Chapter 1. Domain Modelling

In this chapter we'll look into how we can model business processes with code, in a way that's highly compatible with TDD. We'll discuss *why* domain modelling matters, and we'll look at a few key patterns for modelling domains: Entities, Value Objects, and Domain Services.

What is a Domain Model?

In the ???, we used the term *business logic layer* to describe the central layer of a three-layered architecture. For the rest of the book, we're going to use the term *Domain Model* instead. This is a term from the DDD community that does a better job of capturing our intended meaning (see the next sidebar for more on DDD).

The *domain* is a fancy way of saying *the problem you're trying to solve*. We currently work for an online retailer of furniture. Depending on which system I'm talking about, the domain might be purchasing and procurement, or product design, or logistics and delivery. Most programmers spend their days trying to improve or automate business processes; the domain is the set of activities that those processes support.

A model is a map of a process or phenomenon that captures some useful property. Humans are exceptionally good at producing models of things in their heads. For example, when someone throws a ball toward you, you're able to predict its movement almost unconsciously, because you have a model of how objects move in space. Your model isn't perfect by any means. Humans have terrible intuitions about how objects behave at near-light speeds or in a vacuum because our model was never designed to cover those cases. That doesn't mean the model is wrong, but it does mean that some predictions fall outside of its domain.

THIS IS NOT A DDD BOOK. YOU SHOULD READ A DDD BOOK.

Domain-driven design, or DDD, is where the concept of domain modelling was popularized,[1] and it's been a hugely successful movement in transforming the way

people design software by focusing on the core business domain. Many of the architecture patterns that we cover in this book, like Entity, Aggregate and Value Objects (see Chapter 5), and Repository pattern (in the next chapter) all come from the DDD tradition.

In a nutshell, DDD says that the most important thing about software is that it provides a useful model of some problem. If we get that model right, then our software delivers value and makes new things possible.

If we get it wrong, it becomes an obstacle to be worked around. In this book we can show the basics of building a domain model, and building an architecture around it that leaves the model as free as possible from external constraints, so that it's easy to evolve and change.

But there's a lot more to DDD, and the processes, tools and techniques for developing a domain model. We hope to give you a taste for it though, and cannot encourage you enough to go on and read a proper DDD book.

- The original "blue book", *Domain-Driven Design* by Eric Evans (Addison-Wesley, 2003)
- Or, some people prefer the "red book", *Implementing Domain-Driven Design*, by Vaughn Vernon (Addison-Wesley, 2013).

The Domain Model is the mental map that business owners have of their businesses. All business people have these mental maps, they're how humans think about complex processes.

You can tell when they're navigating these maps because they use business speak. Jargon arises naturally between people who are collaborating on complex systems.

Imagine that you, our unfortunate reader, were suddenly transported light years away from Earth aboard an alien spaceship with your friends and family and had to figure out, from first principles, how to navigate home.

In your first few days, you might just push buttons randomly, but soon you'd learn which buttons did what, so that you could give one another instructions. "Press the red button near the flashing doo-hickey and then throw that big lever over by the radar gizmo," you might say.

Within a couple of weeks, you'd become more precise as you adopted words to describe the ship's functions: "increase oxygen levels in cargo

bay three" or "turn on the little thrusters." After a few months you'd have adopted language for entire complex processes: "Start landing sequence," or "prepare for warp." This process would happen quite naturally, without any formal effort to build a shared glossary.

So it is in the mundane world of business. The terminology used by business stakeholders represents a distilled understanding of the domain model, where complex ideas and processes are boiled down to a single word or phrase.

When we hear our business stakeholders using unfamiliar words, or using terms in a specific way, we should listen to understand the deeper meaning and encode their hard-won experience into our software.

We're going to use a real-world domain model throughout this book, specifically a model from our current employment. Made.com is a successful furniture retailer. We source our furniture from manufacturers all over the world and sell it across Europe.

When you buy a sofa or a coffee table, we have to figure out how best to get your goods from Poland or China or Vietnam, and into your living room.

At a high level, we have separate systems that are responsible for buying stock, selling stock to customers, and shipping goods to customers. There's a system in the middle that needs to coordinate the process by allocating stock to a customer's orders; see Figure 1-1.

Figure 1-1. Context diagram for the allocation service

For the purposes of this book, we're imagining a situation where the business decides to implement an exciting new way of allocating stock. Until now, the business has been presenting stock and lead times based on what is physically available in the warehouse. If and when the warehouse runs out, a product is listed as "out of stock" until the next shipment arrives from the manufacturer.

The innovation is: if we have a system that can keep track of all our shipments and when they're due to arrive, then we can treat the goods on those ships as real stock, and part of our inventory, just with slightly

longer lead times. Fewer goods will appear to be out of stock, we'll sell more, and the business can save money by keeping lower inventory in the domestic warehouse.

But allocating orders is no longer a trivial matter of decrementing a single quantity in the warehouse system. We need a more complex allocation mechanism. Time for some domain modelling.