Extracted from:

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

This PDF file contains pages extracted from *Domain Modeling Made Functional*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

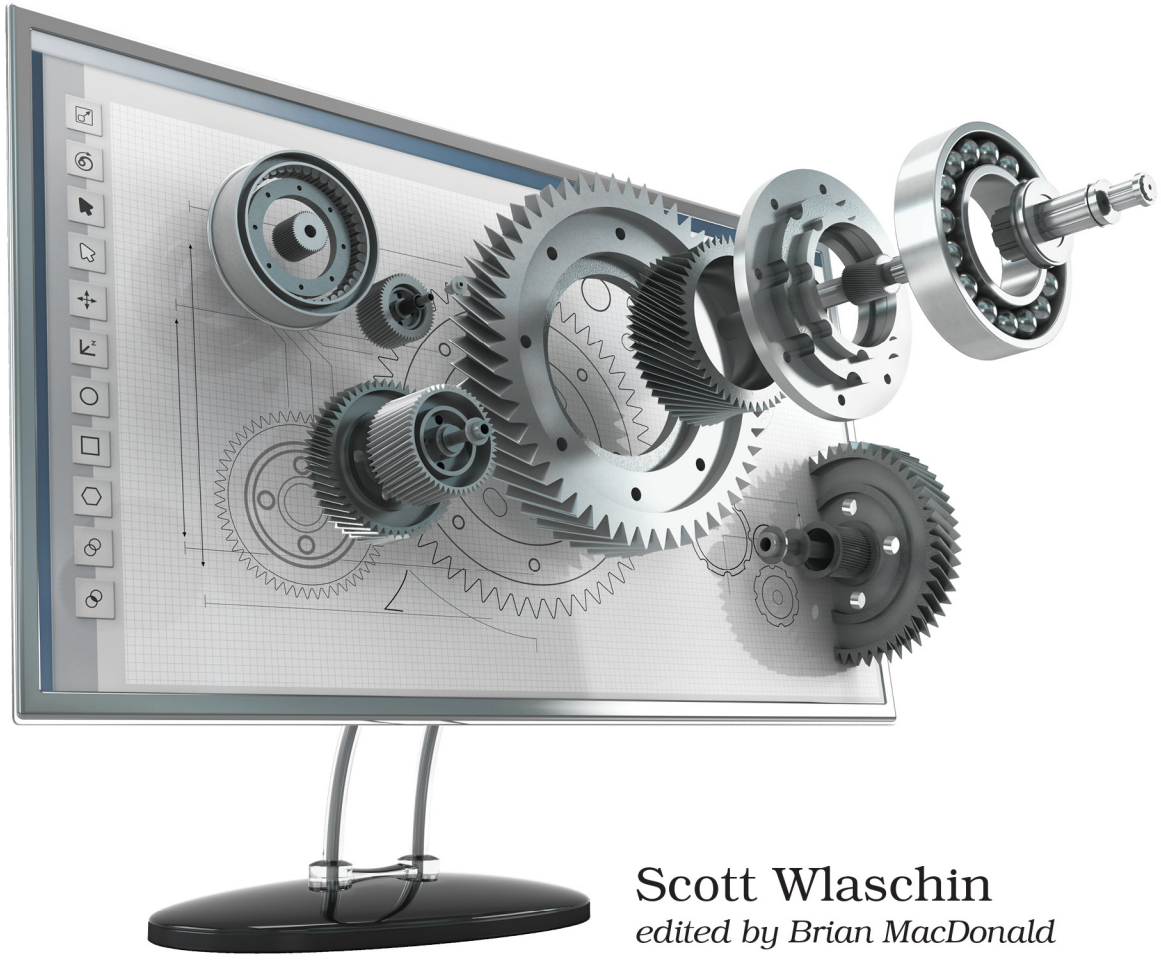
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#



Scott Wlaschin
edited by Brian MacDonald

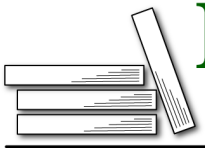
Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

Scott Wlaschin

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Indexing: Potomac Indexing, LLC
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

