Discovering Domain Knowledge

In this chapter we will learn the domain-driven design tool for effective communication

and knowledge sharing: **the ubiquitous language:**

* Here we will use it to learn the intricacies of business domains.
* Later in the book we will use it to model and implement their business logic in software.

# Business Problems

Software systems are solutions to business problems. In the context of business domains, “problem” has a broader meaning than just a puzzle or a math problem. A business problem can be challenges associated with optimizing workflows and processes, minimizing manual labor, managing resources, supporting decisions, managing data, and so on.

Business problems appear both at the business domain and subdomain levels. **Subdomains are finer-grained problem domains whose goal is to provide solutions for specific business capabilities.**

A knowledge management subdomain optimizes the process of storing and retrieving information. A clearing subdomain optimizes the process of executing financial transactions. An accounting subdomain keeps track of the company’s funds.

# Knowledge Discovery

To design an effective software solution, we have to grasp at least **the basic knowledge of the business domain**. As we discussed earlier, this knowledge belongs to domain experts.

By no means should we, nor can we, become domain experts. That said, **it’s crucial for us to understand domain experts and to use the same business terminology they use.**

To be effective, **the software has to mimic the domain experts’ way of thinking about**

**the problem—their mental models.**

*The next two sentences are not quite clear for me at this point but we will probably understand them better later on:*

**Without an understanding of the business problem and the reasoning behind the requirements, our solutions will be limited to “translating” business requirements into source code.**

What if the requirements miss a crucial edge case? Or fail to describe a business concept, limiting our ability to implement a model that will support future requirements?

Effective knowledge sharing between domain experts and software engineers requires

effective communication.

# Communication

It’s safe to say that almost all software projects require the collaboration of stakeholders

in different roles: domain experts, product owners, engineers, UI and UX designers,

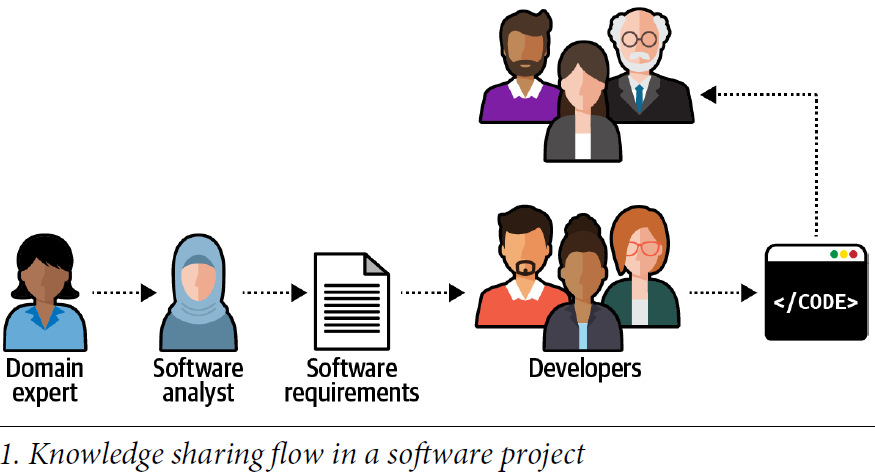
project managers, testers, analysts, and others. As in any collaborative effort, **the**

**outcome depends on how well all those parties can work together**:

* do all stakeholders agree on what problem is being solved?
* What about the solution they are building—do they hold any conflicting assumptions about its functional and nonfunctional requirements?

Agreement and alignment on all project-related matters are essential to a project’s success.

Below is the traditional flow of domain knowledge with a mediator/translator(analyst, project manager, etc.) between the domain experts and engineers:

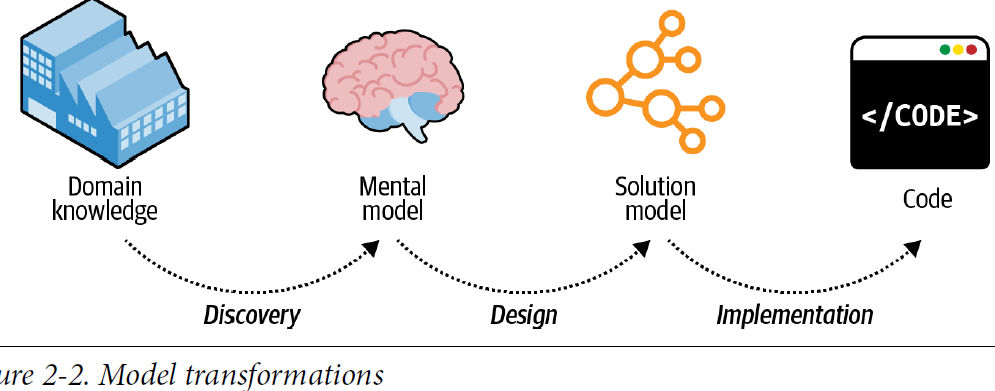


During the traditional software development lifecycle, the domain knowledge is “translated” into **an engineer-friendly form known as an *analysis model*, which is a description of the system’s requirements rather than an understanding of the business domain behind it.**

While the intentions may be good, such mediation is hazardous toknowledge sharing. In any translation, information is lost; in this case, domainknowledge that is essential for solving business problems gets lost on its way to thesoftware engineers. This is not the only such translation on a typical software project.

