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 using strategic design

In this section, let's demystify some commonly used concepts and terms when working with domain-driven design. First and foremost, we need to understand what we mean by the first "D" — **domain**.

What is a domain?

The foundational concept when working with domain-driven design is the notion of a domain. But what exactly is a domain? The word *domain*, which has its origins in the 1600s to the Old French word *domaine* (power), Latin word *dominium* (property, right of ownership) is a rather confusing word. Depending on who, when, where and how it is used, it can mean different things:

Noun [edit]

domain (*plural* **domains**)

1. A geographic area owned or controlled by a single person or organization. [quotations ▼]
   *The king ruled his **domain** harshly.*
2. A field or sphere of activity, influence or expertise.
   *Dealing with complaints isn't really my **domain**: get in touch with customer services.*
   *His **domain** is English history.*
3. A group of related items, topics, or subjects. [quotations ▼]
4. (*mathematics*) The set of all possible mathematical entities (points) where a given function is defined.
5. (*mathematics*, *set theory*) The set of input (argument) values for which a function is defined.
6. (*mathematics*) A ring with no zero divisors; that is, in which no product of nonzero elements is zero.
   Hyponym: integral domain
7. (*mathematics*, *topology*, *mathematical analysis*) An open and connected set in some topology. For example, the interval (0,1) as a subset of the real numbers.
8. (*computing*, *Internet*) Any DNS domain name, particularly one which has been delegated and has become representative of the delegated domain name and its subdomains. [quotations ▼]
9. (*computing*, *Internet*) A collection of DNS or DNS-like domain names consisting of a delegated domain name and all its subdomains.
10. (*computing*) A collection of information having to do with a domain, the computers named in the domain, and the network on which the computers named in the domain reside.
11. (*computing*) The collection of computers identified by a domain's domain names.
12. (*physics*) A small region of a magnetic material with a consistent magnetization direction.
13. (*computing*) Such a region used as a data storage element in a bubble memory.
14. (*data processing*) A form of technical metadata that represent the type of a data item, its characteristics, name, and usage. [quotations ▼]
15. (*taxonomy*) The highest rank in the classification of organisms, above kingdom; in the three-domain system, one of the taxa *Bacteria*, *Archaea*, or *Eukaryota*.
16. (*biochemistry*) A folded section of a protein molecule that has a discrete function; the equivalent section of a chromosome

Figure 1. **Domain***: Means many things depending on context*

In the context of a business however, the word domain covers the overall scope of its primary activity — the service it provides to its customers. This is also referred as the ***problem domain***. For example, Tesla operates in the domain of electric vehicles, Netflix provides online movies and shows, while McDonald's provides fast food. Some companies like Amazon, provide services in more than one domain — online retail, cloud computing, among others. The domain of a business (at least the successful ones) almost always encompasses fairly complex and abstract concepts. To cope with this complexity, it is usual to decompose these domains into more manageable pieces called subdomains. Let us understand subdomains in more detail next.

What is a subdomain?

At its essence, Domain-driven design provides means to tackle complexity. Engineers do this by breaking down complex problems into more manageable ones. the domain of a business into multiple manageable parts called **subdomains**. This facilitates better understanding and makes it easier to arrive at a solution. For example, the online retail domain may be divided into subdomains such as product, inventory, rewards, shopping cart, order management, payments,

shipping, etc. as shown below:

