Swinging those testing tools in F# - Mikael Lundin

Abstract

F# is taking testing to another level by removing the ceremonial of setting up hierarchies of testing classes and mocking frameworks. Instead it lets us focus on what's most important, the code we want to test. By utilizing expressive and functional programming we swing those testing tools that makes test automation a joy, instead of repeated grudge it can be in C#.

Functional Programming and Testing

I heard of a guy who heard about a guy that shipped an application in Haskell. Writing pure functions can be a challenge, but writing pure programs are unbelievable hard.

Writing code in C# you spend a lot of time writing ceremonial code. One thing I love about F# is that I can spend my time focusing on the problem at hand and instead of writing cruft. Sometimes it ends up with a pure approach and sometimes not. The important thing is to get the work done.

Convincing your manager that your new project is going to be F# can be a challange. Who will maintain the code once you've gone to greener pastures? A good way to get started writing F#, and kind of sneeking it in, is to write your tests in F#. You can even test C# code using F# without any hassle, as they both compile to the same IL code.

Here I want to show you some of the tools you can use to start flying into the world of functional programming and testing.

F# interactive

One of the things lacking in C# is a REPL (read-eval-print loop). In F# it is a natural thing to work with fsx, script files, and evaluating code in the **F# interactive** before adding the code to an fs-file and compiling it into an assembly.

```
// join " " ["Hello"; "FSharp"]
// => "Hello FSharp"
let join (separator : string) =
    List.reduce (fun s1 s2 -> s1 + separator + s2)
```

Code Listing 1 - Function that joins a list of string together with a separator between words.

What I do when writing F# code is that I provide an example on how to execute a function as a comment just before, so I can easily evaluate it with **Alt+Enter** and also play around with the arguments.

Working this way interactivly with the REPL is one kind of testing that drives the design of your code in more proactive way than test-first development ever will. You can test out your functionality without even starting a debugger, and modify the code until it fits all the scenarios that you can come up with during development.

Xunit in F#

Testing functions interactivly through F# interactive will however not create a reproducable result. It will not provide the security needed when you want to refactor, like a great test suite will. Even if the functionality ”works” you will still need to write those unit tests to make sure it keeps on working.

What test runner is the best one, is a matter of personal preference. In C# I don’t think it matters much as long as you stay away from MSTest, the Microsoft Unit Test framework shipped with Visual Studio.

In F# there are some benefits of using Xunit, as it has a better extensibility story it provides a better experience than NUnit.

```
[<Fact>]
let ``join should separate two string with a space`` () =
    Assert.Equal<string>("Hello FSharp",
        join " " ["Hello"; "FSharp"])
```

Code Listing 2 - Using Xunit from F# looks pretty much the same as in C#.

The first thing you will notice when writing unit tests in F# is that you don’t need a class to hold the tests. You can write them directly in a new FS-file or organize them in different modules.

The second thing you will notice is that you can write natural language as the test name, and won’t have to use pesky camelcase or use underscore to delimit words from one another.

This results in a much easier to read test suite and means alot when a test is failing and you get the testname in clear text.

FsUnit

The testing story of F# starts to become really interesting when you take functional programming into account. The FsUnit framework take advantage of the forward pipe operator in order to make asserts more expressive.

```
[<Fact>]
let ``join should create a comma separated list`` () =
    join "," ["1"; "Mikael"; "Lundin"]
    |> should equal "1,Mikael,Lundin"
```

Code Listing 3 - FsUnit let us express asserts in a more expressive way

FsUnit framework is not a test execution library but an assert library that will help us write tests that are easy to understand from a glance.

It is also useful when we want to assert something in a new way. In this FsUnit provides an easy extensibility model that will make new asserts trivial to implement. Consider wanting to use regular expressions to assert on the test result.

