecture.



With Clean Architecture, the **Domain** and **Application** layers are at the centre of the design. This is known as the **Core** of the system.

The **Domain** layer contains enterprise logic and types and the **Application** layer contains business logic and types. The difference is that enterprise logic could be shared across many systems, whereas the business logic will typically only be used within this system.

**Core** should not be dependent on data access and other infrastructure concerns, so those dependencies are inverted. This is achieved by adding interfaces or abstractions within **Core** that are implemented by layers outside of **Core**. For example, if you wanted to implement the Repository pattern you would do so by adding an interface within **Core** and adding the implementation within **Infrastructure**.

All dependencies flow inwards, and **Core** has no dependency on any other layer. **Infrastructure** and **Presentation** depend on **Core**, but not on one another.

**Resources**:

https://github.com/maldworth/OnionWebApiStarterKit
https://github.com/josecuellar/Onion-DDD-Tooling-DotNET
Clean Architecture with ASP.NET Core 2.1 | Jason Taylor
You can also find Visual studio Templates to initialize Projects for example
https://github.com/jasontaylordev/CleanArchitecture

## 3.2 A Functional Applications Architecture

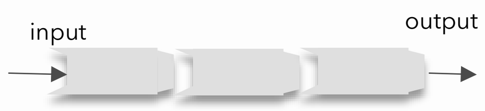
Where does the Functional Programming fit in the larger Architectural view? In the article form Scott Wlaschin "A primer on functional architecture" he recognizes three principles of functional programming that can be applied to the architectural level:

1. The first is that **functions** are standalone values. In a functional architecture, the basic unit is also a function, but a much larger business-oriented one that I like to call a **workflow**

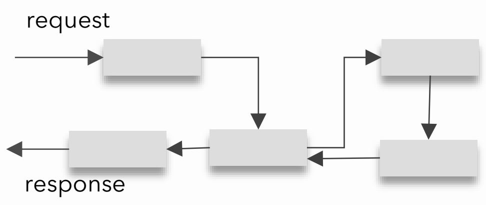
2. Second, **composition** is the primary way to build systems. Two simple functions can be composed just by connecting the output of one to the input of another

3. functional programmers try to use **pure functions** as much as possible

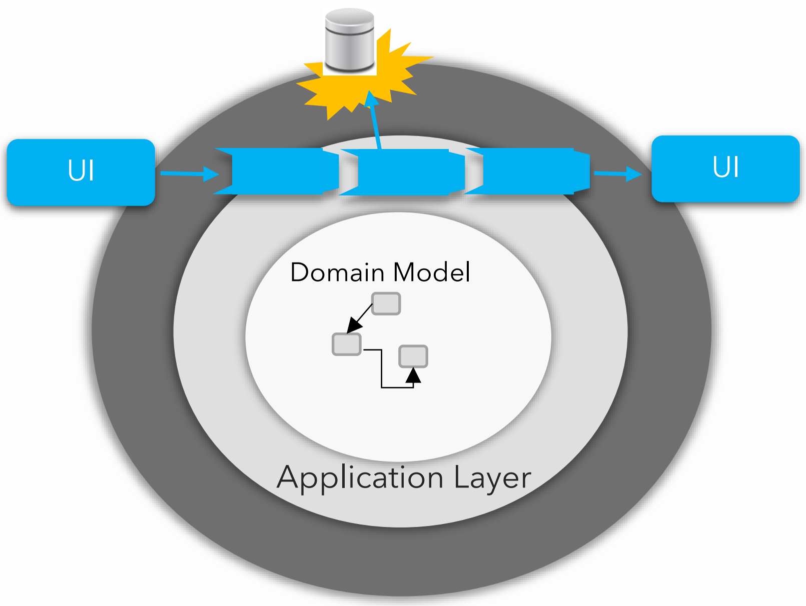
The general idea is that you write as much of your application as possible in an FP style chain computation using the Task, Option, Either, Validation Monads or their combinations and avoiding side effects and coupling. This would lead to the usual pipeline computation style:



Instead of the chaotic object oriented:



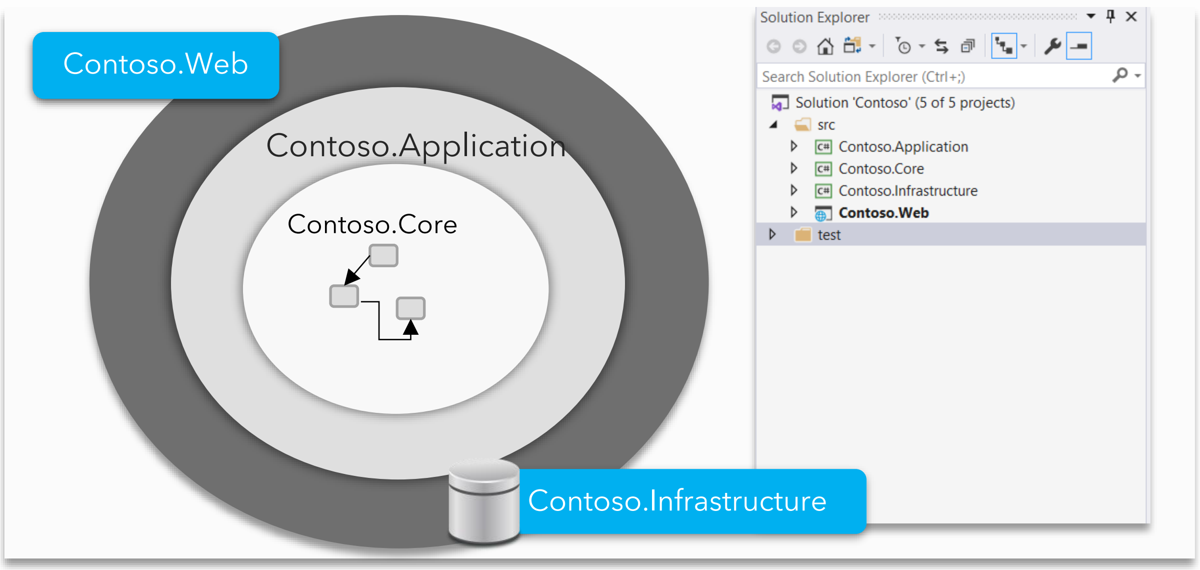
The idea for a functional architecture as laid out by Wlaschin is a pipeline that goes through the layers of the Architecture with minimal coupling and side effects.



## 3.3 The Contoso Clean Architecture with .NET core and language-ext

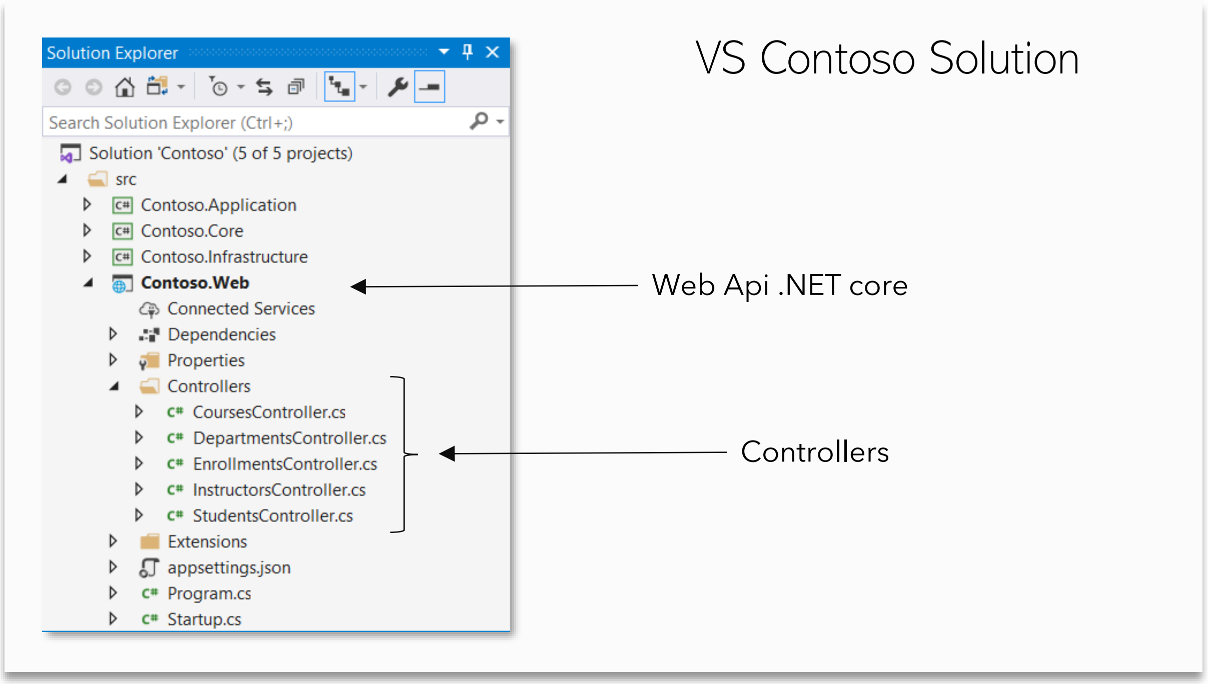
The Contoso application **approximates a Functional architecture Style embedded in a Clean architecture**.

