2010 Seventh International Conference on Information Technology

A Case Study of the Factors Associated with using the F# Programming Language for Software Test Automation

James D. McCaffrey
Volt Information Sciences, Inc.
Redmond, WA 98052

Adrian Bonar
The Microsoft Corporation
Redmond, WA 98052

Abstract

Functional programming languages, which emphasize a paradigm in which code modules return a value, have traditionally been used primarily in research and academia rather than in commercial software development and testing. This paper presents the results of a case study which investigated the use of a functional programming language for writing software test automation. Several software test engineers with significant procedural programming experience, but minimal functional programming experience, were given a short training class which focused on writing test automation using the F# functional programming language. Survey results suggested that the technical factors (such as immutability and pipelining) associated with test automation written using a functional programming language were less important than the subjective factors (such as the similarities between programming paradigms), and that the use of a functional programming language may provide indirect value to a software testing effort.

Keywords: Functional programming, programming languages, software quality, software testing, test automation.

1. Introduction

This paper presents the results of a case study of the perceived value of technical and subjective factors associated with using a functional programming language for writing software test automation which verifies the functionality of a software application written using an imperative programming language. An imperative programming paradigm (also called a procedural paradigm) is essentially a loosely defined collection of coding techniques with an emphasis on sequences of instructions which manipulate system state. A functional programming paradigm emphasizes the construction of a computer program as a collection of mathematical-like functions which return values and can be composed together. There is no clear delineation between the two programming paradigms and both paradigms can be used within a particular program or system [1].

Programming languages can be characterized based on the extent to which they support and encourage either imperative programming or functional programming. Examples of programming languages which are generally categorized as imperative languages include C, C++, Java, C#, and Visual Basic. Examples of programming languages which are generally categorized as functional languages include Scheme, LISP, Haskell, and OCaml. Functional programming languages have traditionally been used primarily in research and academia rather than for commercial software development and testing [2]. This study was motivated in large part by the creation of the F# programming language. Although F# supports both imperative and functional programming styles, the language is derived from languages usually classified as either functional languages or mixed-model languages and is generally described as strongly encouraging a functional programming style. The F# language is the first primarily-functional programming language to be included as part of the Microsoft Visual Studio integrated development environment application, and as such has received significant marketing support and has significant importance and visibility to the field of commercial application program development. The literature on programming languages lists several claimed technical advantages of using functional programming versus using imperative programming [3]. The purpose of this study was to investigate some of these assertions of functional programming language advantages, as implemented by the F# language, in a production environment.

F# is a typed functional programming language which is fully compliant with the .NET Framework [4]. The F# language was derived in part from the hybrid function-imperative language OCaml (Objective Caml) which in turn was derived from Caml (Categorical Abstract Machine Language), which in turn was

978-0-7695-3984-3/10 $26.00 © 2010 IEEE
DOI 10.1109/ITNG.2010.253

derived from a fully functional language named ML (Meta Language). F# also borrows features from other programming languages, notably Haskell, in particular sequence expressions and workflows. In spite of F#'s relationship to OCaml and Haskell, F# is substantially different from pure functional programming languages especially with regards to F#'s object-oriented characteristics and .NET related characteristics such as dynamic loading and reflection. F# is therefore perhaps best characterized as a multi-paradigm programming language that encourages functional programming. F# will be distributed as a fully supported language in the .NET Framework and Visual Studio ecosystem as part of Visual Studio 2010.

2. Experiment

This study was intended to answer three research questions. First, to what extents do software test engineers value certain technical features of a functional programming language when used to write test automation? Second, to what extent will software test engineers actually make use of those technical features of a functional programming language which they value? Third, to what extents do software test engineers value certain subjective features of a functional programming language when used to write test automation? The overall design of the study was to simulate a software production environment by exposing a group of software engineers, who had significant experience with imperative style programming but little or no experience with functional programming, to a fundamental software test automation program implemented with the F# programming language, and then gauge the opinions of those engineers using a survey. Specifically, a group of six professional software engineers were given a four-hour training session which explained how to perform Web application HTTP request-response testing with a test harness program written using the F# language. The premise is that a cohesive example of realistic test automation presented in context better exposes F# and functional programming characteristics than a set of isolated code snippets. A disadvantage of this approach was that only certain features of F# and functional programming were exposed by the test harness. After the training session the engineers completed a short survey which addressed the three research questions presented above.

2.1. The Web application under test

The system under test used in this study was a simple but representative calculator Web application which runs in a Web browser and accepts two integer inputs and an operation (addition or subtraction) from the user, sends those inputs as an HTTP request to a Web server where the result is computed and used to create an HTTP response in the form of an HTML page which is then returned to the client browser and rendered. The application under test is illustrated in Figure 1.