Consider the following test where **`match'`** is an assertion that we've invented ourselves. It takes a regular expression as and argument and matches that to the join result.

```
[<Fact>]
let ``join should separate words using space`` () =
    join " " ["lions"; "sleeps"; "tonight"]
    |> should match' "\w+(\s\w+){2}"
```

Code Listing 4 - Should match' is an assertion that is not built into FsCheck, and it takes a regular expression for matching

We implement this assertion in a few lines of code based on NHamcrest.

```
let match' p =
    CustomMatcher<obj>(sprintf "Matches %s" p,
        fun c ->
            match c with
            | :? string as str -> Regex.IsMatch(str, p)
            | _ -> false)
```

Code Listing 5 - Implementation of a match' assertion that will assert result with a regular expression.

With just a few lines of code we can extend the FsUnit assert library and create our own asserts, making this a very powerful library for creating expressive tests.

Unquote

Another impressive assertion framework for F# is Unquote. It takes advantage of the quoted expressions feature of F#, evaluating one statement at a time until it comes down to a primitive result.

```
<@ 1 + 2 = 3 @>

val it : Quotations.Expr<bool> =
  Call (None, op_Equality,
      [Call (None, op_Addition, [Value (1), Value (2)]), Value (3)])
```

Code Listing 6 - A quoted expression returns the abstract syntax tree for the expression

Unquote provides tools for decompiling quoted expressions and evaluating them, but first and mostly it provides the means of reducing a quoted expression down to its primitive parts.

```
unquote <@ (30 + 6) / 3 = (3 * 7) - 9 @>

(30 + 6) / 3 = 3 * 7 - 9
36 / 3 = 21 - 9
12 = 12
true

val it : unit = ()
```

Code Listing 7 - Unquote will reduce both sides of the equal sign until the result is primitive

This becomes very useful in testing when tests are failing and it becomes tricky to figure out what value what asserted to an expected value.

```
[<Fact>]
let ``join should ignore empty strings`` () =
  test <@ join " " ["hello"; ""; "world"] = "hello world" @>
```

Code Listing 8 - Using unquote in a Xunit test scenario

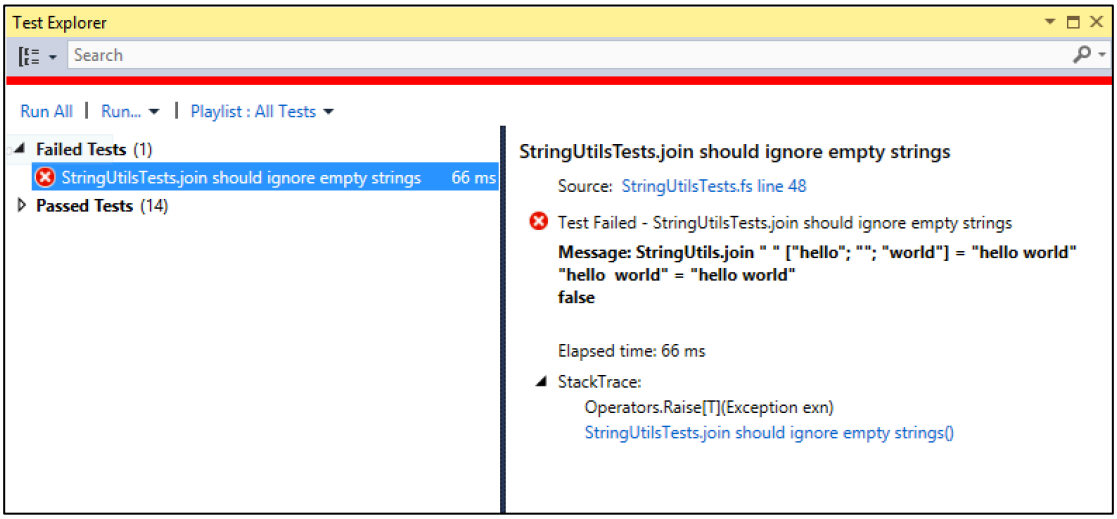

Figure 1 - When the unquote test fails, it will present the complete reduction of the quoted code expression

Having this tool around when working with F# code an especially algorithms and calculations is invaluable. You can reduce statements directly from FSI or evaluating code though FSX script files.

