Outline for
"Domain Driven Design: a Functional Approach using F#"
by Scott Wlaschin

Expected number of pages: 250 (8 chapters x 30 pages per chapter)

Chapter 1: Why is domain-driven design important?

A developer's job is to *solve a problem through software*, and coding is just one aspect of software development. Good design and communication are just as important, if not more so.

If you think of software development as a pipeline with an input (requirements) and an output (the final deliverable), then the "garbage in, garbage out" rule applies. No amount of coding can fix unclear requirements or a bad design.

In this book I'll show you how to minimize the "garbage-in" by using a design approach focused on clear communication and shared domain knowledge: Domain-Driven Design.

This book will use a functional programming language (F#) for domain modeling. If you think that functional programming is all about abstract algorithms and impractical code, I hope to change your mind and show you that F# is in fact an excellent language for domain modeling, and can produce code that is not only more concise and more testable, but also easier to understand.

In this first chapter, we'll start by looking at a real-world example of an order-taking system.

We'll dive into various scenarios of how it can be used, and the various subsystems involved. As we do this, we'll develop the core principles of Domain-Driven Design, such as "domain", "ubiquitous language", "bounded context", and so on.

Chapter 2: The basics of F#

In this chapter, I'll introduce you to the basics of F# and functional programming.

This book cannot possibly explain everything about functional programming, and I will not attempt to. I'll cover only what you need for this book:

- "Functions as things", and why this leads to immutability and other common features of FP.
- Composition as the overarching design principle.
- Functional type systems (aka "Algebraic types"), and how they are completely different from OO-style classes.

- And finally, the F# syntax that you need to know to read the example code in this book. This book will be staying at a high-level, and so the syntax needed will be minimal.

I will not be discussing scary concepts such as monads, functors, and so on – they are important but you do not need to understand them to read this book.

Chapter 3: Domain modeling with types

A key principle of agile development is to make the code the primary source of documentation. But reading source code can be confusing for a non-developer, and it is hard to separate the design from the code.

Is it possible to use *types* as a design tool and avoid the need for UML diagrams and the like?

The answer is yes, and in this chapter we'll see how you can use a functional type system to capture the domain model accurately, concisely, and in a way that can be read by customers and non-developers.

We'll start with a challenge for you. Here's a simple design for a customer record -- how many things are wrong with it, and how would you improve it?

(code example)

We'll work through this example and on the way discover the various kinds of functional types and how they can be used to improve this code.

Finally, we'll see how you can "make illegal states unrepresentable" – code which captures the requirements in such as way that you literally cannot write bad code! Not only does this improve the design, but it means that you need to write many fewer unit tests.

Chapter 4: Building a complete use-case

In this chapter, we'll take one use-case from our order-taking example and design it using functional principles.

A business process is often thought of as a series of document transformations, and we'll see we can model the use-case in the same way: as a "pipeline" built from a series of smaller "pipes" using composition, using the concept of "transformation-oriented programming."

We'll see that each step in the pipeline will be designed to be stateless, which means that they can be tested independently.

As part of this implementation we'll see how to do error handling in a functional way (no exceptions!) and how common error cases can be included in the domain model.

We'll also look at how to process pipelines in parallel rather than in series, and finally we'll look at how to integrate promises/async into the pipeline.

Chapter 5: Combining use-cases into a functional architecture

In the previous chapter we looked at one single use-case. In this chapter we'll step back to look at the big picture. How do we connect these pipelines together? How should they be grouped and organized?

First, we will look at the inputs and outputs of our pipelines. Where does the data come from, and how can we make sure that we trust it?

Next, we will see that this "pipeline" approach naturally leads to a "ports and adapters" architecture and then the DDD concepts of a "bounded context" and "anti-corruption layers".

Finally, we'll go back to the requirements gathering process, and look at "event storming" as a way to capture the important inputs to the system using "commands" and "domain events".

Chapter 6: Evolving a design and keeping it clean

It is all too common for a domain model to start off clean and elegant, but as the requirements change, the model gets messy and the various subsystems become entangled and hard to test.

In this chapter, we'll take a look at how to ensure that the various components in a domain stay cohesive and decoupled, even as the design and requirements change, using the previous use-cases as real-world examples.

We'll also look at how you can design security into the domain, so that code is only callable to those authorized to use it. It turns out that decoupling and authorization are related and can use the same techniques, because the security principle of POLA ("Principle of Least Authority") is directly analogous to the familiar "Interface Segregation Principle".

Finally, we'll apply both of these principles to API design, creating an API that is resistant to design changes and that cannot be misused because illegal calls are impossible.

Chapter 7: Persisting the domain model

So far in this book, all the design discussions have ignored how to store the data in the database. In this chapter, we'll take the domain model from the previous chapters and look at various ways to integrate a persistence mechanism into it.

We'll start with the Command-Query Separation principle, also known as "asking a question shouldn't change the answer", and see how this principle directly follows from working with immutability.

We'll also introduce the idea of "event sourcing" as a great fit for stateless services and develop a simple implementation for the order-taking system.

Chapter 8: Putting it all together, exposing an API

In the final chapter, we'll integrate all the techniques developed so far, and use them to build a simple RESTFul web service using HATEAOS.

Appendix: Glossary

Confused by DDD terms like "Anti-Corruption Layer", or functional programming terms like "Monoid"? The glossary contains concise definitions of confusing words and phrases, with links to deeper explanations in the main content of the book.