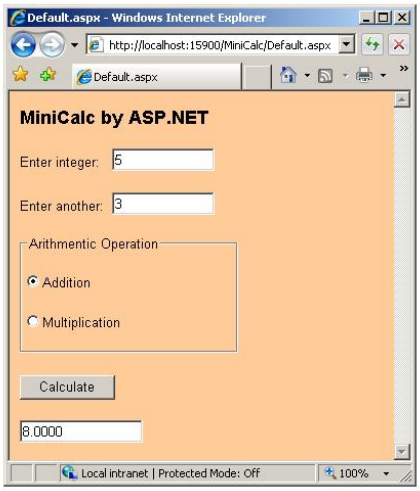

Figure 1. Web application under test.

The Web application under test was implemented using ASP.NET technology but the study test harness can, with minor modifications, exercise any Web application. The most fundamental form of Web application test automation is HTTP request-response testing [5]. An HTTP request-response test harness programmatically sends an HTTP request to the Web application under test which simulates user input, then fetches the resulting HTTP response as an HTML stream, and then examines the response for an expected value in order to determine and log a test case pass/fail result.

2.2. The test automation harness

The F# test automation harness which served as the training vehicle for the study was an adaptation of a harness originally created to illustrate characteristics of the F# language to software professionals [6]. The test harness contained code which illustrated seven characteristics of functional programming in general or the F# language in particular. A sample run of the test harness is shown in Figure 2.

```
C:\Harness>harness.exe

Begin F# HTTP request-response test
URL test = http://localhost:15900/MiniCalc.aspx

================================
Test case: 001
Comment  : Add 5 and 3
Input    : TextBox1=5&...Button1=clicked
Expected : value="8.0000"
Pass
================================

================================
Test case: 002
Comment  : Multiply 5 and 3
Input    : TextBox1=5&...Button1=clicked
Expected : value="15.0000"
Pass
================================

Number pass = 2
Number fail = 0

End test run
C:\Harness>
```

Figure 2. Sample test run.

The first F# functional programming paradigm characteristic exposed by the study test harness is implicit strong data typing:

```
printfn "Begin F# HTTP request-response test"
let url =
"http://localhost:15900/MiniCalc/Default.aspx"
printfn "URL under test = %s \n" url
```

Here the let keyword is used to bind a string value to an identifier named URL. As an alternative to explicitly specifying a data type of string, F# allows programmers to implicitly specify a type. This is a strong data typing mechanism; implicitly determined data types are not dynamic and cannot change at run time. Note that implicit strong data typing is a characteristic which is directly associated with the F# language, but only indirectly associated with functional programming in general.

The second F# characteristic exposed by the harness is the tuple data type:

```
let getVSandEV (url : string) : (string *
string)  =
  let wc = new WebClient()
  let st = wc.OpenRead(url)
  . . .
  (viewState, eventValidation)
```

Here the test harness defines a function named getVSandEV() which sends an initial priming request to the Web application under test in order to determine the application's ViewState and EventValidation values. These two values are used by ASP.NET applications to maintain state and are required when programmatically sending HTTP requests to such applications. The function signature indicates that the function accepts a single string input argument and returns a 2-tuple of string values. Note that functions are assumed to return a value, which is the last value in a function's definition, and so the use of a return keyword is not necessary.

The third F# and functional programming characteristic illustrated by the test harness is object immutability by default:

```
let testCases =
    [ for line in TestData do
        let tokens = line.Split(':')
        yield { Id = Int32.Parse(tokens.[0]);
                Description = tokens.[1];
                Operation = tokens.[2];
                Input = tokens.[3];
                Expected = tokens.[4]} ]
```

By default, most F# objects are immutable and defining test case data as an immutable list, signified by "[" and "]", rather than a mutable array, normally signified by "[|" and "|"], is a more thematic functional language approach. Here the harness makes use of an immutable list to define test case input data consisting of a test case identifier, description, operation, input, and expected value. Mutable objects allow and encourage state transformation which is a violation of a fundamental functional programming principle. Inline list generation can be leveraged, as it is here, to define a complex list without violating functional programming fundamentals.

The fourth and fifth F# and functional programming characteristics illustrated by the test harness are pipelining and lambda expressions:

```
TestData |>
  Seq.iter(fun (test) ->
    printfn "Test case: %A" test.Id
    ...
```

Here the list containing test case data is piped to the Seq.iter() (sequence iterator) function which accepts a lambda expression defined by the F# "fun" keyword. The "|>" pipeline operator can be used to pass information to F# functions and is more thematic for F# programs than passing information as function arguments. The Seq.iter() function iterates through an enumerable collection and is more thematic than using an explicit looping control structure such as one defined by a foreach or similar keyword. The use of lambda expressions, which define anonymous functions, is very thematic of the F# language and functional programming in general.