The Contoso application is consists of four projects the three of them are class libraries targeting the .NET standard 2.0 framework (Contoso.Application, Contoso.Core, Contoso.Infrastructure) and the "UI" Project named Contoso.Web is a .NET 3.0 Web application project compatible with the rest of the projects



## 3.4 **Web Api**

This layer contains the code for the Web API endpoint logic including the Controllers. The API project for the solution will have a single responsibility, and that is only to handle the HTTP requests received by the web server and to return the HTTP responses with either success or failure.



We will handle exceptions and errors that have occurred in the Domain or Data projects to effectively communicate with the consumer of APIs. This communication will use HTTP response codes and any data to be returned located in the HTTP response body.

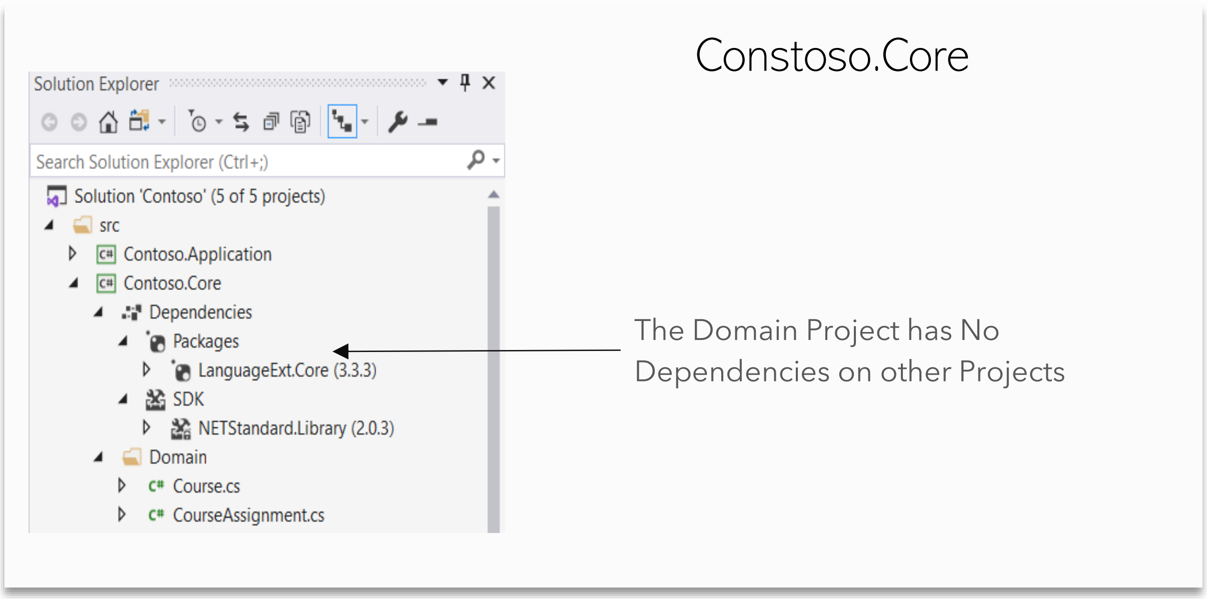
In ASP.NET Core Web API, routing is handled using Attribute Routing. If you need to learn more about Attribute Routing in ASP.NET Core go here. We are also using dependency injection to have the MediatR instance injected into the Controller.

```
[Produces("application/json")]
 [Route("api/[controller]")]
 [ApiController]
 public class CoursesController : ControllerBase
 {
     private readonly IMediator _mediator;
     public CoursesController(IMediator mediator) => _mediator = mediator;

     [HttpGet("{courseId}")]
     public Task<IActionResult> Get(int courseId) =>
         _mediator.Send(new GetCourseById(courseId)).ToActionResult();

     [HttpPost]
     public Task<IActionResult> Create(CreateCourse createCourse) =>
         _mediator.Send(createCourse).ToActionResult();

     [HttpPut]
     public Task<IActionResult> Update(UpdateCourse updateCourse) =>
         _mediator.Send(updateCourse).ToActionResult();

     [HttpDelete]
     public Task<IActionResult> Delete(DeleteCourse deleteCourse) =>
         _mediator.Send(deleteCourse).ToActionResult();
 }
```