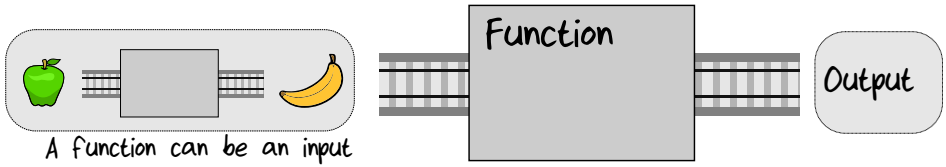
Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

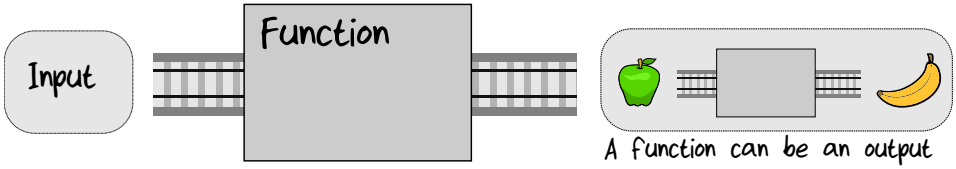
Printed in the United States of America.
ISBN-13: 978-1-68050-254-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Functions Are Things

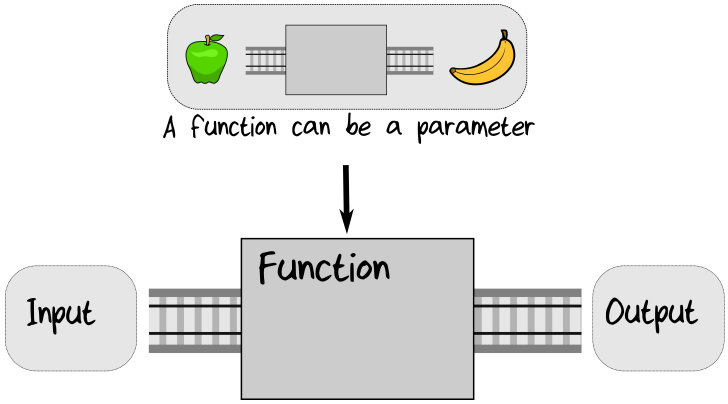
In the functional programming paradigm, functions are things in their own right. And if functions are things, then they can be passed as input to other functions:



Or they can be returned as the output of a function:



Or they can be passed as a parameter to a function to control its behavior:



Treating functions as things opens up a world of possibilities. It's hard to get your head around at first, but you can already see that even with this basic principle you can build up complex systems quite quickly.

Jargon Alert: "Higher-Order Functions"

Functions that input or output other functions or take functions as parameters are called *higher-order functions*, often abbreviated to *HOFs*.

Treating Functions as Things in F#

Let's take a look at how "functions as things" works in F#. Here are four function definitions:

```
let plus3 x = x + 3            // plus3 : x:int -> int
let times2 x = x * 2           // times2 : x:int -> int
let square = (fun x -> x * x)  // square : x:int -> int
let addThree = plus3           // addThree : (int -> int)
```

The first two definitions are just like the ones we've seen before. In the third definition, the let keyword is used to assign a name (square) to an anonymous function, also known as a *lambda* expression. In the fourth definition, the let keyword is used to assign a name (addThree) to a function defined earlier (plus3). Each of these functions is an int -> int function that takes an int as input and outputs a new int.

Now, because functions are things, we can put them in a list:

```
// listOfFunctions : (int -> int) list
let listOfFunctions =
  [addThree; times2; square]
```

In F#, list literals use square brackets as delimiters, with semicolons (not commas!) as element separators.

We can now loop through the list and evaluate each function in turn:

```
for fn in listOfFunctions do
  let result = fn 100   // call the function
  printfn "If 100 is the input, the output is %i" result

// Result =>
// If 100 is the input, the output is 103
// If 100 is the input, the output is 200
// If 100 is the input, the output is 10000
```

The let keyword is not just for function definitions—it's used generally to assign names to values. So for example, here is let used to assign a name to the string "hello":

```
// myString : string
let myString = "hello"
```

The fact that the same keyword (let) is used to define both functions and simple values is not an accident. Let's look at an example to see why. In this first snippet, I define a function called square:

```
// square : x:int -> int
let square x = x * x
```

And in this second snippet I'm assigning the name square to an anonymous function. Is let defining a simple value here or a function?

```
// square : x:int -> int
let square = (fun x -> x * x)
```

The answer is both! A function is a thing and can be assigned a name. So the second definition of square is essentially the same as the first, and they can be used interchangeably.