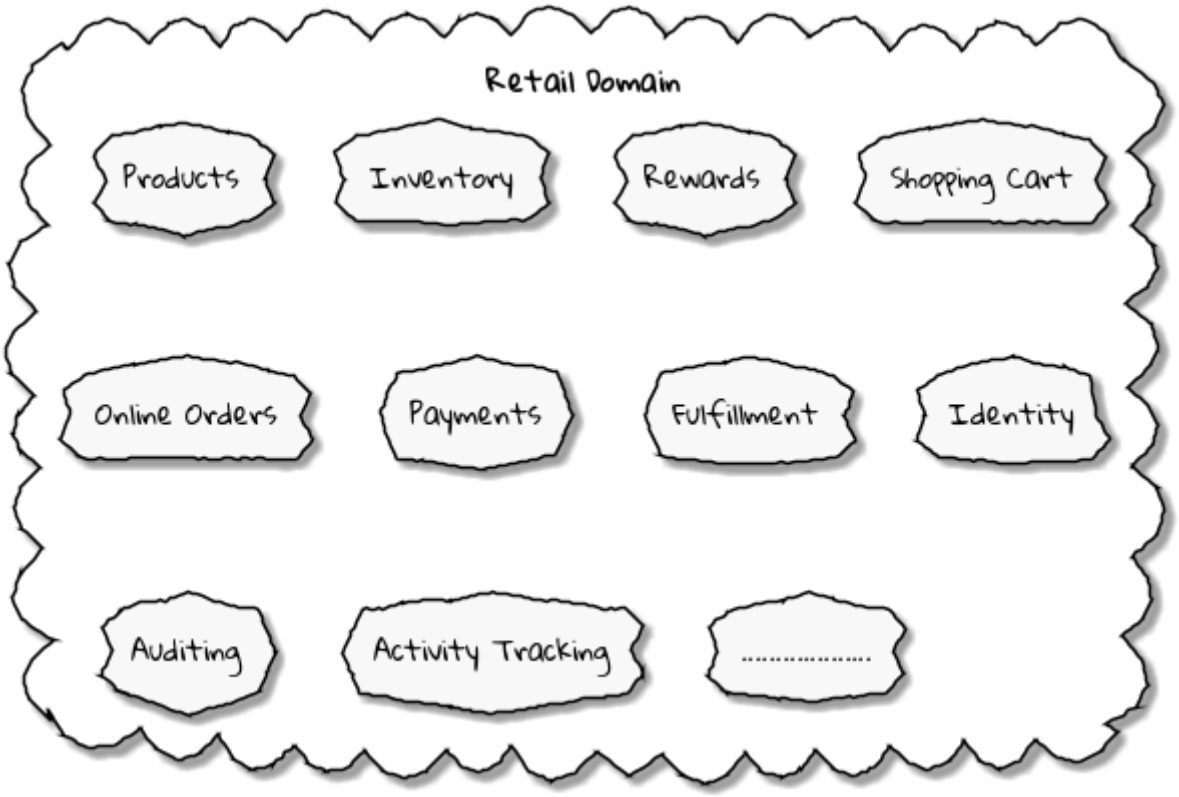

Figure 2. Subdomains in the Retail domain

In certain businesses, subdomains themselves may turn out to become very complex on their own and may require further decomposition. For instance, in the retail example above, it may be required to break the products subdomain into further constituent subdomains such as catalog, search, recommendations, reviews, etc. as shown below:

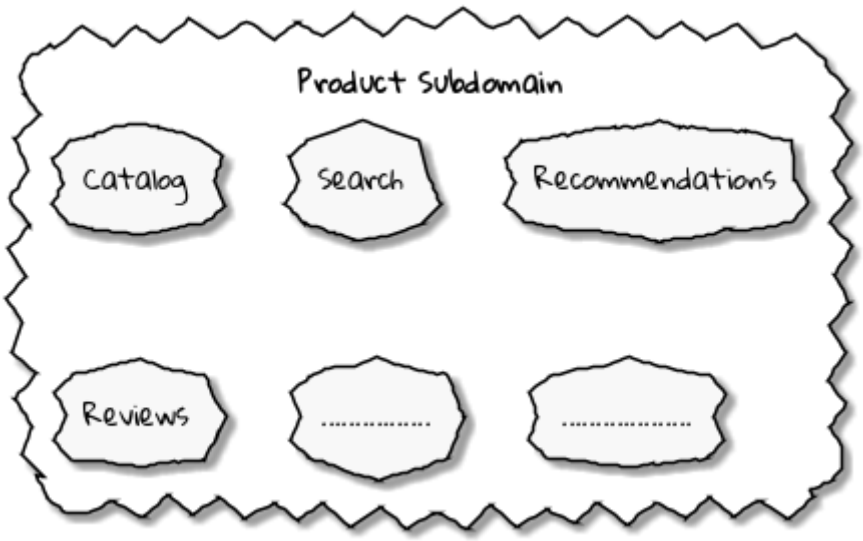

Figure 3. Subdomains in the Products subdomain

Further breakdown of subdomains may be needed until we reach a level of manageable complexity. Domain decomposition is an important aspect of DDD. Let's look at the types of subdomains to understand this better.