Mocking

What came first, mocking or dependency injection? When you need to unit test some code that have external dependencies, you create a fake instance of that dependency, sometimes called a mock or a stub, which you inject in order to easily switch it out.

I usually call these test fakes, to avoid distinguish stubs from mocks, but basically a stub is just something filling out the blanks and a mock is when we want to assert on the actual interaction of the fake object.

The most trivial way of stubbing some data, is to send in a function as dependency with the correct function signature. This method is fine to use in C#, but often forgotten as it's not a functional first programming language.

```
type Score = { Name : string; Score : int }

let getHighScore (csvReader : string -> string list list) =
    csvReader "highscore.txt"
    |> List.map (fun row ->
        match row with
        | name :: score :: [] ->
            { Name = name; Score = score |> int }
        | _ -> failwith "Expected row with two columns")

    |> List.sortBy (fun score -> score.Score)
    |> List.rev
```

Code Listing 9 - Reading highscore from a text file and returning a list of Scores sortend in descending order

Here the dependency is a function that takes the path to the CSV file as argument and returns a string list list, simply a matrix of columns and rows.

One thing we can do in F# but not in C# is to create a type alias for the dependency function. The closest we get to this in C# is the interface.

```
type CsvReader = string -> string list list

let getHighScore (csvReader : CsvReader) =
    // ...
```

Code Listing 10 - We create an alias for the function signature and call it CsvReader

Writing the code to test this will not require us to introduce any mocking tools or frameworks. We will simply supply a function that fulfills the function signature, and that way fill the dependency of the function.

```
[<Fact>]
let ``should return highscore in descending order`` () =
    // arrange
    let getData (s : string) = [
            ["Mikael"; "1234"];
            ["Jonas"; "321"];
            ["Bill"; "4321"]]

    // act
    let result = getHighScore getData

    // assert
    result |> List.map (fun row -> row.Score)
    |> should equal [4321; 1234; 321]
```

Code Listing 11 - The getData function fulfills the function signature string -> string list list and will act as dependency stub to CsvReader

Sometimes it is appropriate to work with interfaces, especially when you need to bundle more functions and properties into one dependency.

```
type ICsvReader =
    abstract member FileName : string
    abstract member ReadFile : unit -> string list list

let getHighScore (csvReader : ICsvReader) =
    csvReader.ReadFile()
    // ...
```

Code Listing 12 - When you want to bundle functions and properties into a dependency, it is best to use an interface

Traditionally when testing this in C# we would have to use a mocking framework to create a fake ICsvReader or we would make a stub implementation of it. In F# we can use object expressions to create an instance of this interface immediatly, as long as we supply an implementation for all its members.

```
[<Fact>]
let ``should return highscore in descending order`` () =
    // arrange
    let csvReader =
        { new ICsvReader with
            member this.FileName = "highscore.txt"
            member this.ReadFile () = [
                ["Mikael"; "1234"];
                ["Jonas"; "321"];
                ["Bill"; "4321"]]
        }

    // act
    let result = getHighScore csvReader

    // assert
    result |> List.map (fun row -> row.Score)
    |> should equal [4321; 1234; 321]
```

Code Listing 13 - With object expression we can create an instance of the ICsvReader interface by supplying implementations for its members

There is a small bit of waste associated with this method, where FileName member must be implemented, but is never actually used. This becomes unmaintainable when dealing with really large interfaces.

