Table of Contents

The Mechanics of Domain-Driven Design

> 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

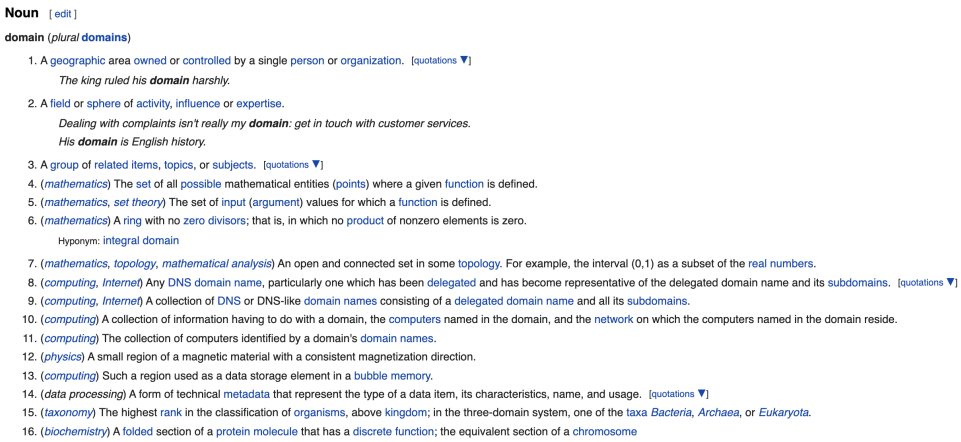
manageable chunks and translate it into reliably working software.

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:

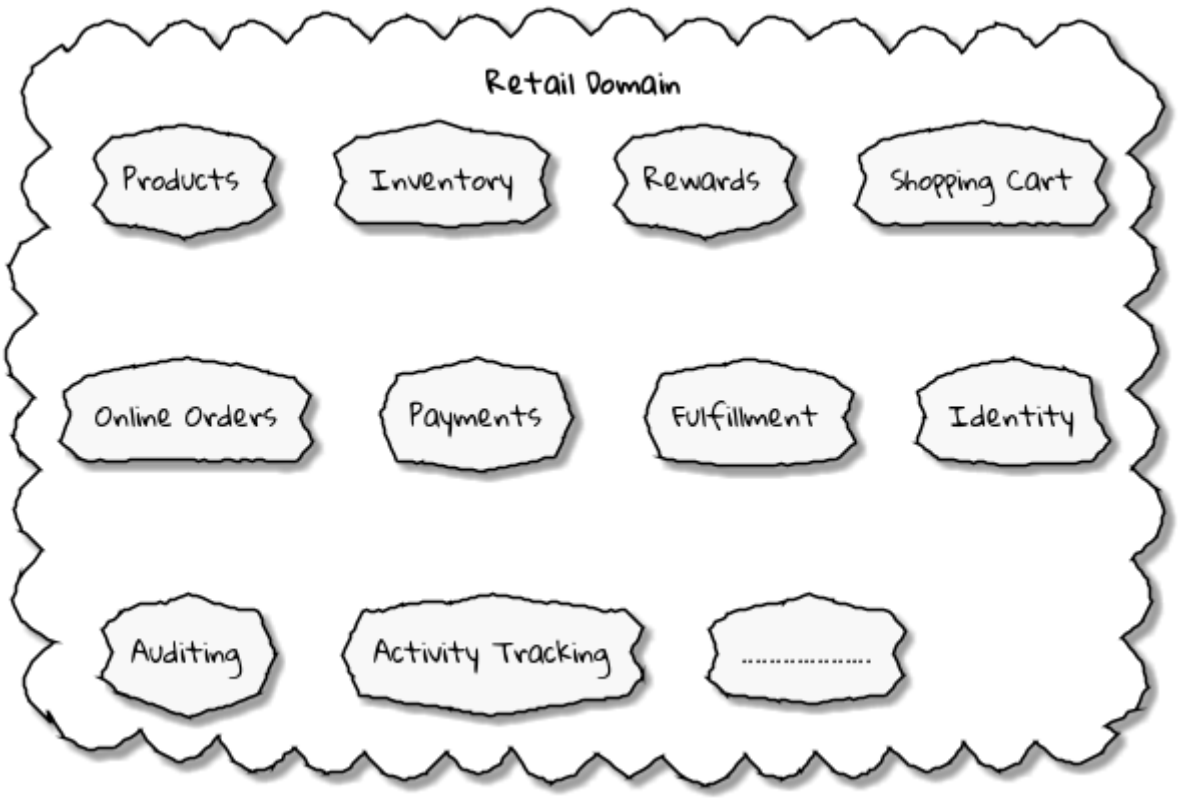


Figure 2. Subdomains in the Retail domain

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:

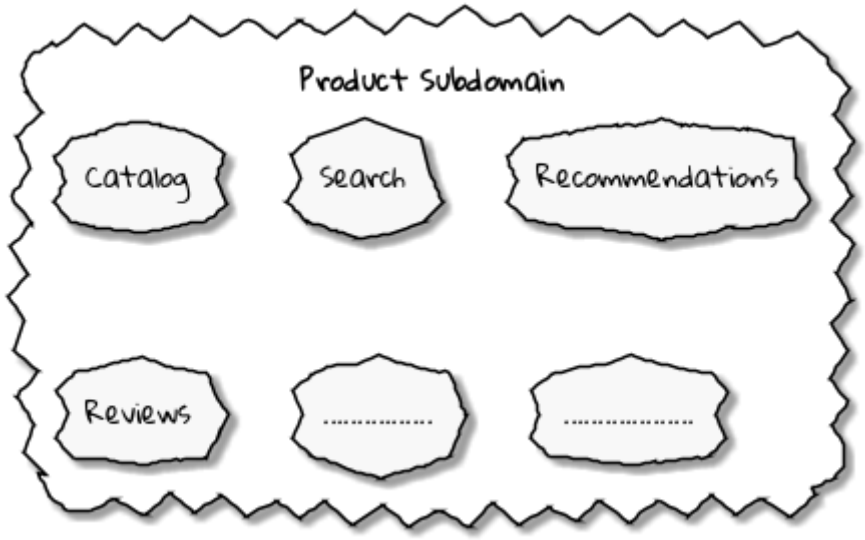


Figure 3. Subdomains in the Products subdomain

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, the business may want to focus on providing an enriched experience to place online orders. This will make the *online orders* and *shopping cart* part of the core domain.
- **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:

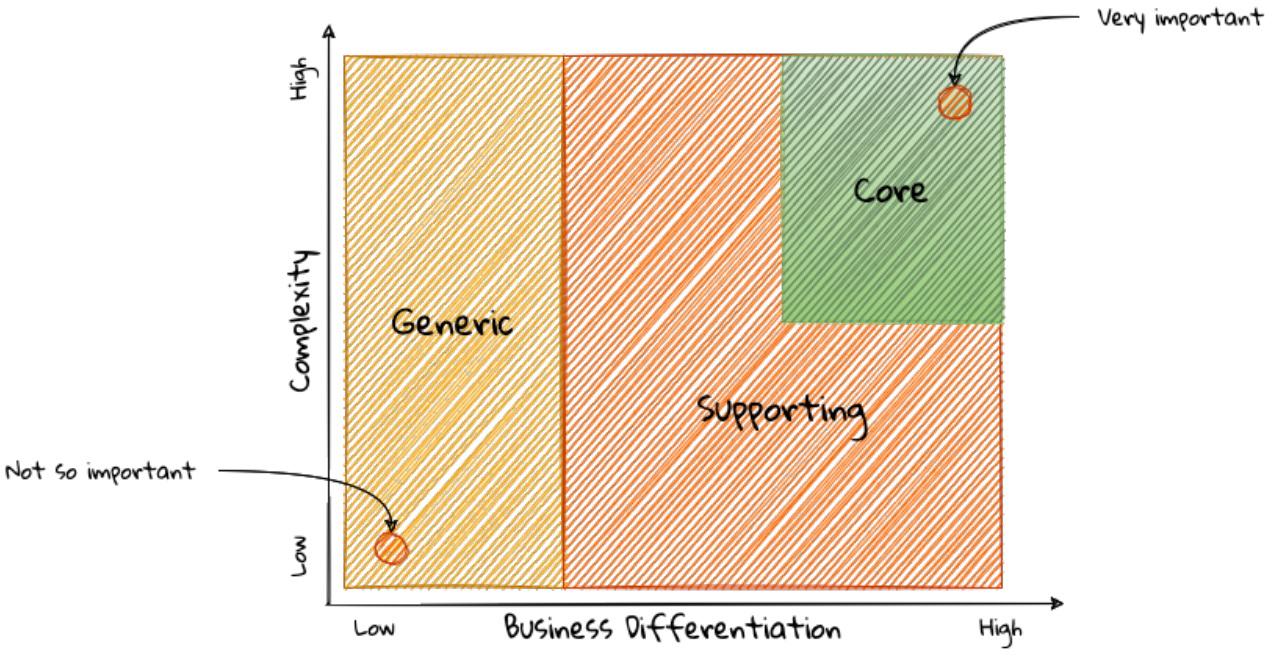


Figure 4. Importance of subdomains

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 as depicted here:

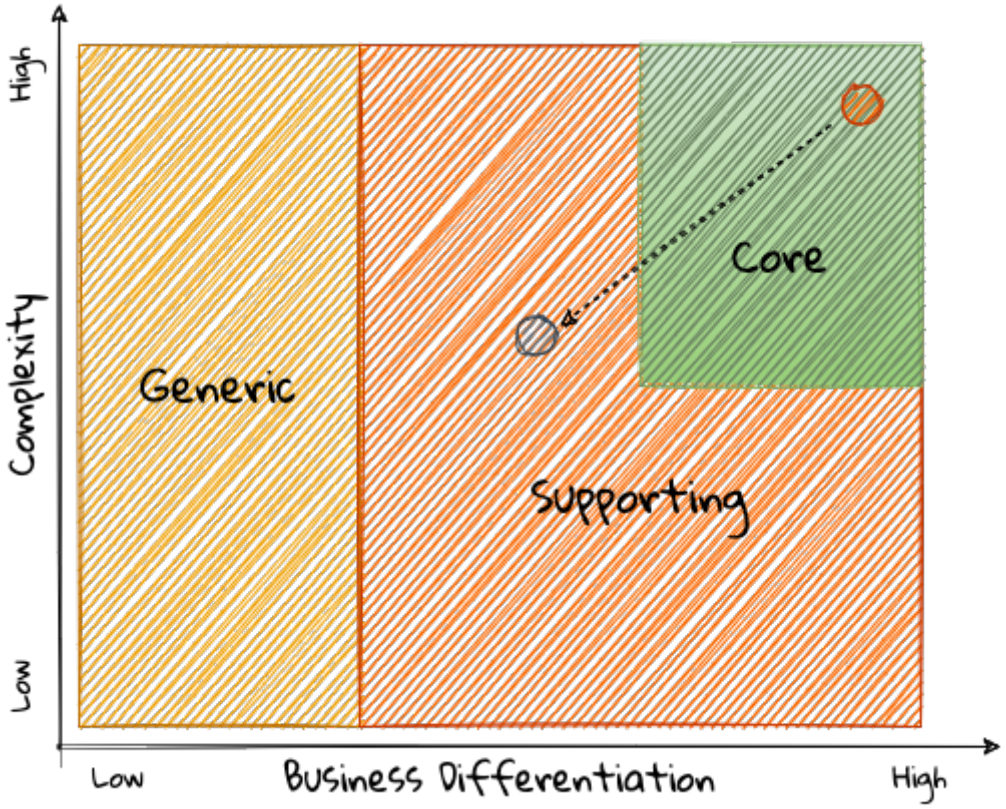


Figure 5. Core domain erosion

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:

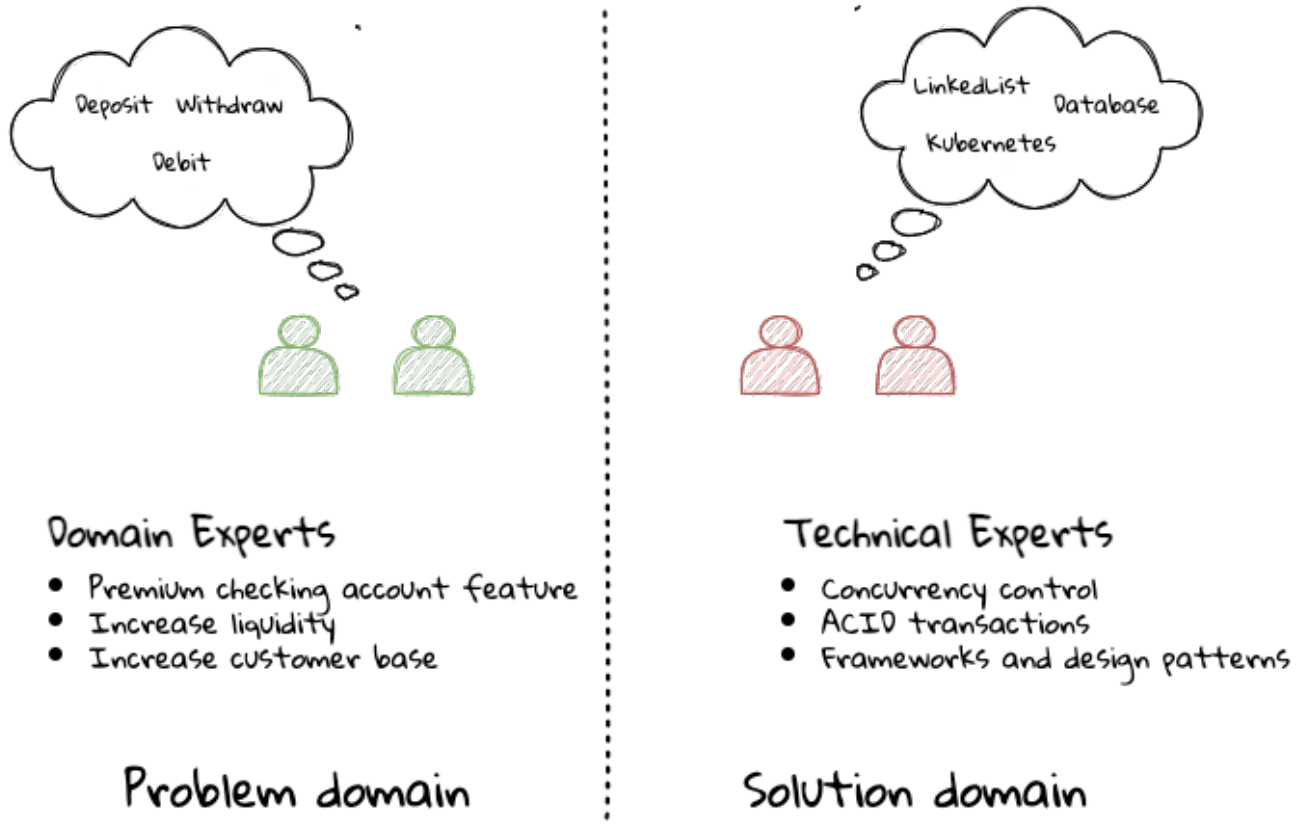


Figure 6. Divide originating in language

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

Now that we have defined the problem and the motivations surrounding it, let's examine how it can inform the solution.

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:

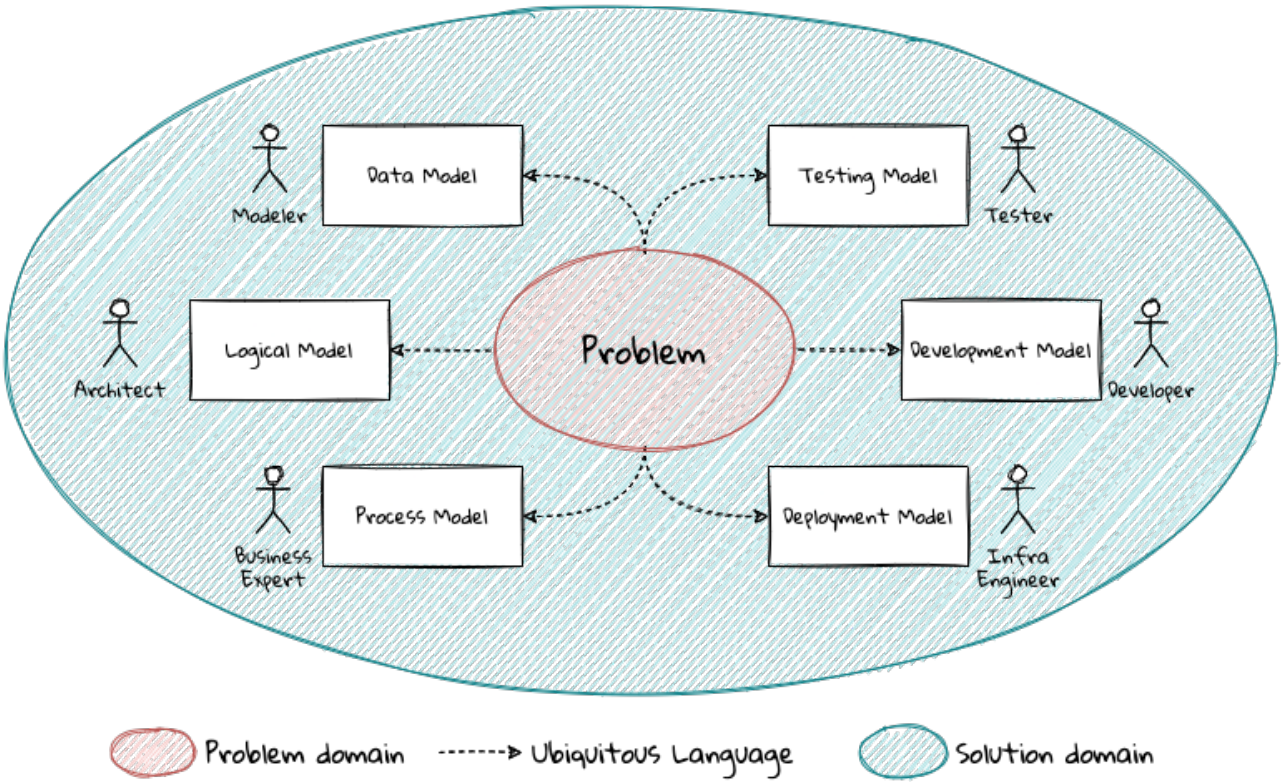


Figure 7. Multiple models to represent the solution to the problem using the ubiquitous language

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.

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:

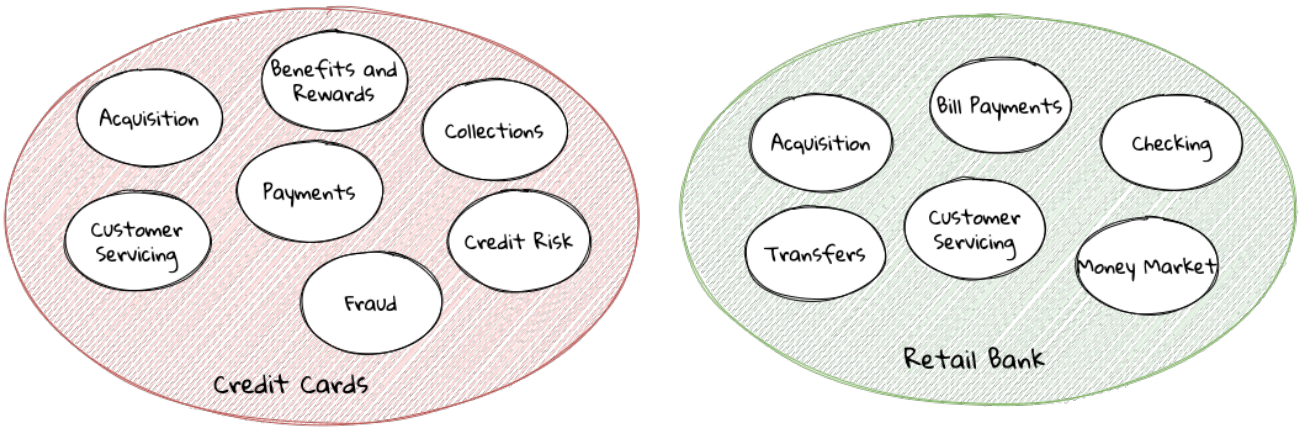


Figure 8. Banking subdomains at Acme bank

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

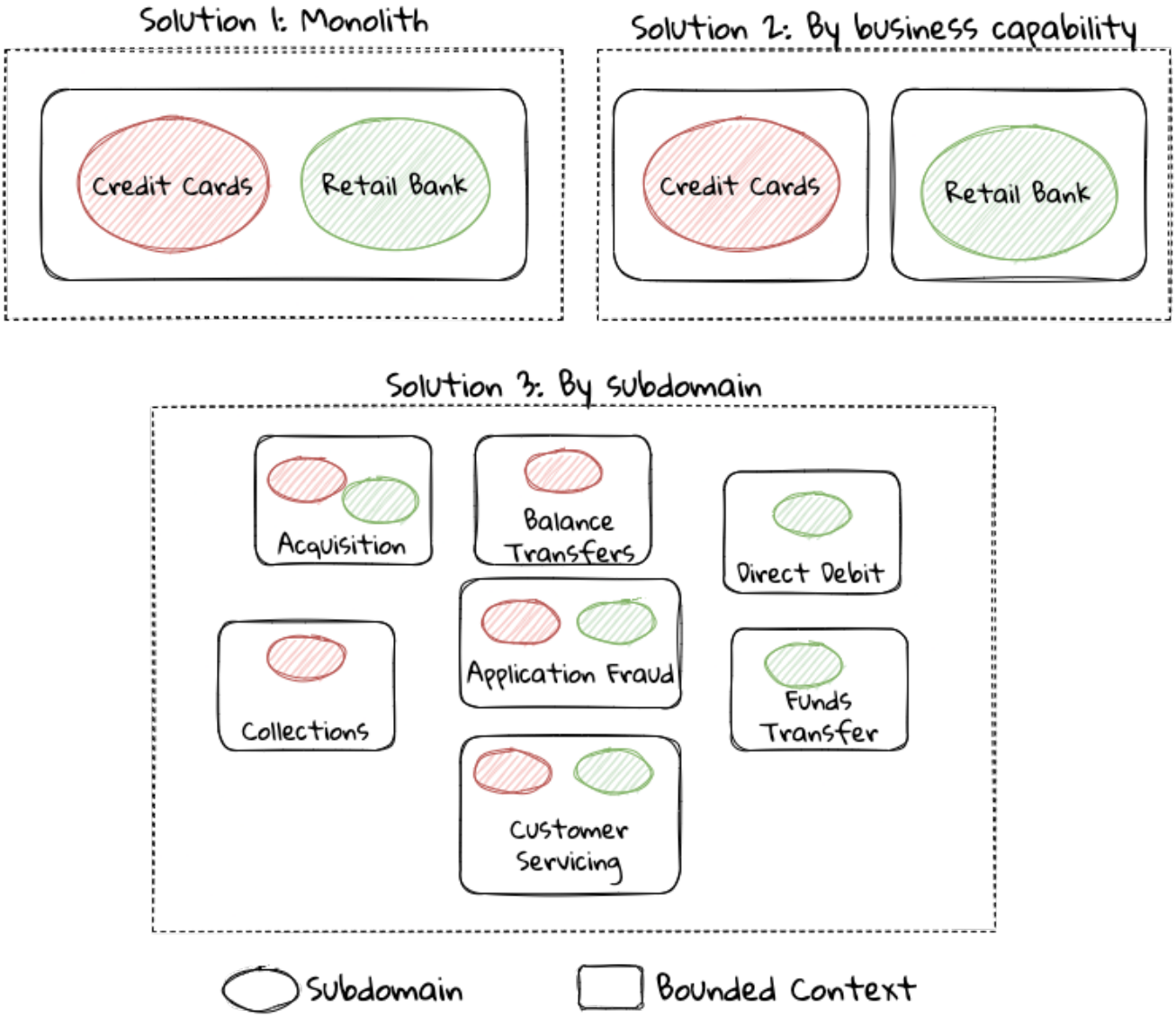


Figure 9. Bounded contexts options at Acme bank

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

Conway's Law asserts that organizations are constrained to produce application designs which are copies of their communication structures. Your current organizational structures may not be optimally aligned to your desired solution approach. The **inverse Conway maneuver**[1] may be applied to achieve isomorphism with the business architecture.

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.