Types of subdomains

Breaking down a complex domain into more manageable subdomains is a great thing to do. However, not all subdomains are created equal. With any business, the following three types of subdomains are going to be encountered:

- **Core**: The main focus area for the business. This is what provides the biggest differentiation and value. It is therefore natural to want to place the most focus on the core subdomain. In the retail example above, shopping cart and orders might be the biggest differentiation — and hence may form the core subdomains for that business venture. It is prudent to implement core subdomains in-house given that it is something that businesses will desire to have the most control over. In the online retail example above,
- **Supporting**: Like with every great movie, where it is not possible to create a masterpiece without a solid supporting cast, so it is with supporting or auxiliary subdomains. Supporting subdomains are usually very important and very much required, but may not be the primary focus to run the business. These supporting subdomains, while necessary to run the business, do not usually offer a significant competitive advantage. Hence, it might be even fine to completely outsource this work or use an off-the-shelf solution as is or with minor tweaks. For the retail example above, assuming that online ordering is the primary focus of this business, catalog management may be a supporting subdomain.
- **Generic**: When working with business applications, one is required to provide a set of capabilities **not** directly related to the problem being solved. Consequently, it might suffice to just make use of an off-the-shelf solution. For the retail example above, the identity, auditing and activity tracking subdomains might fall in that category.

It is important to note that the notion of core vs. supporting vs. generic subdomains is very context specific. What is core for one business may be supporting or generic for another. Identifying and distilling the core domain requires deep understanding and experience of what problem is being attempted to be solved.

Given that the core subdomain establishes most of the business differentiation, it will be prudent to devote the most amount of energy towards maintaining that differentiation. This is illustrated in the core domain chart here:

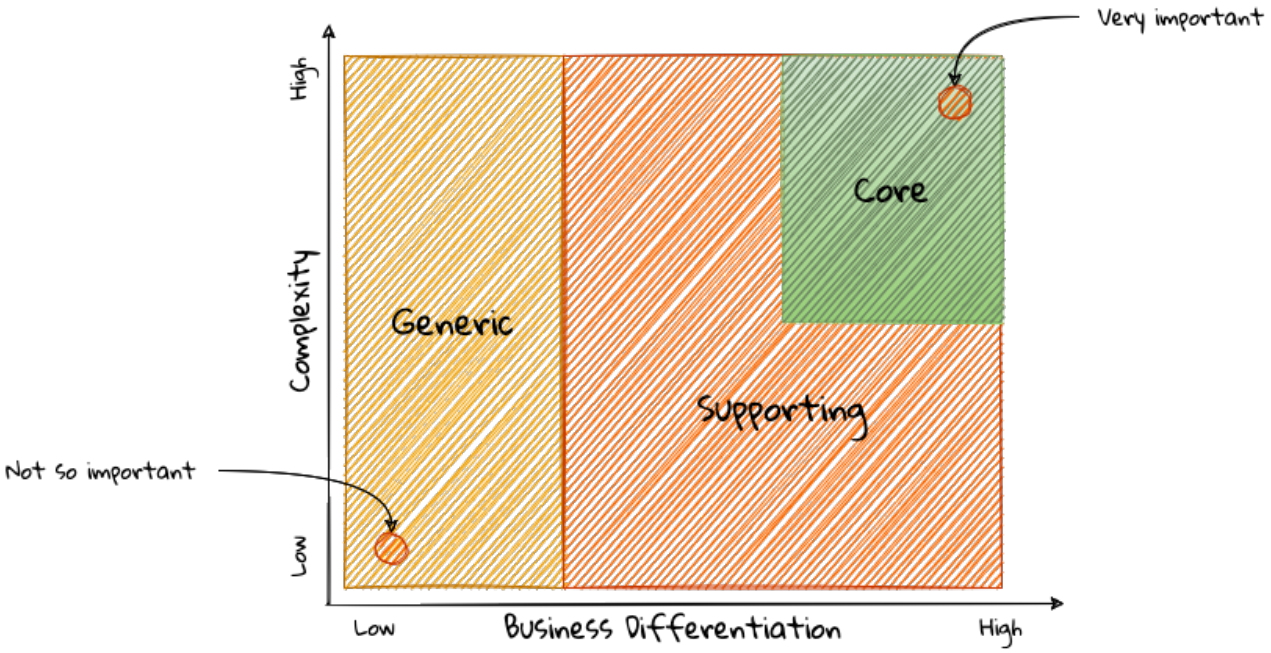

Figure 4. Importance of subdomains

Over a period of time, it is only natural that competitors will attempt to emulate your successes. Newer, more efficient methods will arise, reducing the complexity involved, disrupting your core. This may cause the notion of what is currently core, to shift and become a supporting or generic capability.