Foq is a simple mocking framework that takes after Moq, but with some functional elements. It will allow us to create fake objects where we only supply an implementation for a part of an interface.

```
[<Fact>]
let ``should return highscore in descending order`` () =
    // arrange
    let csvReader =
        Mock<ICsvReader>()
            .Setup(fun da -> <@ da.ReadFile() @>)
                .Returns([["Mikael"; "1234"];
                          ["Jonas"; "321"];
                          ["Bill"; "4321"]])
            .Create()

    // act
    let result = getHighScore csvReader

    // assert
    result |> List.map (fun row -> row.Score)
    |> should equal [4321; 1234; 321]
```

Code Listing 14 - The Foq mocking framework allow us to create fake objects out of interfaces without supplying implementations for all its members

The obvious benefits of using a mocking framework is that the coupling between the test and the interface becomes less, meaning the test itself is less brittle in terms of changes in the interface.

The syntax for mocking is as always a handful and may lessen the readability and in long term maintainability of the test.

TickSpec

There is a very popular testing tool for C# called SpecFlow, which allows you to create executable specifications out of a DSL called Gherkin.

TickSpec is the F# equivalent of this framework, and it uses some of the F# features to make executable specifications an even smoother experience.

Consider the following feature file for Conway's Game of Life.

```
Feature: Conway's Game of Life

Scenario 1: Any live cell with fewer than two live
neighbours dies, as if caused by under-population.
      Given a live cell
      And has 1 live neighbour
      When turn turns
      Then the cell dies

Scenario 2: Any live cell with two or three live neighbours
lives on to the next generation.
      Given a live cell
      And has 2 live neighbours
      When turn turns
      Then the cell lives

Scenario 3: Any live cell with more than three live
neighbours dies, as if by overcrowding.
      Given a live cell
      And has 4 live neighbours
      When turn turns
      Then the cell dies

Scenario 4: Any dead cell with exactly three live neighbours
becomes a live cell, as if by reproduction.
      Given a dead cell
      And has 3 live neighbours
      When turn turns
      Then the cell lives
```

Code Listing 15 - Gherkin is a DSL that structures your tests into specifications that are human readable

The definition file we write that will make these specifications executable, can be written in a way that is very bare bone and readable.

```
let mutable cell = Dead(0, 0)
let mutable cells = []
let mutable result = []

let [<Given>] ``a (live|dead) cell`` = function
    | "live" -> cell <- Live(0, 0)
    | "dead" -> cell <- Dead(0, 0)
    | _ -> failwith "expected: dead or live"

let [<Given>] ``has (\d) live neighbours?`` (x) =
    let rec _internal x =
        match x with
        | 0 -> [cell]
        | 1 -> Live(-1, 0) :: _internal (x - 1)
        | 2 -> Live(1, 0) :: _internal (x - 1)
        | 3 -> Live(0, -1) :: _internal (x - 1)
        | 4 -> Live(0, 1) :: _internal (x - 1)
        | _ -> failwith "expected: 4 >= neighbours >= 0"
    cells <- _internal x

let [<When>] ``turn turns`` () =
    result <- GameOfLife.next cells

let [<Then>] ``the cell (dies|lives)`` = function
    | "dies" -> GameOfLife.isDead (0, 0) result
                |> should be True
    | "lives" -> GameOfLife.isLive (0, 0) result
                 |> should be True
    | _ -> failwith "expected: dies or lives"
```

Code Listing 16 - In SpecFlow we use attributes to match specifications to regular expressions. In TickSpec we can use the definition name as a regular expression in itself

Compared to SpecFlow there is no code generation that needs to take place in order to execute the specifications, which gives TickSpec a lower bar for implementation. There is no need of a Visual Studio extension, we can just import TickSpec with NuGet and start writing features and definitions.