The analysis model is translated into the software design model (a software design document, which is translated into an implementation model or the source code

itself). **As often happens, documents go out of date quickly. The source code is used to communicate business domain knowledge to software engineers who will maintain the project later**. The figure below illustrates the different translations needed for domain knowledge to be implemented in code.



Such a software development process resembles the children’s game Telephone:3 the

message, or domain knowledge, often becomes distorted. The information leads to

software engineers implementing the wrong solution, or the right solution but to the

wrong problems. In either case, the outcome is the same: a failed software project.

Domain-driven design proposes a better way to get the knowledge from domain

experts to software engineers: by using a ubiquitous language.

# What is a Ubiquitous Language?

Using a ubiquitous language is the cornerstone practice of domain-driven design. The idea is simple and straightforward: if parties need to communicate efficiently, **instead of relying on translations, they have to speak the same language.**

The traditional software development lifecycle implies the following translations:

• Domain knowledge into an analysis model

• Analysis model into requirements

• Requirements into system design

• System design into source code

**Instead of continuously translating domain knowledge, domain-driven design calls for cultivating a single language for describing the business domain: the ubiquitous language.**

**All project-related stakeholders**—software engineers, product owners, domain experts, UI/UX designers—**should use the ubiquitous language when describing the business domain.** Most importantly, domain experts must be comfortable using the ubiquitous language when reasoning about the business domain; **this language will represent both the business domain and the domain experts’ mental models**. Only through the continuous use of the ubiquitous language and its terms can a shared understanding among all of the project’s stakeholders be cultivated.

## Language of the Business

**the ubiquitous language is the language of the business. As such, it should consist of business domain–related terms only. No technical jargon!**

Teaching business domain experts about singletons and abstract factories is not your goal. The ubiquitous language aims to frame the domain experts’ understanding and mental models of the business domain in terms that are easy to understand.

The technical terms like tables, records, etc. will be unclear to domain experts and if engineers are only dealing with technical terms and have a **solution-oriented view of the business domain** and not the language that reflects the business domain, **they won’t be able to completely understand the business logic or why it operates the way it does, which will limit their ability to model and implement an effective solution.**

## Consistency

The ubiquitous language must be precise and consistent. It should eliminate the need for assumptions and should make the business domain’s logic explicit. Since ambiguity hinders communication, **each term of the ubiquitous language should have one and only one meaning.**

*I add: It means that each term has one explicit meaning and can’t be interpreted as multiple concepts and each concept is also represented by one term not synonymous words:*

### Example of Ambiguity

Let’s say that in some business domain, the term *policy* has multiple meanings: it can

mean a regulatory rule or an insurance contract. **The exact meaning can be worked out in human-to-human interaction, depending on the context. Software, however, doesn’t cope well with ambiguity, and it can be cumbersome and challenging to model the “policy” entity in code.**

Ubiquitous language demands a single meaning for each term, so “policy” should be modeled explicitly using the two terms regulatory rule and insurance contract.

### Example of Synonymous terms

**Two terms cannot be used interchangeably in a ubiquitous language**. For example,

many systems use the term *user*. However, a careful examination of the domain experts’ lingo may reveal that *user* and other terms are used interchangeably: for example, *user*, *visitor*, *administrator*, *account*, etc.

Synonymous terms can seem harmless at first. However, in most cases, they denote different concepts. In this example, both *visitor* and *account* **technically** refer to the

System’s users; however, in most systems, unregistered and registered users represent different roles and have different behaviors. For example, the “visitors” data is used mainly for analysis purposes, whereas “accounts” actually uses the system and its functionality.

It is preferable to use each term explicitly in its specific context.

**Understanding the differences between the terms in use allows for building simpler and clearer models and implementations of the business domain’s entities.**

# Model of the Business Domain

let’s look at the ubiquitous language from a different perspective: modeling.

## What is a Model?

**A model is not a copy of the real world** but a human construct that helps us make

sense of real-world systems.

A canonical example of a model is a map. Any map is a model, including navigation

maps, terrain maps, world maps, etc.

None of these maps represents all the details of our planet. Instead, **each map contains just enough data to support its particular purpose: the problem it is supposed to solve.**

## Effective Modeling

All models have a purpose, and an effective model **contains only** the details needed to

fulfill its purpose. For example, **you won’t see subway stops on a world map**. On the

other hand, you cannot use a subway map to estimate distances. **Each map contains just the information it is supposed to provide.**

## Modeling of the Business Domain

**When cultivating a ubiquitous language, we are effectively building a model of the business domain.**

**The model is supposed to capture the domain experts’ mental models**— their thought processes about how the business works to implement its function.