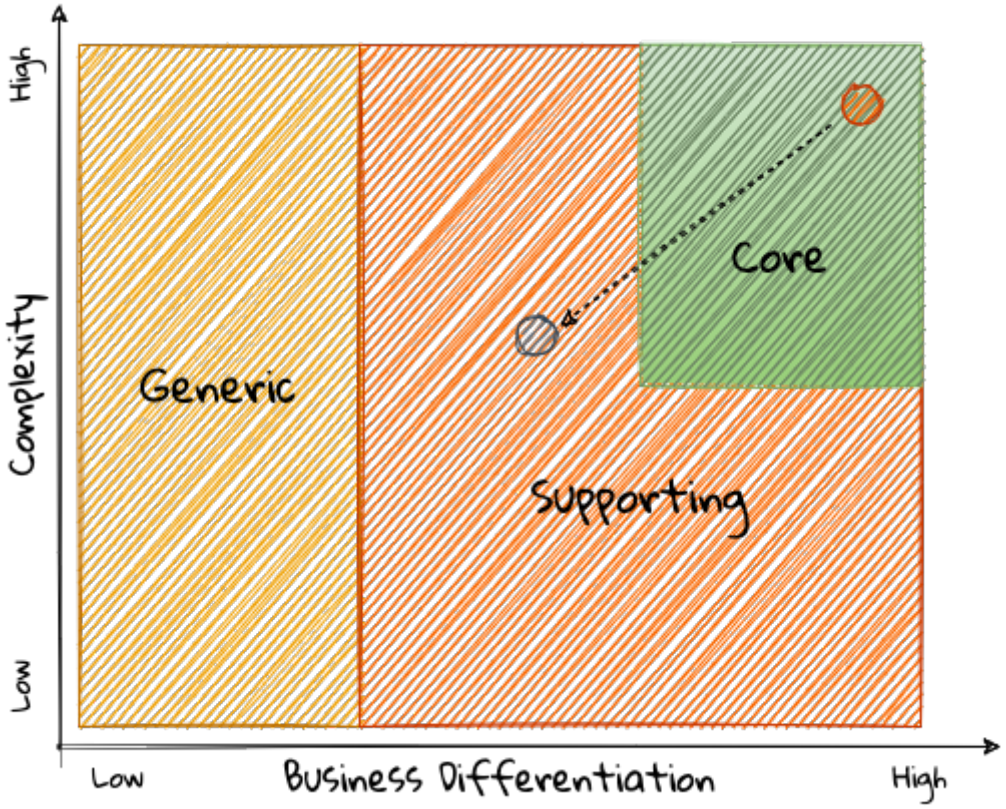

Figure 5. Core domain erosion

To continue running a successful operation, it is required to constantly innovate in the core. For example, when AWS started the cloud computing business, it only provided simple infrastructure

(IaaS) solutions. However, as competitors like Microsoft, Google and others started to catch up, AWS has had to provide several additional value-added services (for example, PaaS, SaaS, etc).

As is evident, this is not just an engineering problem. It requires deep understanding of the underlying business. That's where domain experts can play a significant role.

Domain and technical experts

Any modern software team requires expertise in at least two areas — the functionality of the domain and the art of translating it into high quality software. At most organizations, these exist as at least two distinct groups of people.

Domain experts — those who have a deep and intimate understanding of the domain. Domain experts are subject-matter experts (SMEs) who have a very strong grasp of the business. Domain experts may have varying degrees of expertise. Some SMEs may choose to specialize in specific subdomains, while others may have a broader understanding of how the overall business works.

Technical experts on the other hand, enjoy solving specific, quantifiable computer science problems. Often, technical experts do not feel it worth their while understanding the context of the business they work in. Rather, they seem overly eager to only enhance their technical skills that are a continuation of their learnings in academia.

While the domain experts specify the **why** and the ***what***, technical experts, (software engineers) largely help realize the ***how***. Strong collaboration and synergy between both groups is absolutely essential to ensure sustained high performance and success.

A divide originating in language

While strong collaboration between these groups is necessary, it is important to appreciate that these groups of people seem to have distinct motivations and differences in thinking. Seemingly, this may appear to be restricted to simple things like differences in their day-to-day language. However, deeper analysis usually reveals a much larger divide in aspects such as goals, motivations etc. This is illustrated in the picture here:

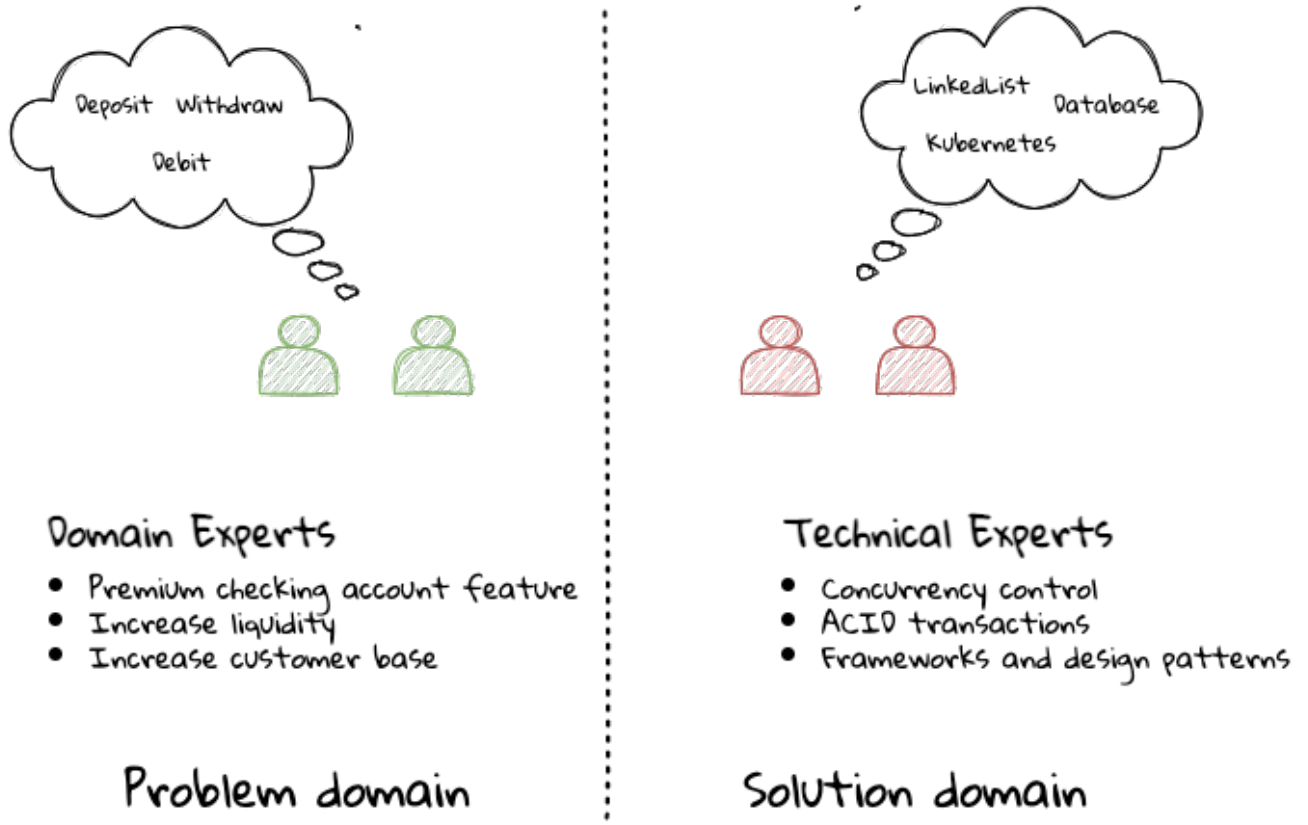

Figure 6. Divide originating in language

But this is a book primarily focused towards technical experts. Our point is that it is not possible to be successful by just working on technically challenging problems without gaining a sound understanding of the underlying business context.

Every decision we take regarding the organization, be it requirements, architecture, code, etc. 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. As mentioned above, 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 process of refinement is usually messy and 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.

Promoting a shared understanding using a ubiquitous language

Previously, we saw how organizational silos can result in valuable information getting diluted. At a credit card company I used to work with, the words plastic, payment instrument, account, PAN (Primary Account Number), BIN (Bank Identification Number), card were all used by different team

members to mean the exact same thing - the ***credit card*** when working in the same area of the application. On the other hand, a term like ***user*** would be used to sometimes mean a customer, a relationship manager, a technical customer support employee. To make matters worse, a lot of these muddled use of terms got implemented in code as well. While this might feel like a trivial thing, it had far-reaching consequences. Product experts, architects, developers, all came and went, each regressively contributing to more confusion, muddled designs, implementation and technical debt with every new enhancement — accelerating the journey towards the dreaded, unmaintainable, big ball of mud.

DDD advocates breaking down these artificial barriers, and putting the domain experts and the developers on the same level footing by working collaboratively towards creating what DDD calls a ***ubiquitous language*** — a shared vocabulary of terms, words, phrases to continuously enhance the collective understanding of the entire team. This phraseology is then used actively in every aspect of the solution: the everyday vocabulary, the designs, the code — in short by **everyone** and **everywhere**. Consistent use of the common ubiquitous language helps reinforce a shared understanding and produce solutions that better reflect the mental model of the domain experts.

