Extracted from:

Domain Modeling Made Functional

Tackle Software Complexity with Domain-Driven Design and F#

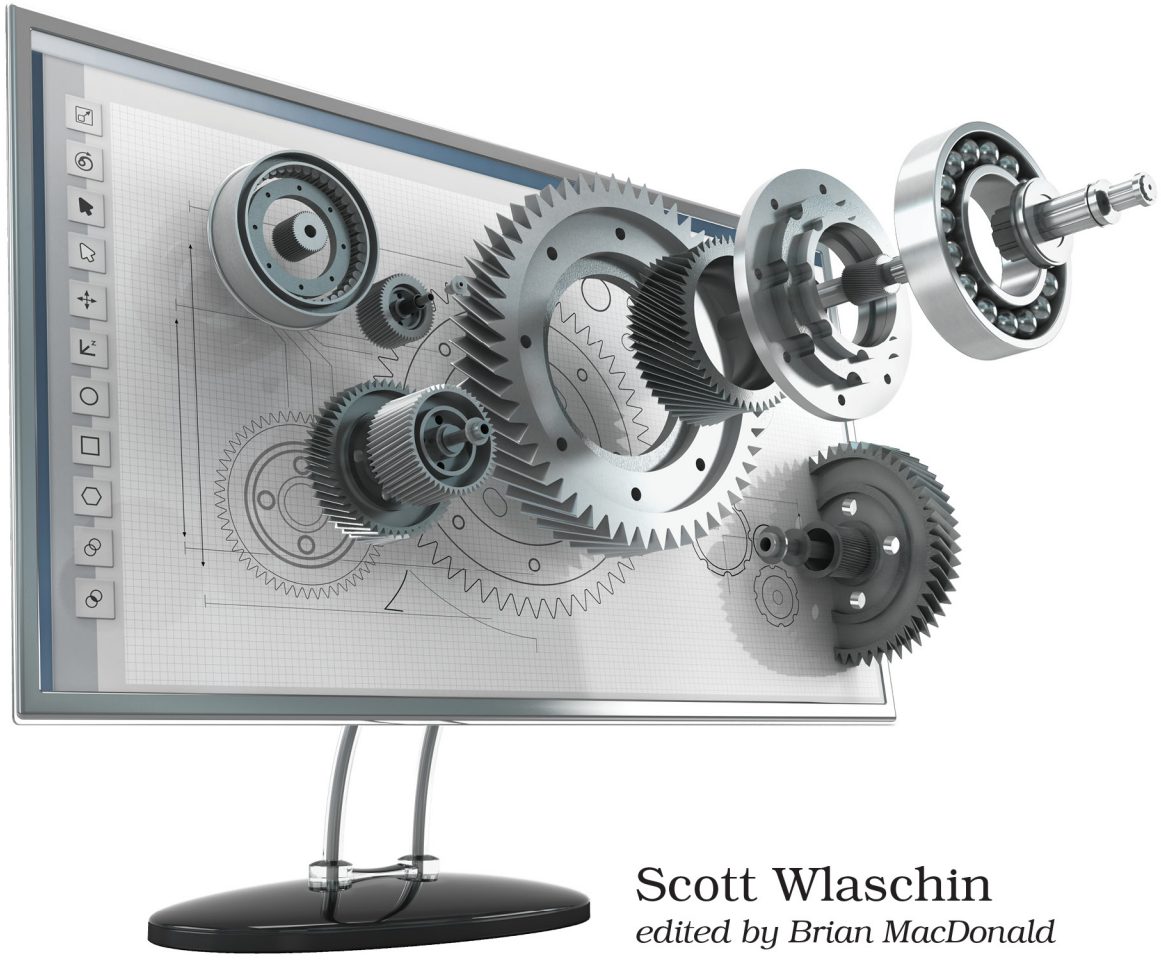
The Pragmatic Bookshelf

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

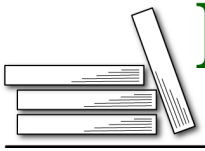
Domain Modeling Made Functional

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*.

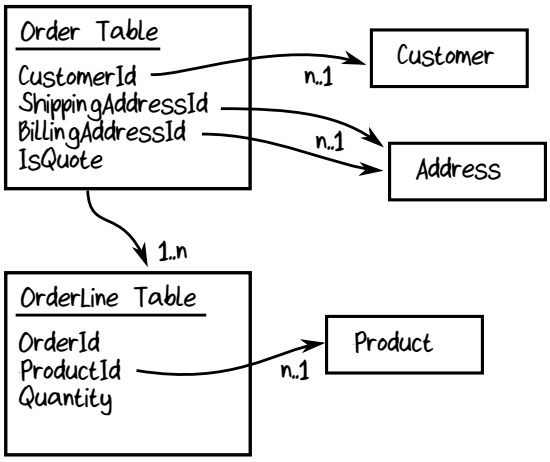
Printed in the United States of America.
ISBN-13: 978-1-68050-254-1
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

Fighting the Impulse to Do Database-Driven Design

At this point, if you are like most developers, you can't help but start sketching out a low-level design and implementation immediately.

For example, you might look at that order form and see that it consists of customer information, some addresses, a list of order lines, and so on.

If you have a lot of database experience, your first instinct might be to think about tables and the relationships between them. You might envision an Order table, an OrderLine table, and Customer, Address, and Product tables. And then you'll probably want to describe the relationships between them as shown in the figure.