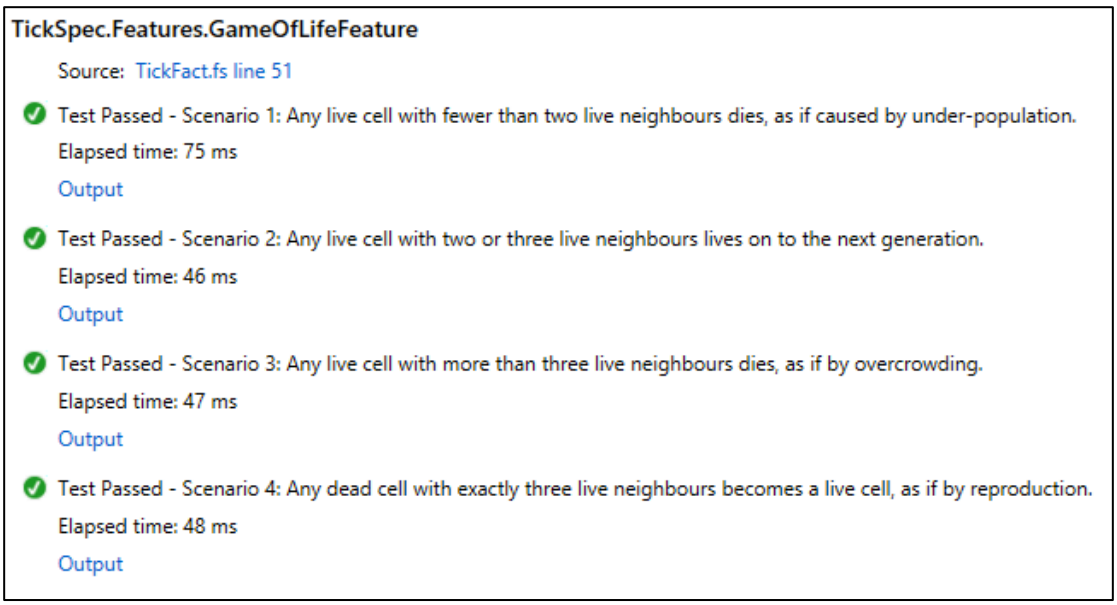

Figure 2 - We can run our tests without the need of code generation or a third part Visual Studio plugin

Canopy

When it comes to web testing, Selenium has won the war against WatiN, which was a port of the Ruby framework WatiR. However, Selenium does not provide a very good functional API and this is the problem that Canopy tries to fix.

Canopy is a functional API on top of Selenium. A nice little tool that will let you write shorter and more expressive web tests, without worrying about the details.

```
[<EntryPoint>]
let main argv =

    start firefox

    context "Mikael Lundin"

    "should contain my name" &&& fun _ ->
        // navigate
        url "http://mikaellundin.name"

        // assert
        "h1.name" == "Mikael Lundin"

    run()

    quit()

    0 // return an integer exit code
```

Code Listing 17 - This code will open a firefox browser, navigate to start page and assert that h1.name contains my name

Canopy does not integrate to any existing testing frameworks. Instead you create a new console application and run your tests directly from the compiled executable.

