Managing Domain Complexity

We learned that the ubiquitous language should reflect the domain experts’ mental models of the business domain’s inner workings and underlying principles.

**Since our goal is to use ubiquitous language to drive software design decisions, the language must be clear and consistent**. It should be free of ambiguity, implicit assumptions, and extraneous details.

However, on an organizational scale, the domain experts’ mental models can be inconsistent themselves. Different domain experts can use different models of the same business domain. Let’s take a look at an example.

*I add: he doesn’t mean that their models are wrong, but the fact that a single term can represent different processes/entities depending on the department for example.*

# Inconsistent Models

Take a telemarketing company as an example. The term “lead” means differently in the sales and marketing departments. The marketing department uses it as simply a notification that somebody’s interested in our product. In the sales department OTOH, it means the entire process of a sale and carries much more details.

How do we formulate a ubiquitous language in the case of this telemarketing company?

* On the one hand, we know the ubiquitous language has to be consistent—each term should have one meaning.
* On the other hand, we know the ubiquitous language has to reflect the domain experts’ mental models.
* In this case, the mental model of the ”lead” is inconsistent among the domain experts in the sales and marketing departments.

This ambiguity doesn’t present that much of a challenge in person-to-person communications.

However, it is more difficult to represent such a divergent model of the business domain in software. **Source code doesn’t cope well with ambiguity**.

* If we were to bring the sales department’s complicated model into marketing, it would introduce complexity where it’s not needed— far more detail and behavior than marketing people need for optimizing advertising campaigns.
* But if we were to try to simplify the sales model according to the marketing world view, it wouldn’t fit the sales subdomain’s needs, because it’s too simplistic for managing and optimizing the sales process. We’d have an overengineered solution in the first case and an under-engineered one in the second.

## The Traditional Solution

The traditional solution to this problem is to design a single model that can be used

for all kinds of problems. Such models result in enormous entity relationship diagrams

(ERDs) spanning whole office walls.

Would this model be effective?

As the saying goes, “jack of all trades, master of none.” Such models are supposed to be suitable for everything but eventually are effective for nothing.

No matter what you do, you are always facing complexity:

* the complexity of filtering out extraneous details
* the complexity of finding what you do need
* and most importantly, the complexity of keeping the data in a consistent state.

## Another Bad solution

Another solution would be to prefix the problematic term with a definition of the context: ”marketing lead” and “sales lead.”

That would allow the implementation of the two models in code. However, this approach has two main disadvantages.

* First, it induces cognitive load. When should each model be used? The closer the implementations of the conflicting models are, the easier it is to make a mistake.

(why this is a problem will probably be more clear as we go further through the book)

* Second, **the implementation of the model won’t be aligned with the ubiquitous language.** No one would use the prefixes in conversations. People don’t need this extra information; they can rely on the conversation’s context.

Let’s turn to the domain-driven design pattern for tackling such scenarios: the bounded

context pattern.

# What is a Bounded Context?