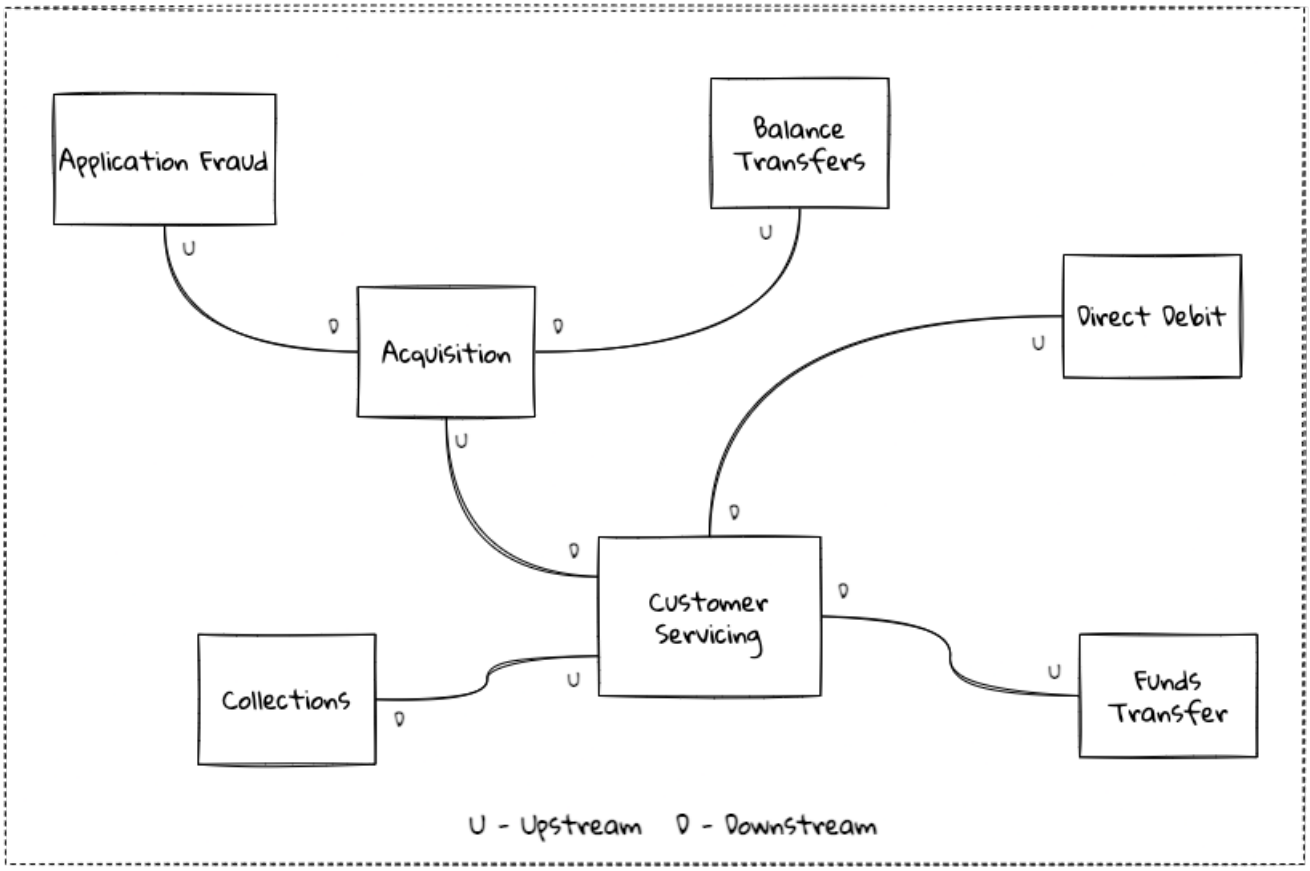


Figure 10. Sample context map for Acme bank

The context map shows the bounded contexts the relationship between them. These relationships can be a lot more nuanced than what is depicted here. 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 their 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.

Business model canvas

As we have mentioned several times, it is important to make sure that we are solving the right problem before attempting to solving it right. The business model canvas is a quick and easy way to establish that we are solving a valuable problem in a single visual that captures nine elements of your business namely:

- *Value propositions*: what do you do?
- *Key activities*: how do you do it?
- *Key resources*: what do you need?
- *Key partners*: who will help you?
- *Cost structure*: what will it cost?
- *Revenue streams*: how much will you make?
- *Customer segments*: who are you creating value for?
- *Customer relationships*: who do you interact with?
- *Channels*: How do you reach your customers?