Github: source

## 3.5 Domain Model

The Domain model here resides solely in the **Contoso.Core** project and has no dependencies on any other solution project or tool like Entity Framework for example.


Constoso.Core
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Contoso' (5 of 5 projects)
src
Contoso.Application
Contoso.Core
Dependencies
Packages
LanguageExt.Core (3.3.3)
SDK
NETStandard.Library (2.0.3)
Domain
Course.cs
CourseAssignment.cs
The Domain Project has No Dependencies on other Projects

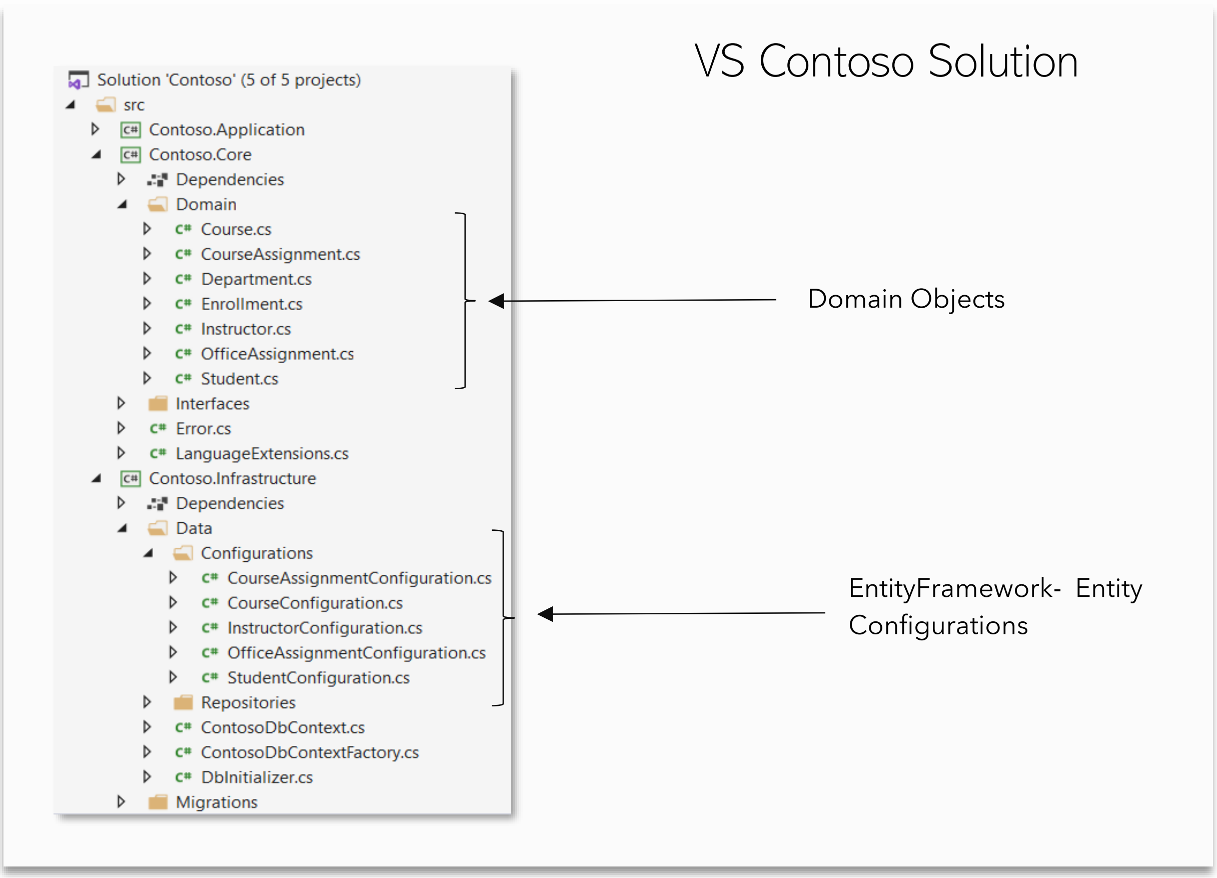
The domain model might seem trivial because in most implementations of the Clean architecture the domain objects are just POCO (Plain old CLR object) objects. For example:

```
public class Course
    {
        public int CourseId { get; set; }
        public string Title { get; set; }
        public int Credits { get; set; }
        public int? DepartmentId { get; set; }

        public Department Department { get; set; }
        public List<Enrollment> Enrollments { get; set; }
        public List<CourseAssignment> CourseAssignments { get; set; }
    }
```