Evolving a domain model and a solution

The ubiquitous language helps establish a consistent albeit informal lingo among team members. To enhance understanding, this can be further refined into a formal set of abstractions — a ***domain model*** to represent the solution in software. 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:

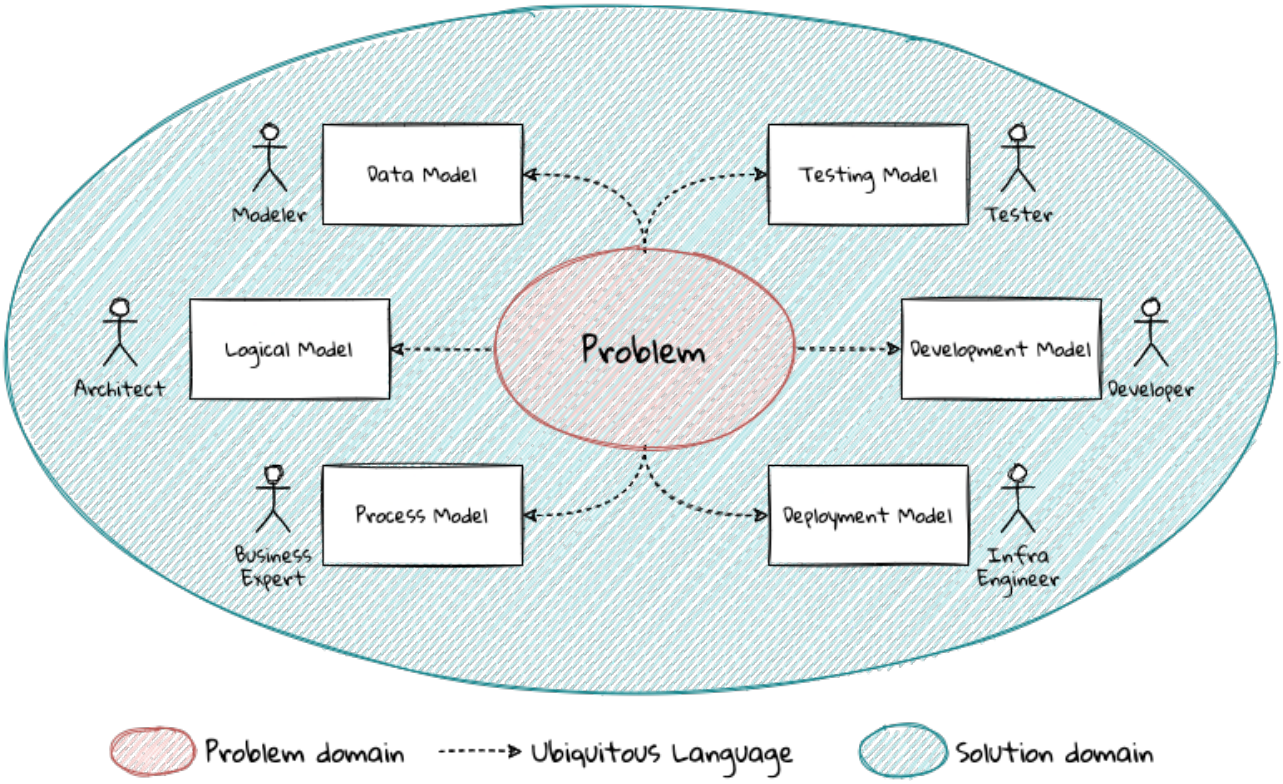

Figure 7. Multiple models to represent the solution to the problem using the ubiquitous language

As illustrated 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 and so on.

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.

Care should be taken to retain focus on solving the business problem at hand at all times. Teams will be better served if they expend the same amount of effort modeling business logic as the technical aspects of the solution. To keep accidental complexity in check, it will be best to isolate the infrastructure aspects of the solution from this model. These models can take several forms, including conversations, whiteboard sessions, documentation, diagrams, tests and other forms of architecture fitness functions. It is also important to note that this is **not** a one-time activity. As the business evolves, the domain model and the solution will need to keep up. This can only be achieved through close collaboration between the domain experts and the developers at all times.

Anemic domain models

Scope of domain models and the bounded context

When creating domain models, one of the dilemmas is in deciding how to restrict the scope of these models. One can attempt to create a single domain model that acts as a solution for the entire problem. On the other hand, we may go the route of creating extremely fine-grained models that cannot exist meaningfully without having a strong dependency on others. There are pros and cons in going each way. Whatever be the case, each solution has a scope — bounds to which it is confined to. This boundary is termed as a **bounded context**.