But if you do this, you are making a mistake. In domain-driven design we let the *domain* drive the design, not a database schema.

It's better to work from the domain and to model it without respect to any particular storage implementation. After all, in a real-world, paper-based system, there is no database. The concept of a "database" is certainly not part of the ubiquitous language. The users do not care about how data is persisted.

In DDD terminology this is called *persistence ignorance*. It is an important principle because it forces you to focus on modeling the domain accurately, without worrying about the representation of the data in a database.

Why is this important? Well, if you design from the database point of view all the time, you often end up distorting the design to fit a database model.

As an example of the distortion that a database-driven model brings, we have already ignored the difference between an order and a quote in the diagram above. Sure, in the database we can have a flag to distinguish them, but the business rules and validation rules are different. For example, we might later learn that an Order must have a billing address but a Quote doesn't. This is hard to model with a foreign key. This subtlety has been lost in database design because the same foreign key does dual duty for both types of relationships.

Of course, the design can be corrected to deal with it, and in the chapter on persistence on page ?, we'll see how to persist a domain-driven design into a relational database. But for now we really want to concentrate on listening to the requirements without prejudice.

Fighting the Impulse to Do Class-Driven Design

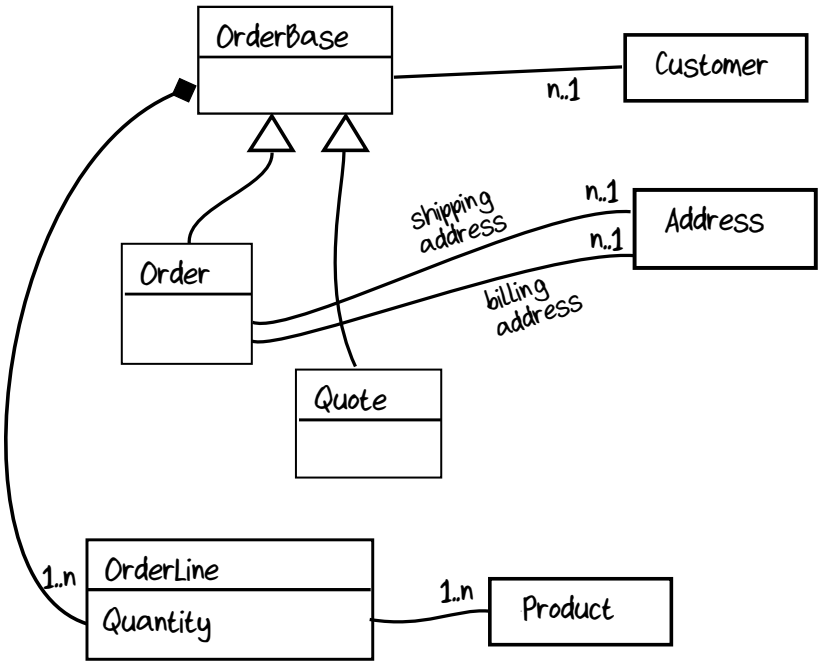
If you're an experienced object-oriented developer, then the idea of not being biased to a particular database model will be familiar, Indeed, object-oriented techniques such as dependency injection encourage you to separate the database implementation from the business logic.

But you, too, may end up introducing bias into the design if you think in terms of objects rather than the domain.

For example, as Ollie is talking, you may be creating classes in your head, like the figure on page 7.

Letting classes drive the design can be just as dangerous as letting a database drive the design—again, you're not really listening to the requirements.

In the preliminary design above we have separated orders and quotes, but we have introduced an artificial base class, `OrderBase`, that doesn't exist in the real world. This is a distortion of the domain. Try asking the domain expert what an `OrderBase` is!



The lesson here is that we should keep our minds open during requirements gathering and not impose our own technical ideas on the domain.

Documenting the Domain

OK, we want to avoid biasing ourselves with technical implementations, but then how *should* we record these requirements?

We could use visual diagrams (such as UML), but these are often hard to work with and not detailed enough to capture some of the subtleties of the domain.

Later in this book we'll see how to create an accurate domain model in code, but for now, let's just create a simple text-based language that we can use to capture the domain model:

- For workflows, we'll document the inputs and outputs and then just use some simple pseudocode for the business logic.
- For data structures, we'll use `AND` to mean that both parts are required, such as in `Name AND Address`. And we'll use `OR` to mean that either part is required, such as in `Email OR PhoneNumber`.

Using this mini-language, then, we can document the Place Order workflow like this:

```
Bounded context: Order-Taking

Workflow: "Place order"
   triggered by:
      "Order form received" event (when Quote is not checked)
   primary input:
      An order form
   other input:
      Product catalog
   output events:
      "Order Placed" event
   side-effects:
      An acknowledgment is sent to the customer,
      along with the placed order
```

And we can document the data structures associated with the workflow like this:

```
bounded context: Order-Taking

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

data OrderLine =
    Product
    AND Quantity
    AND Price

data CustomerInfo = ???   // don't know yet
data BillingAddress = ??? // don't know yet
```

The Provide Quote workflow and its associated data structures can be documented in a similar way.

Note that we have not attempted to create a class hierarchy or database tables or anything else. We have just tried to capture the domain in a slightly structured way.

The advantage of this kind of text-based design is that it's not scary to non-programmers, which means it can be shown to the domain expert and worked on together.

The big question is whether can we make our code look as simple as this, too. In a following chapter, *Domain Modeling with Types*, we'll try to do just that.