The sixth F# and functional programming characteristic illustrated by the test harness is the principle of pattern matching:

```
match (test.Operation) with
| "Addition"       -> TurnOnTest test
| "Multiplication" -> TurnOffTest test
| _                -> printfn "Error"
```

The 'match' keyword defines the source object of the pattern, in this case the test's operation, and the "|" operator is used to match the source against a pattern. Pattern matching is more thematic in functional programming than alternatives such as decision control using an if...then or similar statement.

In F#, pattern matching also aids well in exception handling:

```
try
  ...
with
  | :? System.Exception as e ->
    printfn "Fatal: %s" e.Message
```

Here the try...with construction is used to capture and handle any test harness runtime exceptions. In this case the ":?" token is used to indicate matching against a type rather than matching against a value.

The seventh F# and functional programming characteristic illustrated by the test harness was the principle of required handling of return values:

```
printfn "\nEnd F# test run"
Console.ReadLine() |> ignore
```

Here the ReadLine() method is used simply to pause test harness execution until the user presses the <enter> key. The ReadLine() method returns a string value. In non-functional programming languages the return value can be typically be ignored. In F#, function return values must be explicitly handled, in this case by piping the return to the special ignore keyword.

3. Results

A four-hour training class based on the test harness described in the previous section of this paper was delivered to six professional software test engineers. The experience and skill levels of the subjects were not controlled or measured, but there was no reason to believe that the subjects' backgrounds were atypical in any way. Following the training class, a 15-question survey was administered to the subjects. For each of the seven F# and functional programming characteristics the subjects were asked to evaluate the usefulness of the characteristic on a scale of 1 to 5 (not at all useful, somewhat useful, neutral, useful, very useful). Additionally, the six subjects were asked to evaluate the likelihood that they would use each technique, if it were available, when programming with an imperative language on a scale of 1 to 5 (not at all likely, somewhat likely, neutral, likely, very likely). Finally, subjects were asked to comment on their impressions of writing software test automation using the F# language. The results of the survey are shown in Table 1.

Table 1. Survey results.

	Utility	Likelihood
Implicit strong typing	2.33	3.00
Tuple data type	3.67	4.67
Immutability by default	3.33	2.83
Pipelining	1.67	1.67
Lambda expressions	2.17	2.83
Required return value handling	2.67	2.33
Meta pattern matching	2.00	2.00

The results suggest that of the seven characteristics illustrated by the F# test harness, the tuple data type and the immutability by default principle were most valued by the subjects. Additionally, implicit strong data typing and the tuple data type were the two features most likely to be used when writing test automation with an imperative programming language. Four of the comments made by the six subjects were fairly similar. Five of six of the subjects indicated that F# has the feel of a scripting language and would most likely be useful for writing relatively small test harness programs rather than for creating large test systems. Four of six subjects indicated that F# has a rather unusual feel compared to imperative languages which would inhibit them from adopting the language unless required to do so. Four of six subjects commented that, even though it seemed to be a minor characteristic, the tuple data type was very useful. Finally, all six subjects commented to the effect that they felt the most useful aspect of writing test automation with a functional

programming language was the indirect effect of stimulating thought about how functional programming principles can be used with imperative languages.

4. Conclusions

Because this is an informal case study rather than a formal experimental study, it is not possible to draw any definitive conclusions from the results. That said however, the results do seem to suggest that software test engineers may derive value from being exposed to test automation written using the F# language, but that this value is more of a subjective nature (as indicated by survey comments) than a consequence of the technical features of the F# language. The results of this study suggest that there is reason to warrant additional investigations of the factors associated with writing software test automation using the F# language, including controlled research experiments which investigate possible test automation productivity gains and improved quality of a target system under test.

5. References

[1] Hongwei Xi, "Imperative Programming with Dependent Types", *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science*, 2000, pp. 375-387.

[2] P. Wadler, "Why No One Uses Functional Languages", *ACM SIGPLAN Notices*, 1988, pp. 23-27.

[3] R. Govindarajan, "Exception Handlers in Functional Programming Languages," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, Aug. 1993, pp. 826-834.

[4] Syme, Don, Granicz, Adam, and Cisternino, Antonio, *Expert F#*, Apress Publishing, New York, 2007.

[5] McCaffrey, James, *.NET Test Automation Recipes: A Problem-Solution Approach*, Apress Publishing, New York, 2006.

[6] J. McCaffrey, "Request-Response Testing with F#", *MSDN Magazine*, vol. 24, no. 7, July 2009.