Extracted from:

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

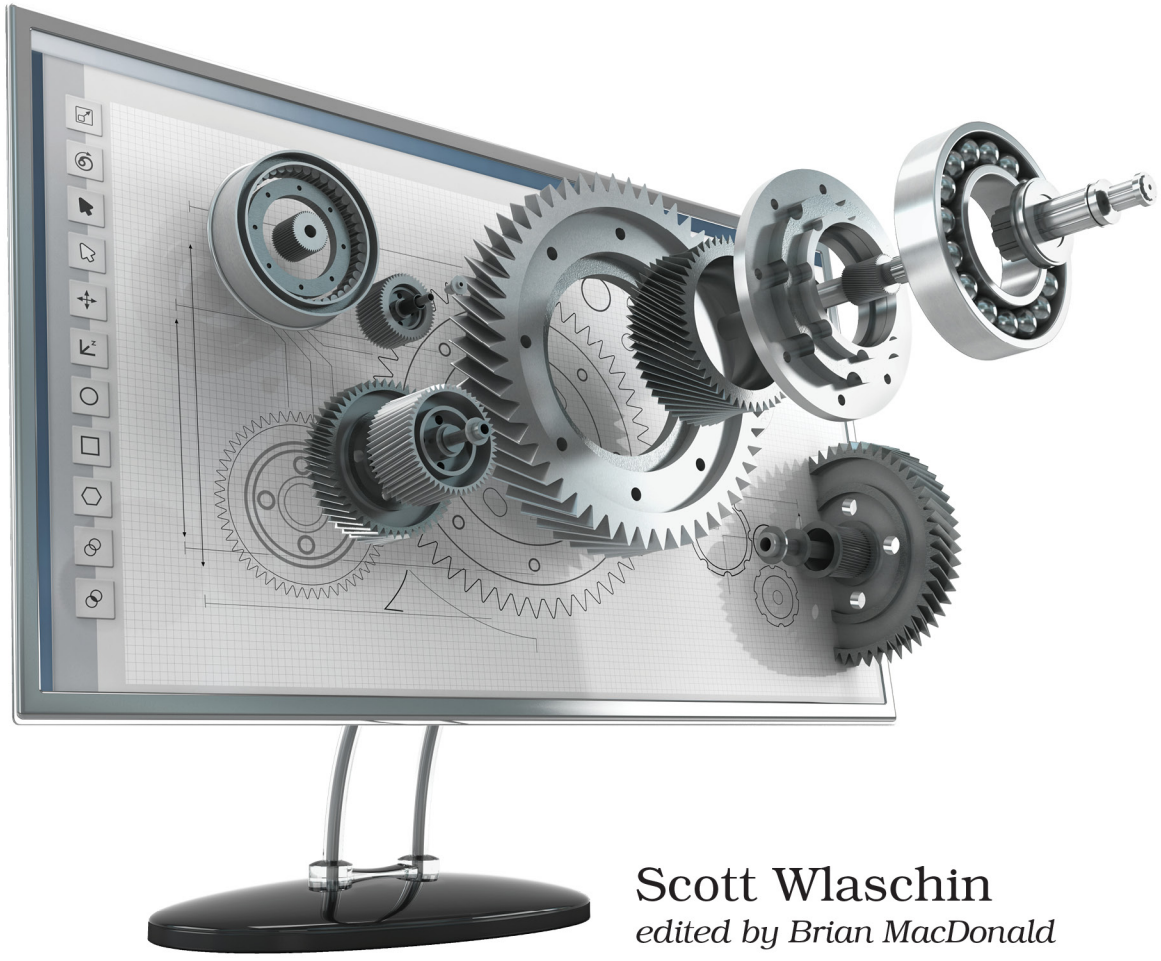
The Pragmatic Bookshelf

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

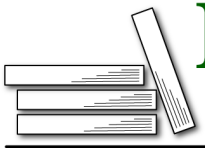
Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

Scott Wlaschin

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

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*.