Functions as Input

We said that "functions as things" means that they can be used for input and output, so let's see what that looks like in practice.

First, let's look at using a function as an input parameter. Here's a function called evalWith5ThenAdd2, which takes a function fn, calls it with 5, and then adds 2 to the result.

```
let evalWith5ThenAdd2 fn =
  fn(5) + 2

// evalWith5ThenAdd2 : fn:(int -> int) -> int
```

If we look at the type signature at the bottom, we can see that the compiler has inferred that fn must be an (int -> int) function.

Let's test it now. First, we'll define add1, which is an (int -> int) function, and then pass it in.

```
let add1 x = x + 1     // an int -> int function
evalWith5ThenAdd2 add1 // fn(5) + 2 becomes add1(5) + 2
//                     // so output is 8
```

The result is 8, as we would expect.

We can use *any* (int -> int) function as a parameter. So let's define a different one, such as square and pass it as a parameter:

```
let square x = x * x     // an int -> int function
evalWith5ThenAdd2 square // fn(5) + 2 becomes square(5) + 2
//                       // so output is 27
```

And this time the result is 27.

Functions as Output

Now let's turn to functions as output. Why would you want to do that?

Well, one very important reason to return functions is that you can "bake in" certain parameters to the function.

For example, say you have three different functions to add integers, like this:

```
let add1 x = x + 1
let add2 x = x + 2
let add3 x = x + 3
```

Obviously, we would like to get rid of the duplication. How can we do that?

The answer is to create an "adder generator"—a function that returns an "add" function with the number to add baked in:

Here's what the code would look like:

```
let adderGenerator numberToAdd =
  // return a lambda
  fun x -> numberToAdd + x

// val adderGenerator :
//    int -> (int -> int)
```

Looking at the type signature, it clearly shows us that it takes an int as input and emits an (int -> int) function as output.

We could also implement adderGenerator by returning a named function instead of an anonymous function, like this:

```
let adderGenerator numberToAdd =
  // define a nested inner function
  let innerFn x =
    numberToAdd + x

  // return the inner function
  innerFn
```

As we've seen with the square example earlier, both implementations are effectively the same. Which one do you prefer?

Finally, here's how adderGenerator might be used in practice:

```
// test
let add1 = adderGenerator 1
add1 2     // result => 3

let add100 = adderGenerator 100
add100 2   // result => 102
```

Currying

Using this trick of returning functions, *any* multiparameter function can be converted into a series of one-parameter functions. This method is called *currying*.

For example, a two-parameter function such as `add`:

```
// int -> int -> int
let add x y = x + y
```

can be converted into a one-parameter function by returning a new function, as we saw above:

```
// int -> (int -> int)
let adderGenerator x = fun y -> x + y
```

In F#, we don't need to do this explicitly—every function is a curried function! That is, any two-parameter function with signature `'a -> 'b -> 'c` can also be interpreted as a one-parameter function that takes an `'a` and returns a function (`'b -> 'c`), and similarly for functions with more parameters.

Partial Application

If every function is curried, that means you can take any multiparameter function and pass in just one argument, and you'll get a new function back with that parameter baked in but all the other parameters still needed.

For example, the `sayGreeting` function below has two parameters:

```
// sayGreeting: string -> string -> unit
let sayGreeting greeting name =
  printfn "%s %s" greeting name
```

But we can pass in just one parameter to create some new functions with the greeting baked in:

```
// sayHello: string -> unit
let sayHello = sayGreeting "Hello"

// sayGoodbye: string -> unit
let sayGoodbye = sayGreeting "Goodbye"
```

These functions now have one remaining parameter, the name. If we supply that, we get the final output:

```
sayHello "Alex"
// output: "Hello Alex"

sayGoodbye "Alex"
// output: "Goodbye Alex"
```

This approach of "baking in" parameters is called *partial application* and is a very important functional pattern. For example, we'll see it being used to do dependency injection in *Implementation: Composing a Pipeline* when we start implementing the order-taking workflow.