Contents

[Introduction 1](#_Toc153875288)

[Chapter 02 1](#_Toc153875289)

[Aide Memory 2](#_Toc153875290)

# Introduction

We took **Implementing Domain-Driven Design, Vaugh Vernon 2013** book notes.

## Chapter 01

|  |  |  |
| --- | --- | --- |
| Concept | Page |  |
|  | 1 | When implemented correctly, DDD helps us reach the point where our design is exactly how the software works |
|  | 8 | The resulting software generally does not reflect a recognizable realization of the mental model of the domain experts, or perhaps it does so only partially. Over time this disconnect becomes costly. The translation of domain knowledge into software is lost as developers transition to other projects or leave the company |
|  | 20 | At this point some of you may be thinking, “Our designs never really leave the whiteboard. We just draw some structure, and once agreement on that is reached, we are set free to implement. Scary.” If so, try not to distinguish design from implementation. **Remember that when practicing DDD, the design is the code and the code is the design**. In other words, whiteboard diagrams aren’t the design, just a way to discuss the challenges of the model. |
| Ubiquitous Language | 21 | The Ubiquitous Language is a shared language developed by the team—a team composed of both domain experts and software developers.  Naturally, the domain experts have a heavy influence on the Language because they know that part of the business best and may be influenced by industry standards. However, the Language is more centered **on how the business itself thinks and operates**. Also, many times two or more domain experts disagree on concepts and terms, and they are actually wrong about some because they haven’t thought of every case before. So, as the experts and developers **work together to craft a model of the domain**, they use discussion with both consensus and compromise to achieve the very best Language for the project. The team **never compromises on the quality of the Language**, just on the best concepts, terms, and meanings. Initial consensus is not the end, however. The Language grows and changes over time as tiny and large breakthroughs are achieved, much like any other living language.  This is no gimmick to get developers to be on the same page as domain experts. It’s not just a bunch of business jargon being forced on developers. **It’s a real language that is created by the whole team**—domain experts, developers, business analysts, everyone involved in producing the system.  The artifacts that originally led us down an inspiring path to developing a useful Ubiquitous Language that was just right for our specialized domain **will very likely be rendered obsolete over time**. That’s why in the end **it is team speech and the model in the code** that are the most enduring and the only guaranteed current denotations of the Ubiquitous Language. |
|  | 24 | Example of source code that express the language:    You can read the code and easily comprehend it. You can also test it and confirm that it does exactly what it is meant to do, and that it doesn’t do anything that it shouldn’t.  Thus, the Ubiquitous Language is a team pattern used to capture the concepts and terms of a specific core business domain in the software model itself. The software model incorporates the nouns, adjectives, verbs, and richer expressions formally formulated and spoken by the close-knit team. Both the software and the tests that verify the model’s adherence to the tenets of the domain capture and adhere to this Language, the same one spoken by the team. |
| Bounded Context | 25 | There is one Ubiquitous Language per Bounded Context.  Bounded Contexts are relatively small, smaller than we might at first imagine. A Bounded Context is large enough only to capture the complete Ubiquitous Language of the isolated business domain, and no larger.  The Language is ubiquitous only within the team that is working on the project that develops in an isolated Bounded Context. |
| Context Maps | 25 | Bounded Context’s related to each other via Context Maps |
|  |  |  |
|  |  |  |

Personal Notes

|  |  |  |
| --- | --- | --- |
| Concept |  |  |
| Ubiquitous Language |  | The way the team talks and the code, it’s a reflection of the ubiquitous language. |
| Test Driven Development |  | Can we use TDD in DDD? Yes. But TDD is used to create tests that focus in the domain model, not business logic. The tests assert the domain model correctly express’s the “ubiquitous language”. The unit tests source code, uses the nouns, verbs, the “language” of the business domain.  We can show the unit test to a domain expert and he should be able to understand it with a help of a developer. |
| The problem of documentation | 27 | Developers now share a common Language as a unified team along with domain experts. They benefit further from the knowledge transfer from the domain experts they work with. **As developers inevitably move on, either to a new Core Domain or out of the organization, training and handoffs are easier**. The chances of developing “tribal knowledge,” where only a select few understand the model, are reduced. The experts, remaining developers, and new ones continue to share a common knowledge that is available to anyone in the organization who requires it. This advantage exists because there remains an express goal to adhere to the Language of the domain. |
| The fallacy of getters/setters | 31 | Again, what happens if we simply provide data accessors to our model? To reemphasize, if we only expose the data accessors for our model objects, the results will look much like a **data model**. |
|  |  |  |
|  |  |  |
|  |  |  |

## Aide Memory