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

Modeling Complex Data

When we documented our domain on page ?, we used *AND* and *OR* to represent more complex models. In *Understanding Types*, we learned about F#'s algebraic type system and saw that it also used *AND* and *OR* to create complex types from simple ones.

Let's now take the obvious step and use the algebraic type system to model our domain.

Modeling with Record Types

In our domain, we saw that many data structures were built from *AND* relationships. For example, our original, simple Order was defined like this:

```
data Order =
    CustomerInfo
    AND ShippingAddress
    AND BillingAddress
    AND list of OrderLines
    AND AmountToBill
```

This translates directly to an F# record structure, like this:

```
type Order = {
  CustomerInfo : CustomerInfo
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : ...
  }
```

We have given each field a name ("CustomerInfo," "ShippingAddress") and a type (CustomerInfo, ShippingAddress).

Doing this shows a lot of still-unanswered questions about the domain—we don't know what these types actually are right now. Is ShippingAddress the same type as BillingAddress? What type should we use to represent "AmountToBill"?

Ideally, we can ask our domain experts to help with this. For example, if your experts talk about billing addresses and shipping addresses as different things, it's better to keep these logically separate, even if they have the same structure. They may evolve in different directions as your domain understanding improves or as requirements change.

Modeling Unknown Types

During the early stages of the design process, you often won't have definitive answers to some modeling questions. For example, you'll know the names of types that you need to model, thanks to the ubiquitous language, but not their internal structure.

This isn't a problem—you can represent types of unknown structure with best guesses, or alternatively you can model them as a type that's *explicitly* undefined, one that acts as a placeholder, until you have a better understanding later in the design process.

If you want to represent an undefined type in F#, you can use the exception type `exn` and alias it to `Undefined`:

```
type Undefined = exn
```

You can then use the `Undefined` alias in your design model, like this:

```
type CustomerInfo = Undefined
type ShippingAddress = Undefined
type BillingAddress = Undefined
type OrderLine = Undefined
type BillingAmount = Undefined

type Order = {
  CustomerInfo : CustomerInfo
  ShippingAddress : ShippingAddress
  BillingAddress : BillingAddress
  OrderLines : OrderLine list
  AmountToBill : BillingAmount
  }
```

This approach means that you can keep modeling the domain with types and compile the code. But when you try to write the functions that process the types, you will be forced to replace `Undefined` with something a bit better.

Modeling with Choice Types

In our domain, we also saw many things that were choices between other things, such as these:

```
data ProductCode =
    WidgetCode
    OR GizmoCode

data OrderQuantity =
    UnitQuantity
    OR KilogramQuantity
```

How can we represent these choices with the F# type system? With choice types, obviously!

```
type ProductCode =
  | Widget of WidgetCode
  | Gizmo of GizmoCode

type OrderQuantity =
  | Unit of UnitQuantity
  | Kilogram of KilogramQuantity
```

Again, for each case we need to create two parts: the "tag" or case label (before the "of") and the type of the data that is associated with that case. The example above shows that the case label (such as Widget) doesn't have to be the same as the name of the type (WidgetCode) associated with it.

Modeling Workflows with Functions

We've now got a way to model all the data structures—the "nouns" of the ubiquitous language. But what about the "verbs," the business processes? In this book, we will model workflows and other processes as function types. For example, if we have a workflow step that validates an order form, we might document it like this:

```
type ValidateOrder = UnvalidatedOrder-> ValidatedOrder
```

It's clear from this code that the ValidateOrder process transforms an unvalidated order into a validated one.

Working with Complex Inputs and Outputs

Every function has only one input and one output, but some workflows might have multiple inputs and outputs. How can we model that? We'll start with the outputs. If a workflow has an outputA *and* an outputB, then we can create a record type to store them both. We saw this with the order-placing workflow: the output needs to be three different events, so let's create a compound type to store them as one record:

```
type PlaceOrderEvents = {
  AcknowledgmentSent : AcknowledgmentSent
  OrderPlaced : OrderPlaced
  BillableOrderPlaced : BillableOrderPlaced
  }
```