There seems to exist a lot of confusion between the terms subdomains and bounded contexts. What is the difference? It turns out that subdomains are problem space concepts whereas bounded contexts are solution space concepts. This is best explained through the use of an example. Let's consider the example of a fictitious Acme bank that provides two products: credit cards and retail bank. This may decompose to the following subdomains as depicted here:

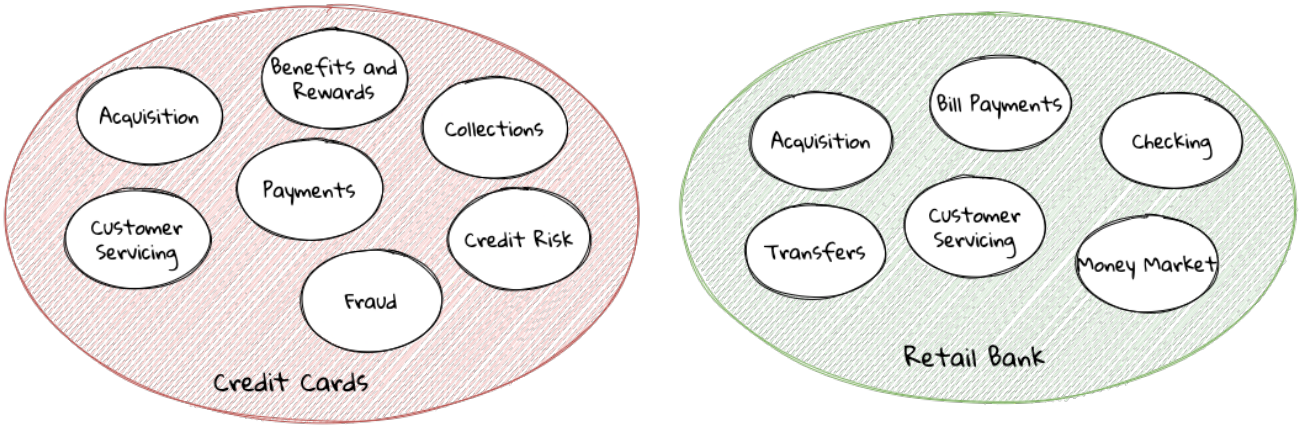

Figure 8. Banking subdomains at Acme bank

When creating a solution for the problem, many possible solution options exist. We have depicted a few options here:

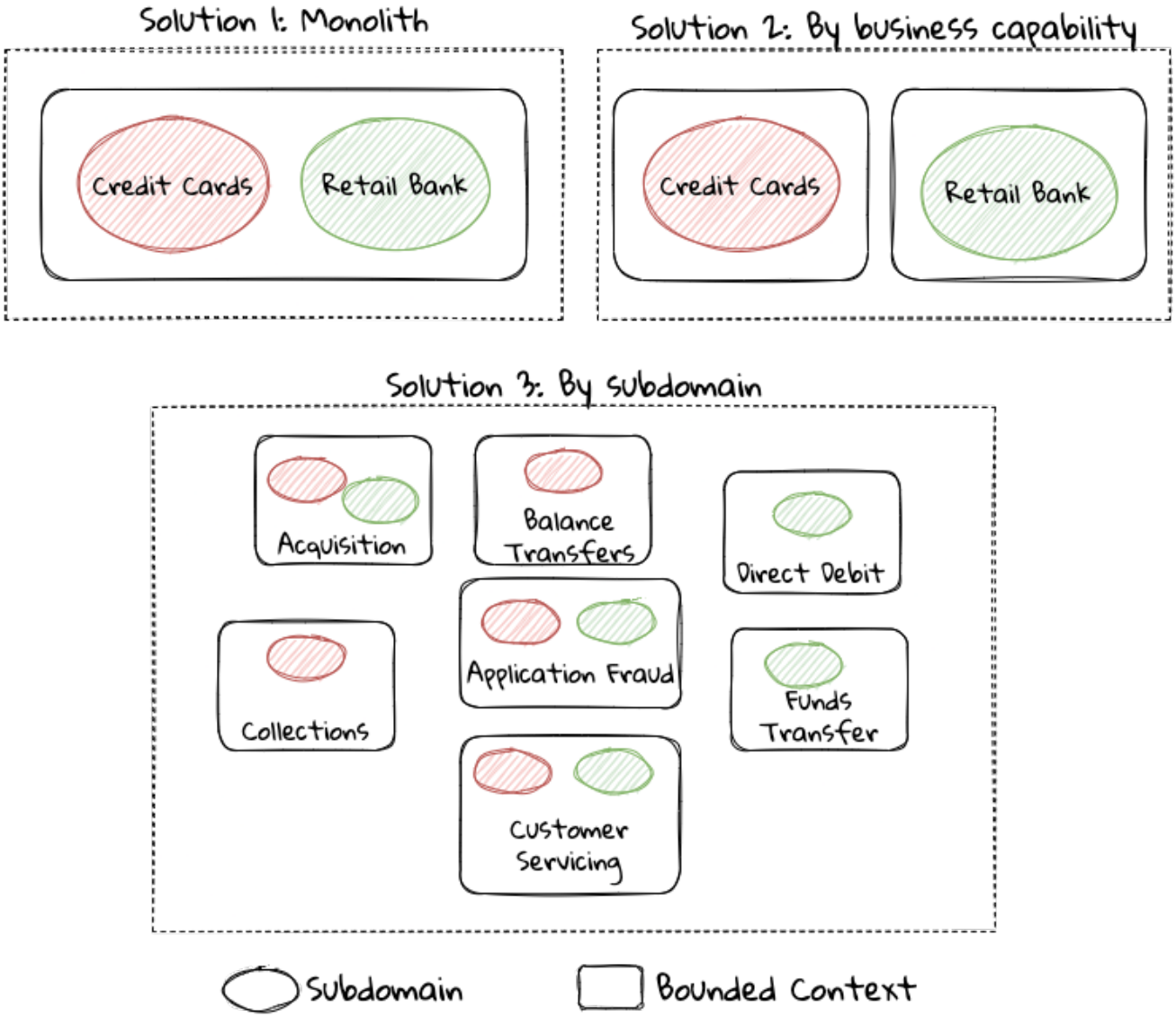

Figure 9. Bounded contexts options at Acme bank

These are just a few examples of decomposition patterns to create bounded contexts. The exact set of patterns one may choose to use may vary depending on currently prevailing realities like:

- Current organizational structures
- Domain experts' responsibilities
- Key activities and pivotal events
- Existing applications

Whatever be the method used to decompose a problem into a set of bounded contexts, care should be taken to make sure that the coupling between them is kept as low as possible.

While bounded contexts ideally need to be as independent as possible, they may still need to communicate with each other. When using domain-driven design, the system as a whole can be represented as a set of bounded contexts which have relationships with each other. These relationships define how these bounded contexts can integrate with each other and are called ***context maps***. A sample context map is shown here.

[sample context map] | *sample-context-map.png*

We will discuss more details on context maps and communication patterns in Chapter 9: Integrating with external systems.

We have now covered a catalog of concepts that are core to the strategic design tenets of domain-driven design. Let's look at some tools that can help expedite this process.

Strategic design tools

To arrive at an optimal solution, it is important to have a strong appreciation of the business goals and its alignment to support the needs of the users of the solution. We introduce a set of tools and techniques we have found to be useful.

These tools are not really tied to DDD in any way and can be practiced regardless. The use of these should be considered to be complementary in your DDD journey.

Wardley maps

A tool that can be used to map your business strategy. It integrates value chain with evolution. Value chain is the chain of systems or activities performed to provide value to the customer. Each of these activities is then placed on the map based on its evolutionary stage.

Impact maps

A visual technique that helps to identify the features for product development that align with primary business goals.

Business model canvas

A strategic tool that enables you to summarize your business model. It's a single image document which focuses on key drivers of your business such as customers, infrastructure, revenue and resources.

Product strategy canvas

A system of achievable goals and visions that work together to align the team around desirable outcomes for both the business and your customers.

Lean canvas

A one-page document that has been adapted from Business Model Canvas that is entrepreneur focused and has customer-centric approach that emphasizes on problem, solution, key metrics and competitive advantage.

Implementing the solution using tactical design

Entities

Entity is an object with unique identity and encapsulates the object behaviour and attributes.

Value objects

Value objects are immutable objects which do not have an identity but hold value and are defined by their attributes.

Aggregates

When entities and objects with similar logic are grouped together, they are called as aggregates.

Services

Business rules within the domain are encapsulated within domain services.

Repositories

Repositories are objects that provide persistence while retrieving domain objects.

Factories

Factories are methods used to create complex objects or aggregates.

Tactical design tools

Behavior-driven development

Test-driven development

Contract testing

Refactoring

Quality storming