Github: source

This is often seen as an anti-pattern in the Object oriented world and especially in DDD world where it has a special name: anemic domain model. Others see this as a form of proper design practice.

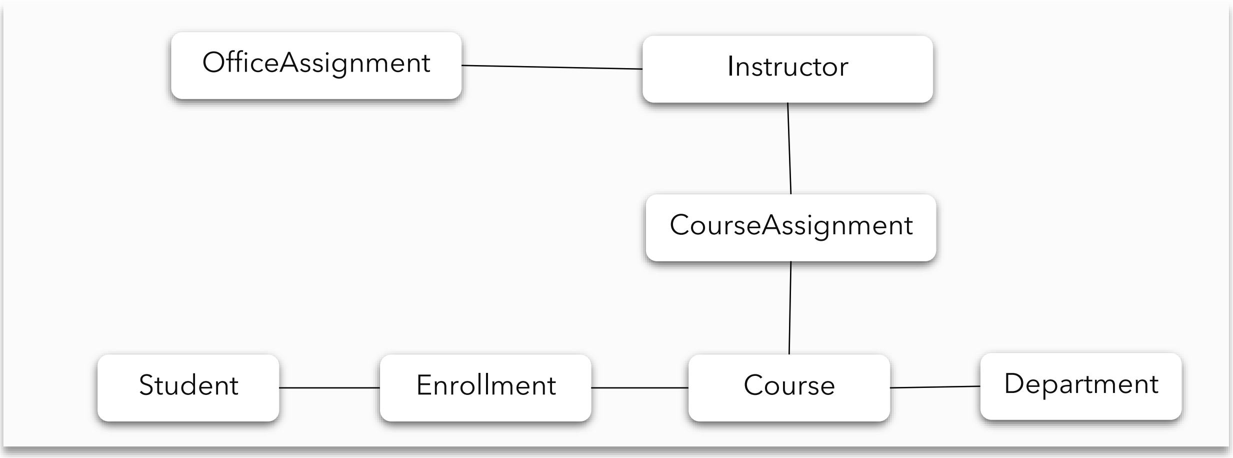
In small scale applications this is simply fine. What we are going to focus here is not where we are going to place the business logic but the use of the **Functional Types in the Domain Model**



VS Contoso Solution

```
Solution 'Contoso' (5 of 5 projects)
  src
    Contoso.Application
    Contoso.Core
      Dependencies
      Domain
        Course.cs
        CourseAssignment.cs
        Department.cs
        Enrollment.cs
        Instructor.cs
        OfficeAssignment.cs
        Student.cs
      Interfaces
      Error.cs
      LanguageExtensions.cs
    Contoso.Infrastructure
      Dependencies
      Data
        Configurations
          CourseAssignmentConfiguration.cs
          CourseConfiguration.cs
          InstructorConfiguration.cs
          OfficeAssignmentConfiguration.cs
          StudentConfiguration.cs
        Repositories
        ContosoDbContext.cs
        ContosoDbContextFactory.cs
        DbInitializer.cs
      Migrations
```

