Discovering

Domain-Driven Design

[Sample chapters]

What is domain-driven design?

"A domain-driven design approach provides principles and patterns to address the challenges faced with developing complex domain models"
- Eric Evans

Let's start with a common problem - developers start coding a software solution with a lack of business knowledge and understanding. When this happens we can end up with developers that write code that does not truly represent business functionality. As software engineers, most of us have experienced a "spaghetti" or "big ball of mud" codebase. A codebase with no clarity, where there are dependencies in all directions. What this really means is the code is often highly coupled, fragile and is difficult to test and maintain. A change to one part of the code may have a big knock-on effect.

Domain-driven design approaches software development with a focus on gaining a rich understanding of business rules and processes from domain experts. The aim is to understand as much about the domain as possible to align the engineering team and the software with the business, therefore improving communication, facilitating changes and decreasing complexity of inherently complex problems.

It's important to remember that domain-driven design is not just about code. It is not purely a technical concern, although there are technical patterns we can use in the code. It is a wider concept that requires engineering teams to seek understanding of the problem space before trying to solve the problem, and modelling the domain using that understanding.

If we align the software we build to the business we can write better, more easily maintainable and testable software. We can also communicate more effectively, thus enabling us to solve complex problems.

The main DDD concepts that we will cover throughout this book include domains, domain experts, ubiquitous language, developing domain models

and bounded contexts. We will also take a look at some software patterns that can be implemented to provide a domain-driven angle in your code.

Entities and value objects

Representing the domain model

We've discovered concepts in our domain and built up a mental model. How do we translate that into code? Domain-driven design introduces three key concepts that allow us to represent our domain model; entities, value objects and aggregate roots.

One crucial thing about these types of objects is they are NOT just a data or persistence concern. These objects are all about behaviour and data in regards to the domain model and business logic.

Entities

A slight tangent to preface this section. Throughout my career as a software developer I have used the term "entity" in almost every project. Particularly because of having a .NET background and using Entity Framework, my understanding of an entity was always just a way to represent a database table in code. There's nothing wrong about this, but if that's your current mental model for entities then disregard that in the context of domain-driven design.

So what is an entity? It's a domain object that has some intrinsic identity. This could be a real-world property of the object, a combination of real-world properties, or a computer generated value like an incremental integer or UUID.

The distinguishing behaviour of an entity is that even if its properties are the same as another instance of the same type, it remains distinct because of its unique identity.

Below are some examples of entities.

Person	Product	Car	Song
Social security number Name Age Price Model Colour	**SKU** Name Price	**VIN** Make Model Colour	**UUID** Name Artist Bpm Lyrics

In each of these types you can see there is a unique identifier. A person has a unique social security number (or a national insurance number in the UK) so we could use that, a product often has a SKU (stock keeping unit) and a car has a VIN (vehicle identification number). These are all real-world values that represent an entity's identity. Songs on the other hand may not have a real-world identifier. In cases like this we can assign a computer generated value that is guaranteed to be unique in our system.

If we have two users with an identical name and age they are not necessarily the same person. Only if the social security number is equal can we determine that the two instances represent the same person. Similarly, we could have two instances of a product that have the same exact name and price, but they may be totally different things. If they have the same SKU however, they are considered to represent the same thing.

Tell, don’t ask

Related to coupling vs cohesion, here’s a technique you can use for writing code in a simple but effective domain-driven way.

Here we have a person and some logic to decide whether the person can wear some socks. There is nothing wrong with this at a first glance. The code works and the rules are checked as they should. But let's think about it a little more.

Can a person put socks on?

What if we have two places where we call the putOnSocks function? Can the person put socks on? We would probably need to check using an if statement everywhere we want to call this function in case they can't. Code duplication isn't always a bad thing, but this is certainly something to keep in mind.

Why? Why? Why?

Another consideration is that although we're checking whether the person can put socks on, if they can't we are throwing a PersonCannotWearSocksError error. This is also fine, but it's not very helpful. Why can't they put socks on?

Breaking the rules

Most importantly, what we've discovered here is some domain logic. More specifically, there's an invariant which is a validation rule that *must* be enforced by the domain object for it to be in a valid state.

In this case we have a rule that a person cannot put socks in if they already have socks on. The rule was already there in the original code but it was outside of the person class and would need to be repeated everywhere to ensure the rules were followed. If the rules are not enforced we would publish two consecutive PersonPutOnSocksEvent events. Can you put socks on if you

haven't taken the previous socks off? (Alright smarty pants, maybe you technically could but not in my made-up scenario). And what about removing socks? Can you remove socks if you're not wearing any? Certainly not! So we definitely shouldn't be able to raise consecutive PersonRemovedSocksEvent events, but the code allows us to do just that. We're essentially leaving our code open to logic bugs.

Let's fix it by refactoring the code using the "Tell, don't ask" approach.

Firstly you'll probably notice there is less logic in the calling code. Why's that? Essentially we've encapsulated our domain logic, moving it inside the domain object itself rather than calling it all from outside. This may seem trivial, but there are some major benefits.

Privacy first

Since we're no longer asking whether the person can put on socks we are able to make the canPutOnSocks function private. This isn't required and we could still expose this function if needed, but what it highlights is that we have more fine grained control over what behaviours and functionality we want to expose from our domain objects. Generally speaking, we should aim to expose as little as possible.

Enforcing invariants

By encapsulating the domain logic inside the person class we now have better control over the invariants and can enforce them within the class itself. For example, now we can tell the person to put on socks without asking first. The domain object is now responsible for ensuring that the person is not already wearing socks. Now the logic is encapsulated we can more easily throw a specific error too since we know *why* the person cannot put on socks - because they're already wearing some. The same logic applies to removing socks as mentioned in the previous section.

Since we're now encapsulating this logic we're also ensuring that the correct domain events are published. We can no longer publish consecutive PersonPutOnSocksEvent or PersonRemovedSocksEvent events. We would need to tell the person to remove their socks before putting socks on.

Testing

Before we had logic outside of our domain object. How would we test it? We'd need to test our calling code to ensure the invariants are enforced. What if we miss one? What if there's a mistake or a conflict of rules? Now we're refactored the code, our encapsulated domain logic is easily testable in one place.

This is another silly and trivial example, but hopefully demonstrates the value of how thinking about domain logic and the way we structure our code can help us enforce business rules and result in increased testability and maintainability.