T**he model has to reflect:**

* the involved business entities
* their behavior
* cause and effect relationships
* and invariants

**The ubiquitous language we use is not supposed to cover every possible detail of the domain.** That would be equivalent to making every stakeholder a domain expert(How? Why?)

Instead, the model is supposed to include just enough aspects of the business domain

to make it possible to implement the required system; that is, to address the specific

problem the software is intended to solve. **we will see how the ubiquitous language can drive low-level design and implementation decisions.**

The only reliable way to verify a business domain’s understanding is to converse with domain experts and **do it in the language they understand: the language of the business.**

# Continuous Effort

All stakeholders should consistently use the ubiquitous language in all project-related

communications to spread knowledge about and foster a shared understanding of the

business domain. **The language should be continuously reinforced throughout the**

**project: requirements, tests, documentation, and even the source code itself should**

**use this language.**

Most importantly, cultivation of a ubiquitous language **is an ongoing process**. It

should be constantly validated and evolved. Everyday use of the language will, over

time, reveal deeper insights into the business domain. **When such breakthroughs**

**happen, the ubiquitous language must evolve to keep pace with the newly acquired**

**domain knowledge.**

# Tools

There are tools and technologies that can alleviate **the processes of capturing and managing a ubiquitous language.**

For example, a wiki can be used as a *glossary* to capture and document the ubiquitous language.

It’s important to make glossary maintenance a shared effort. When a ubiquitous language

is changed, all team members should be encouraged to go ahead and update the glossary.

Despite the obvious advantages of maintaining a glossary of project-related terminology, it has an inherent limitation. **Glossaries work best for “nouns”**: names of entities,

processes, roles, and so on. Although nouns are important, capturing the behavior is

crucial. **The behavior is not a mere list of verbs associated with nouns, but the actual**

**business logic, with its rules, assumptions, and invariants.** Such concepts are much

harder to document in a glossary. Hence, glossaries are best used in tandem with

**other tools that are better suited to capture the behavior; for example, use cases or**

**Gherkin tests.**

Automated tests written in the Gherkin language are not only great tools for capturing

the ubiquitous language but also act as an additional tool for bridging the gap

between domain experts and software engineers. **Domain experts can read the tests**

**and verify the system’s expected behavior**.5 For example, see the following test written

in the Gherkin language:

**Scenario:** Notify the agent about a new support case

**Given** Vincent Jules submits a new support case saying:

**"""**

I need help configuring AWS Infinidash

**"""**

**When** the ticket is assigned to Mr. Wolf

**Then** the agent receives a notification about the new ticket

Managing a Gherkin-based test suite can be challenging at times, especially at the

early stages of a project. However, it is definitely worth it for complex business

domains.

Finally, there are even static code analysis tools that can verify the usage of a ubiquitous

Language’s terms. A notable example for such a tool is NDepend.

While these tools are useful, they are secondary to the actual use of a ubiquitous language

in day-to-day interactions.

# Challenges

There may be ambiguities and even white spots in domain experts’ own understanding of the business domain; for example, defining only the “happy path” scenarios but not considering edge cases that challenge the accepted assumptions.

Furthermore, you may encounter business domain concepts that lack explicit definitions. Asking questions about the nature of the business domain often makes such implicit conflicts and white spots explicit. This is especially common for core subdomains. In such a case, the learning process is mutual—you are helping the domain experts better understand their field.

When introducing domain-driven design practices to a brownfield project, you will notice that there is already a formed language for describing the business domain, and that the stakeholders use it. However, **since DDD principles do not drive that language,**

**it won’t necessarily reflect the business domain effectively**. For example, it may

use technical terms, such as database table names.

Changing a language that is already being used in an organization is not easy. The essential tool in such a situation is patience**. You need to make sure the correct language is used where it’s easy to control it: in the documentation and source code.**

what language should we use if the company is not in an English-speaking country? My advice is to at least use English nouns for naming the business domain’s entities. This will alleviate using the same terminology in code.

# Conclusion

**Domain-driven design’s ubiquitous language is an effective tool for bridging the**

**knowledge gap between domain experts and software engineers. It fosters communication and knowledge sharing by cultivating a shared language that can be used by all the stakeholders throughout the project: in conversations, documentation, tests, diagrams, source code, and so on.**

**All of the project’s stakeholders (domain experts, engineers, and other team members) should be able to contribute to the definition of a ubiquitous language not just team leaders for example.**

As you gain experience in the practice of developing a ubiquitous language and discovering domain knowledge by conversing with the domain experts, you will notice that frequently, this process involves not merely discovering knowledge that is already there, but rather cocreating the model in tandem with domain experts.