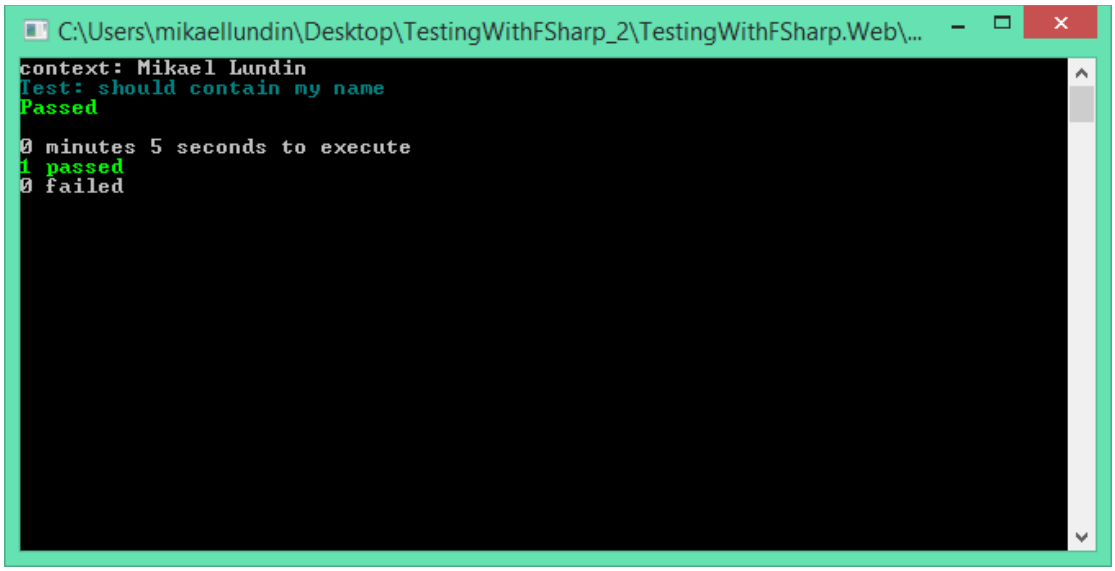

Figure 3 - Canopy executes directly in a console application, not integrating to any external testing framework

Canopy is able to create output that integrates into TeamCity so you can get test output into your build server.

FsCheck

Up until this point you have not seen anything new. All that has been presented are old technologies and methods utilizing a new format, the F# syntax.

F# has an equivilent framework to Haskell's quicktest that has not been generally available to the .NET languages before. FsCheck is a manifestation of property-based testing with F# as a platform.

Property-based testing is a different approach to testing. Unit testing is a tool for developing code, making sure that we can refactor code as we develop and maintain an application.

Integration testing is meant to help us verify our code works together with external systems, databases and services.

Executable specifications are white box tests, that will verify that the system fulfills the specification, and resume regression in that the system keeps fulfilling the specification even after the implementation has changed.

Manual testing is what we call exploration of the system. This is a creative task where a human will make sure, not only that the system solves the problem it was designed to solve, but also that it solved the right problem.

Where does property-based testing fit into this?

Property-based testing is exploration of the system, together with automation. In property-based testing you setup a set of properties that always should be true about the system, and then you explore it with data.

Let's assume that we have a sorting function with an implementation like this.

```
// insort [2; 7; 4; 5; 3]
// => [2; 3; 4; 5; 7]
let insort list =
    let rec _insert item = function
    | [] -> [item]
    | hd :: tl when item < hd -> item :: hd :: tl
    | hd :: tl -> hd :: (_insert item tl)

    list |> List.fold (fun acc item -> _insert item acc) []
```

Code Listing 18 - This function will sort a list of numbers by insertion sort algorithm

We can write a simple unit test that will cover all the lines of this function.

```
[<Fact>]
let ``insort should make a list of numbers ordered`` () =
    insort [6; 3; 7; 2; 4] |> should equal [2; 3; 4; 6; 7]
```

Code Listing 19 - This test completely covers the insort function

Even if this test technically covers the whole sorting function, is it really enough to say that we trust insort to work without flaws? How many tests do we need to write in order to feel that its covered?

Instead we could define a set of properties that always should hold true.

```
[<Property>]
let ``sorting once equals sorting twice`` (l : int list) =
    insort l = insort (insort l)
```

Code Listing 20 - A property is a piece of code that should be true for any data

A common property for a pure function like this, is that the result should be same if running the same function again on the result. Our property could claim that this should be true for any data.