Using this approach, the order-placing workflow can be written as a function type, starting with the raw UnvalidatedOrder as input and returning the PlaceOrderEvents record:

```
type PlaceOrder = UnvalidatedOrder -> PlaceOrderEvents
```

On the other hand, if a workflow has an outputA *or* an outputB, then we can create a choice type to store them both. For example, we briefly talked about categorizing the inbound mail as quotes or orders on page ?. That process had at least two different choices for outputs:

```
workflow "Categorize Inbound Mail" =
    input: Envelope contents
    output:
        QuoteForm (put on appropriate pile)
        OR OrderForm (put on appropriate pile)
        OR ...
```

It's easy to model this workflow: just create a new type, say CategorizedMail, to represent the choices, and then have CategorizeInboundMail return that type. Our model might then look like this:

```
type EnvelopeContents = EnvelopeContents of string
type CategorizedMail =
  | Quote of QuoteForm
  | Order of OrderForm
  // etc

type CategorizeInboundMail = EnvelopeContents -> CategorizedMail
```

Now let's look at modeling inputs. If a workflow has a choice of different inputs (*OR*), then we can create a choice type. But if a process has multiple inputs that are all required (*AND*), such as "Calculate Prices" (below), we can choose between two possible approaches.

```
"Calculate Prices" =
    input: OrderForm, ProductCatalog
    output: PricedOrder
```

The first and simplest approach is just to pass each input as a separate parameter, like this:

```
type CalculatePrices = OrderForm -> ProductCatalog -> PricedOrder
```

Alternatively, we could create a new record type to contain them both, such as this CalculatePricesInput type:

```
type CalculatePricesInput = {
  OrderForm : OrderForm
  ProductCatalog : ProductCatalog
  }
```

And now the function looks like this:

```
type CalculatePrices = CalculatePricesInput -> PricedOrder
```

Which approach is better? In the cases above, where the ProductCatalog is a dependency rather than a "real" input, we want to use the separate parameter approach. This lets us use the functional equivalent of dependency injection. We'll discuss this in detail in *Injecting Dependencies*, on page ?, when we implement the order-processing pipeline.

On the other hand, if both inputs are always required and are strongly connected with each other, then a record type will make that clear. (In some situations, you can use tuples as an alternative to simple record types, but it's generally better to use a named type.)

Documenting Effects in the Function Signature

We just saw that the ValidateOrder process could be written like this:

```
type ValidateOrder = UnvalidatedOrder -> ValidatedOrder
```

But that assumes that the validation always works and a ValidatedOrder is always returned. In practice, of course, this would not be true, so it would better to indicate this situation by returning a Result type (introduced on page ?) in the function signature:

```
type ValidateOrder =
  UnvalidatedOrder -> Result<ValidatedOrder,ValidationError list>

and ValidationError = {
  FieldName : string
  ErrorDescription : string
  }
```

This signature shows us that the input is an UnvalidatedOrder and, if successful, the output is a ValidatedOrder. But if validation failed, the result is a list of ValidationError, which in turn contains a description of the error and which field it applies to.

Functional programming people use the term *effects* to describe things that a function does in addition to its primary output. By using Result here, we've now documented that ValidateOrder might have "error effects." This makes it clear in the type signature that we can't assume the function will always succeed and that we should be prepared to handle errors.

Similarly, we might want to document that a process is asynchronous—it will not return immediately. How can we do that? With another type of course!

In F#, we use the Async type to show that a function will have "asynchronous effects." So if ValidateOrder had async effects as well as error effects, then we would write the function type like this:

```
type ValidateOrder =
  UnvalidatedOrder -> Async<Result<ValidatedOrder,ValidationError list>>
```

This type signature now documents (a) when we attempt to fetch the contents of the return value, the code won't return immediately and (b) when it does return, the result might be an error.

Listing all the effects explicitly like this is useful, but it does make the type signature ugly and complicated, so we would typically create a type alias for this to make it look nicer.

```
type ValidationResponse<'a> = Async<Result<'a,ValidationError list>>
```

Then the function could be documented like this:

```
type ValidateOrder =
  UnvalidatedOrder -> ValidationResponse<ValidatedOrder>
```