Table of Contents

The Mechanics of Domain-Driven Design (30 Pages)

> When eating an elephant, take one bite at a time.
>
> — Creighton Abrams

As mentioned in the previous chapter, many things can render a project to veer off-course. The intent of DDD is to decompose complex problems on the basis of clear domain boundaries and the communication between them. In this chapter, we look at a set of tenets and techniques to arrive at a collective understanding of the problem at hand in the face of ambiguity, break it down into manageable chunks and translate it into reliably working software.

Understanding the problem

Every decision we take regarding the organization, be it requirements, architecture, code has business and user consequences. In order to conceive, architect, design, build and evolve software

effectively, our decisions need to aid in creating the optimal business impact. This can only be achieved if we have a clear understanding of the problem we intend to solve. This leads us to the realization that there exist two distinct domains when arriving at the solution for a problem:

Problem domain

A term that is used to capture information that simply defines the problem while consciously avoiding any details of the solution. It includes details like **why** we are trying to solve the problem, **what** we are trying to achieve and **how** it needs to be solved. It is important to note that the *why*, *what* and *how* are from the perspective of the customers/stakeholders, not from the perspective of the engineers providing software solutions to the problem.

Consider the example of a retail bank which already provides a checking account capability for their customers. They want access to more liquid funds. To achieve that, they need to encourage customers to maintain higher account balances. To do that, they are looking to introduce a new product called the *premium checking account* with additional features like higher interest rates, overdraft protection, no-charge ATM access, etc. The problem domain expressed in the form of why, what and how is shown here:

Table 1. Problem domain: why, what and how

Question	Answer
Why	Bank needs access to more liquid funds
What	Have customers maintain higher account balances
How	By introducing a new product — the premium checking account with enhanced features

Solution domain

A term used to describe the environment in which the solution is developed. In other words, the process of translating requirements into working software (this includes design, development, testing, deployment, etc). Here the emphasis is on the *how* of the problem being solved. However, it is very difficult to arrive at a solution without having an appreciation of the why and the what.

Building on the previous premium checking account example, the code-level solution for this problem may look something like this:

```
class PremiumCheckingAccountFactory {

    Account openPremiumCheckingAccount(Applicant applicant,
                                       MonetaryAmount initialAmount) {

        Salary salary = checkEmployed(applicant);

        if (salary.isBelowThreshold()) {
            throw new InsufficientIncomeException(applicant);
        }

        Account account = Account.createFor(applicant);
        account.deposit(initialAmount);
        account.activate();
        return account;
    }
}
```

This likely appears like a significant leap from a problem domain description, and indeed it is. Before a solution like this can be arrived at, there may need to exist multiple levels of refinement of the problem. As mentioned in the previous chapter, this may lead to inaccuracies in the understanding of the problem, resulting in a solution that may be good, but not one that solves the problem at hand. Let's look at how we can continuously refine our understanding by closing the gap between the problem and the solution domain.

The ubiquitous language and domain models

As we have discussed in the previous chapter, when a problem is presented to us, we subconsciously attempt to form mental representations of potential solutions. Further, the type and nature of these representations (models) may differ wildly based on factors like our understanding of the problem, our backgrounds and experiences, etc. This implies that it is natural for these models to be different. For example, the same problem can be thought of differently by various team members as shown here:

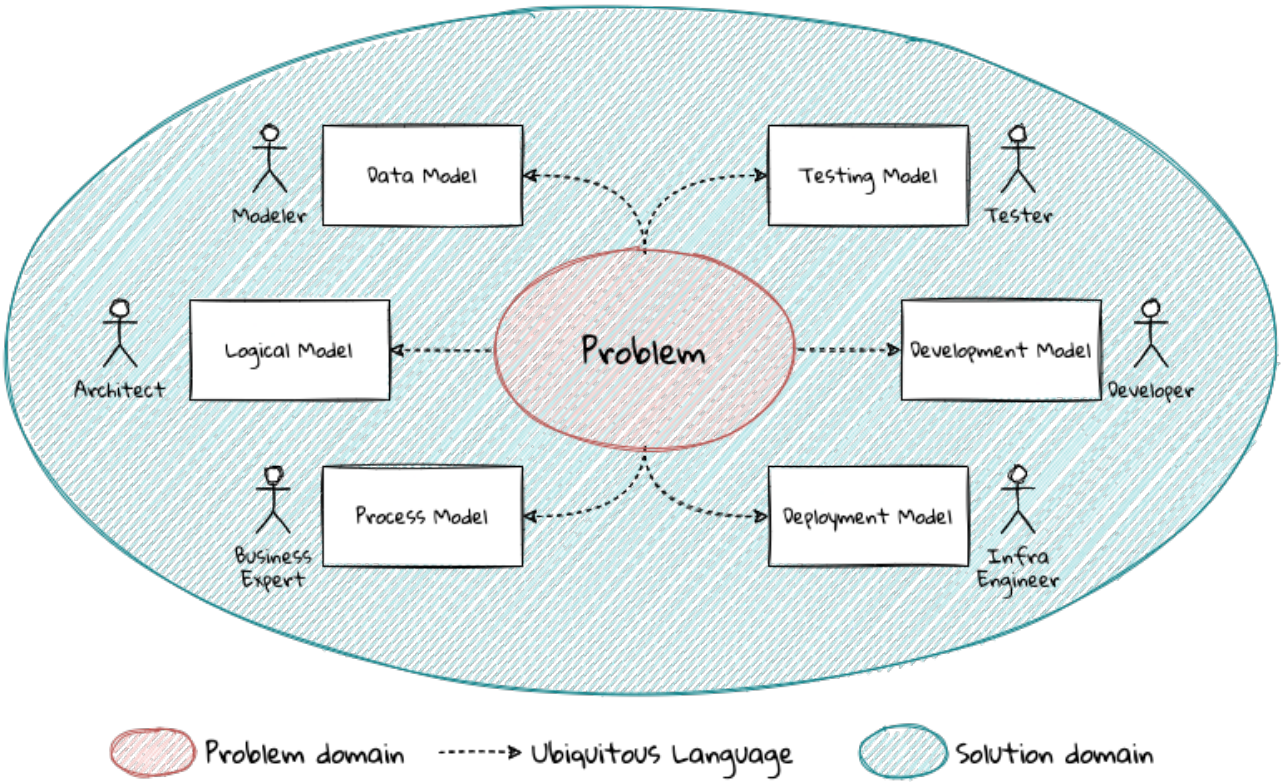

Figure 1. Multiple models to represent the solution to the problem using the ubiquitous language

As iillustrated here, the business expert may think of a process model, whereas the test engineer may think of exceptions and boundary conditions to arrive at a test strategy.

The illustration above is to depict the existence of multiple models. There may be several other perspectives, for example, a customer experience model, an information security model, etc. which are not depicted.

The idea of domain-driven design is for individuals in these varied roles to increase collaboration by promoting the use of the *ubiquitous language* at every step. This enhances the collective understanding, leading to a better quality of the models, and by extension, the quality of the solution. This approach mitigates the risk of the loss in fidelity of information as we transition organizational boundaries.

Modeling tools

Wardley maps

Impact maps

Business model canvas

Product strategy canvas

Domain model

Strategic design

Anemic domain models

Context maps

Bounded contexts

Implementing the solution

Tactical design

Entities

Value objects

Aggregates

Services

Repositories

Factories