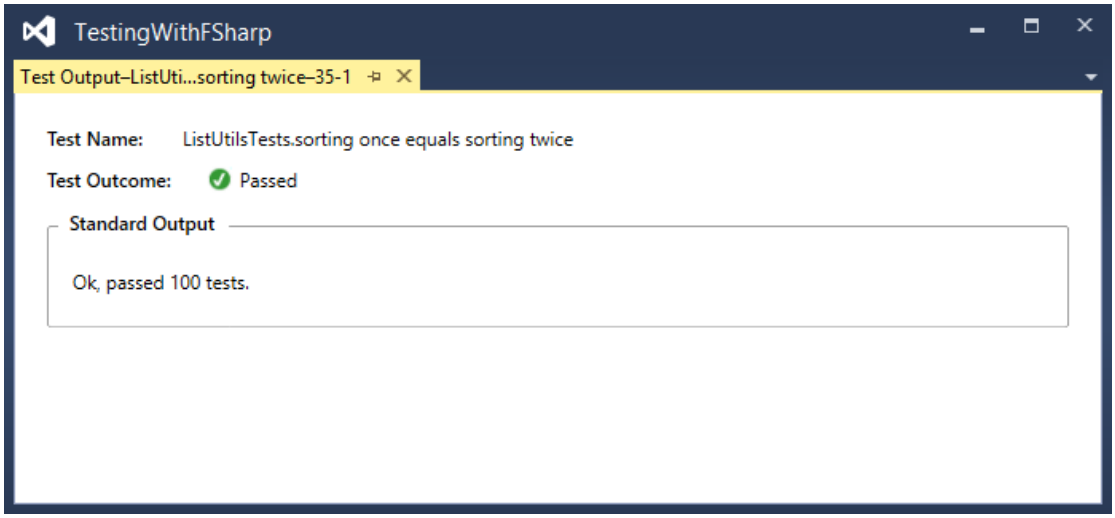

Figure 4 - When running the test, FsCheck will generate 100 data sets for the property to verify that it holds true

Now we can start implementing some more interesting properties for our function.

```
[<Property>]
let ``first is smallest element`` (list : int list) =
    (not list.IsEmpty) ==>
        lazy(List.min list = List.head (insort list))

[<Property>]
let ``last is largest element`` (l : int list) =
    (not l.IsEmpty) ==>
        lazy(List.max l = List.head (List.rev (insort l)))

[<Property>]
let rec ``is ordered result`` (l : int list) =
    let rec _ordered = function
    | [] -> true
    | hd :: [] -> true
    | fst :: snd :: tl ->
        (fst <= snd) && (_ordered (snd :: tl))
    _ordered (insort l)
```

Code Listing 21 - The following properties should always be true for the sort function

We can supply conditions for when the property is true and we can also write our own data generators to make sure that FsCheck generates appropriate data for our properties.

The difference to ordinary unit testing is that we use properties to explore our code, more than we use it to verify our code is working. By challanging our solution with lots of data we will find the edge cases that might make our code break in production.

Summary

Functional programming in F# changes the way we write code, and the way we test. Testing becomes much more interactive by verifying our code in the REPL before adding it to the compiled assembly.

With functional programming we are able to write code without side effects that is easier to test, but not only that; F# rejuvinates the world of automated testing by bringing some new frameworks and techniques to the table.

Testing with F# does not only allow you to test code written with F# but it also stretches into any .NET solution and allow you to write testing code that is both expressive and terse, allowing you to focus on what really matters – the functionality of your solution.

With this I hope that next time you create a test project, you consider doing it in F#.

Biography

Mikael Lundin is a software developer living in Malmö, Sweden. He started programming in Pascal 20 years ago and has been enjoying the craft both professionally and as a hobby through languages and frameworks such as PHP, C#, F#, Ruby, and Node. He has been a practitioner and mentor of test-driven development and agile methodologies for the last 8 years, helping teams succeed in delivering high-quality software.

Mikael has been working with F# for 4 years, providing solutions to clients, publicly speaking about functional programming, and having seminars for colleagues to spread the word. He strongly believes that functional programming is the future of delivering high-quality software.

Mikael Lundin is author of the book Testing with F# that was published 21 February 2015, by PACKT Publishing.

http://amzn.com/1784391239

Samples of presentations by Mikael Lundin.

- **(sv) Valtech Day 2012 - Funktionell programmering på riktigt**
  https://vimeo.com/24819589

- **(sv) Valtech .NET Day 2011 – Code Contracts**
  https://vimeo.com/16091425

- **(sv) Valtech .NET Day 2011 – Testbara webbplatser**
  https://vimeo.com/15122562

- **(sv) Interception med Castle DynamicProxy, Microsoft Unity eller LinFu**
  - https://vimeo.com/15449390
  - https://vimeo.com/15504788