|  |
| --- |
|  |
| LINQ – Beyond Queries |
| Practical Functional Programming Techniques |

# Functional C#

## Objectives

After completing this lab, you should understand how to:

* Transform imperative code to a functional style
* Work with Func<> types in C# to do interesting things!

## Overview

In this lab we’ll rewriting imperative code with a functional style. You can compare and contrast the before and after to see which style you prefer.

## Part 1 – Stepping

1. Open the solution file in the before\Functional folder for this lab.
2. Press Ctrl+R, A to run all tests in the solution. All 4 tests should pass.
3. Open Stepper.cs from the Functional project and examine the DoSteps method.

Although simplified, DoSteps represents a business process we’ve all written at one time or the other. We need to make a series of calls to a component or web service and all the calls need to succeed or out operation fails. One way to write the series of steps is the imperative code we see inside DoSteps – essentially check the return value at each step.

1. Rewrite DoSteps in a declarative manner.

A declarative approach would be to hold the steps we need to execute in a data structure, like an array of Func<bool>. You can start with the code below.

Func<bool>[] steps =

{

Step1,

Step2,

Step3,

Step4

};

1. Implement the code to execute each step.

One approach is to loop through the available steps with a foreach loop.

foreach(var step in steps)

{

if(!step())

{

return false;

}

}

return true;

However, we can also use LINQ to achieve the same effect with less code. Can you execute the steps using LINQ?

public bool DoSteps()

{

Func<bool>[] steps =

{

Step1,

Step2,

Step3,

Step4

};

return steps.All(step => step() == true);

}

Note we could make the last line even shorter:

return steps.All(step => step());

1. Press Ctrl+R, A to run the unit tests – all should pass.
2. Rewrite the Step2 method so that the method will FAIL (return false) the first 2 times it is called, but return true for every subsequent call.

Here is one approach:

bool Step2()

{

step2Count += 1;

if(step2Count < 3)

{

return false;

}

return true;

}

int step2Count = 0;

1. Run the unit tests – the stepper tests should **FAIL**.

What we want to simulate here is a “step” that is performing network access over a flaky connection. Sometimes you don’t want to give up right away, but allow for some retry logic. Instead of embedding “try three times” into each particular method, we’ll implement some generic retry logic to reuse.

1. Add a new static class to the Functional project named RetryExtensions.
2. Implement an extension method for Func<bool> that will retry a function up to three times if the function returns false.

public static class RetryExtensions

{

public static bool WithRetry(this Func<bool> func)

{

int retryCount = 0;

bool result = false;

while(retryCount++ < 3 && !result)

{

result = func();

}

return result;

}

}

1. Use the WithRetry extension method in the Stepper DoSteps method.

return steps.All(step => step.WithRetry());

1. Run the unit tests – they should all pass.

This was a simple example of retry logic. For retrying a function that performs real network access, we’d need to add a try/catch block to the retry logic to catch network errors, and probably a Thread.Sleep to wait some time before retrying an operation (in hopes that the network was working again).

# Part II – Validation

1. Open Movie.cs from the Functional project.
2. Examine the Validate method.
3. Rewrite the Validate method in a functional style. You might find it helpful to build a ValidationRule class that pairs the rule to execute with an error message.

*One possible solution is to start with a ValidationRule class like the following.*

public class MovieValidationRule

{

public MovieValidationRule(Func<Movie, bool> predicate,

string message)

{

Predicate = predicate;

Error = new ValidationError(message);

}

public Func<Movie, bool> Predicate { get; set; }

public ValidationError Error { get; set; }

}

*Note the above class could easily be rewritten using generics to work with any type.*

*The next step would be to provide a property or method to build a collection of rules. The idea is that building this collection will leave the validation logic more readable than the imperative approach. It’s easier to add new rules without worrying about if/else statements. It would also be possible to return/inject/remove rules depending on a context (like using fewer rules for a system admin).*

IEnumerable<MovieValidationRule> Rules

{

get

{

yield return new MovieValidationRule

(

m => String.IsNullOrEmpty(m.Title),

"Title cannot be empty"

);

yield return new MovieValidationRule

(

m => m.Duration < 45 || m.Duration > 240,

"The duration is out of range"

);

yield return new MovieValidationRule

(

m=> m.ReleaseDate.Year < 1860 ||

m.ReleaseDate.Year > 2100,

"The release date is out of range"

);

}

}

*Finally, validation can use these rules in a one line LINQ statement. We filter the rules to the rules that return true, than transform (or map) those rules into a sequence of errors.*

public IEnumerable<ValidationError> Validate()

{

return Rules.Where(r => r.Predicate(this)).Select(r => r.Error);

}

1. Once your rewrite is complete, make sure all unit tests are still passing.

# Part III – Func<> As The New Interface

*In this scenario we’ll look at some code that is difficult to unit test, and apply a Func<> as a layer of indirection.*

1. Add a default constructor to the Movie class in the Functional project.
2. Using the constructor, set the default value for ReleaseDate to the current date and time.

public Movie()

{

ReleaseDate = DateTime.Now;

}

1. Add a new class to the Functional.Tests project with the name “MovieConstructionTests”.
2. Give the new class the [TestClass] attribute (from the Microsoft.VisualStudio.TestTools.UnitTesting namespace).
3. Write a test method to ensure the Movie constructor initializes the ReleaseDate property with the current date and time.

*The trick is – how can you write this test to work in a reliable fashion? DateTime.Now is not an input we can control from a unit test, at least not without an advanced testing tool.*

[TestClass]

public class MovieConstructionTests

{

[TestMethod]

public void Default\_release\_date\_is\_current\_date()

{

// ??

}

}

1. Add a **static** class to the Functional project named SystemTime.
2. Add a public, static field to SystemTime called Now. Now is of type Func<DateTime>, and initialize the field to return a Func which itself returns DateTime.Now.

public static class SystemTime

{

public static Func<DateTime> Now = () => DateTime.Now;

}

1. Implement the previous unit test to set SystemTime.Now to a known DateTime. Construct a new instance of Movie and assert that the ReleaseDate equals the SystemTime value.

[TestMethod]

public void Default\_release\_date\_is\_current\_date()

{

var expectedDate = new DateTime(2002, 6, 11);

SystemTime.Now = () => expectedDate;

var movie = new Movie();

Assert.AreEqual(expectedDate, movie.ReleaseDate);

}

1. Run the unit tests – the new test should fail.
2. Make the new test pass by changing the logic in the Movie constructor to use SystemTime.

public Movie()

{

ReleaseDate = SystemTime.Now();

}

1. Run the unit tests – all the tests should now pass.

# Summary

This lab should give you a taste of what functional programming has to offer. Do you like the functional approach? For “programming in the small” a functional approach often leads to cleaner code, with less branching and looping. In many cases you can make code declarative by describing what you want to do without getting caught up in the details of “how” to do it.