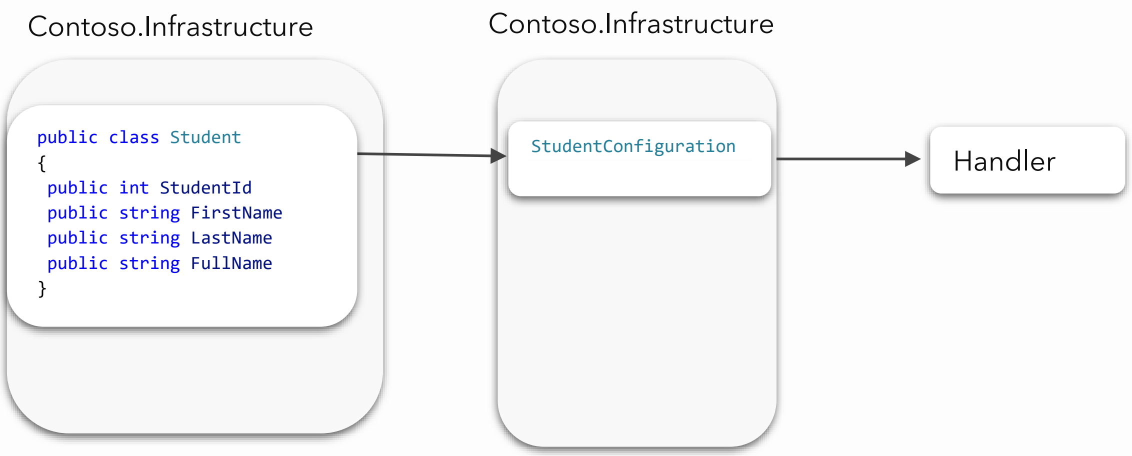
The complete Domain model of this application can be represented in a UML diagram



Github: source

**Code First**

The projects follow the Code first Entity Framework paradigm which means that there is no annotations or dependencies of the Domain Model to the Entity Framework. Instead the correlation between OOP entities and database Entities is achieved using configuration files



```
class StudentConfiguration :
   IEntityTypeConfiguration<Student>
{
    public void Configure(EntityTypeBuilder<Student> builder)
    {
        builder.Property(b => b.FirstName)
            .HasMaxLength(50);
        builder.Property(b => b.LastName)
            .HasMaxLength(50);
    }
}
```

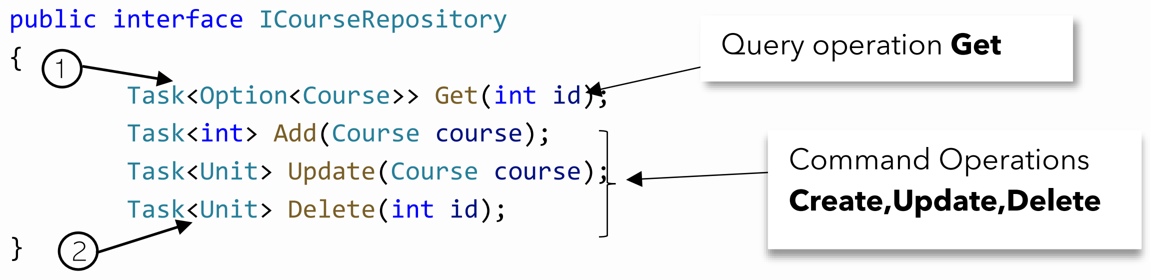
Github: source

### 3.5.1 Repositories

The repositories are the contact point of the Domain with the outer layers. And in this point is where we can use the functional structures of `Option` and `Either`. As you will see in most of the Repositories the `Get` operations have a similar signature:

```
Task<Option<T>> Get(int id)
```

For the methods that we want to retrieve a single Element by its Id we could also use the Either in the case that we care exposing any Errors in the Database.



```
public interface ICourseRepository
{
        Task<Option<Course>> Get(int id);
        Task<int> Add(Course course);
        Task<Unit> Update(Course course);
        Task<Unit> Delete(int id);
}
```

Github: source

Also, the result is wrapped in a `Task` because in most cases the Persistance using Entity Framework 6 can be made Asynchronous using `Async` methods for Save and Query .You can take a look at the Repositories implementations where the `SaveChangesAsync` and `SingleOrDefaultAsync` are used.