|  |
| --- |
|  |
| Functional C# |
| Applying 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#

## 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

};

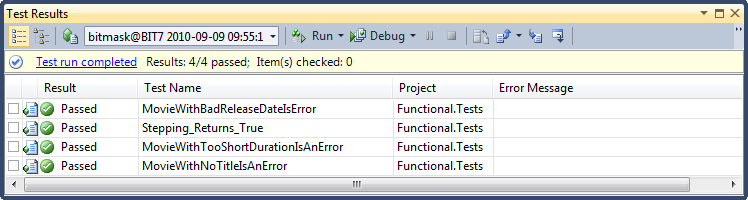
return steps.All(step => step() == true);

}

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. In the Test Results window, click on the “Test run completed” link.



1. In the test summary, notice the amount of time it took to execute the tests (it should be 4 seconds or more).

If our business logic doesn’t require us to run the steps in any particular order, and the steps generally return true unless there is an exceptional condition, then we might consider parallelizing the operations to increase the speed. Fortunately, LINQ makes this easy!

1. Use the AsParallel operator to spread the stepping work across multiple threads.

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

1. Run all the unit tests again, and compare the new execution time. The time should drop to under 2 seconds.

# 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.

# 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.