Here is a sample canvas for a popular movie subscription provider:

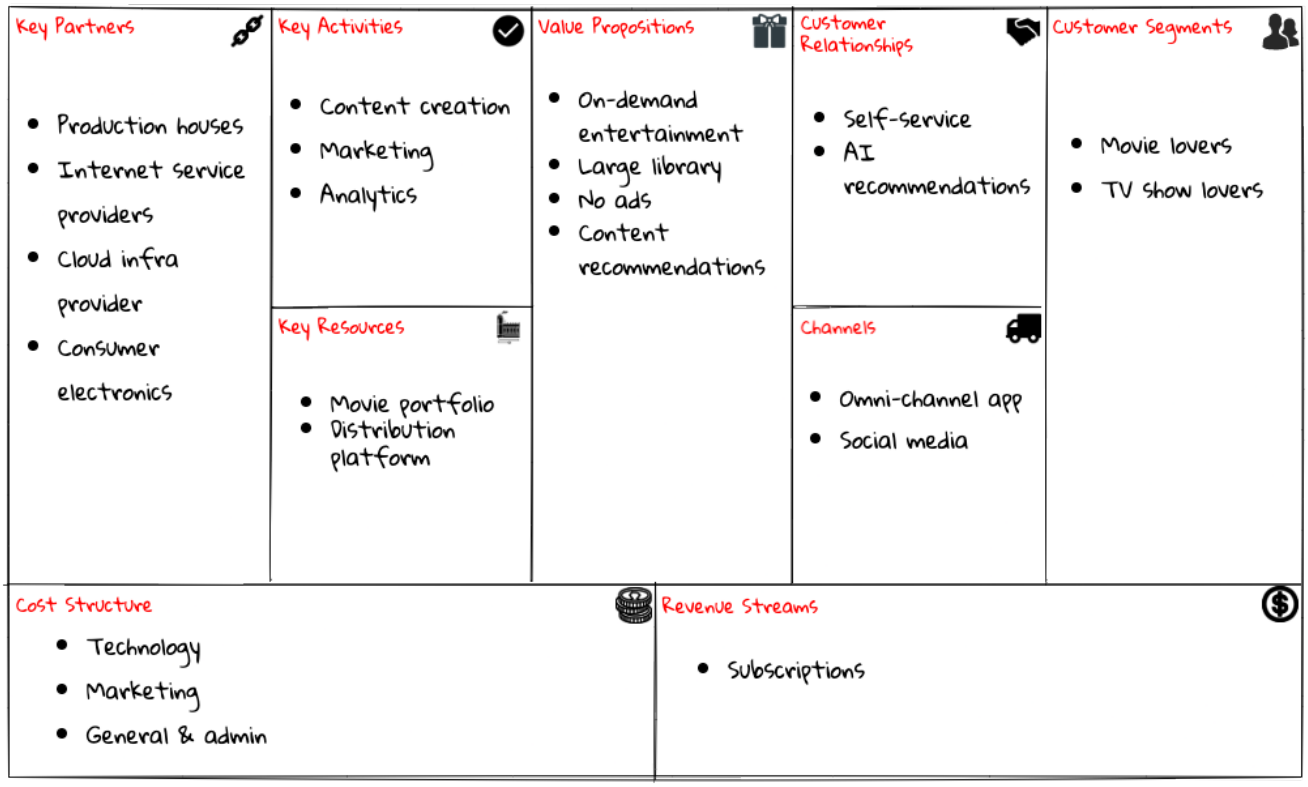


Figure 11. Business model canvas of a popular online movie subscription provider

The business model canvas helps establish a shared understanding of the big picture among a varied set of groups including business stakeholders, domain experts, product owners, architects and developers. We have found it very useful when embarking on both greenfield and brownfield engagements alike.

A variation of the business model canvas is the *lean canvas*, which is 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.

Wardley maps

The business model canvas can help establish clarity of purpose at a high level. The Wardley map is another tool to help build a business strategy. It provides a sketch of the people that the system is built for, followed by the benefits the system offers them and a chain of needs required to provide those benefits (called the *value chain*). Next the value chain is plotted along an evolution axis which ranges from something that is uncharted and uncertain to something that is highly standardized.

Here is a sample Wardley map for a bank that is looking to provide a suite of next generation credit card products:

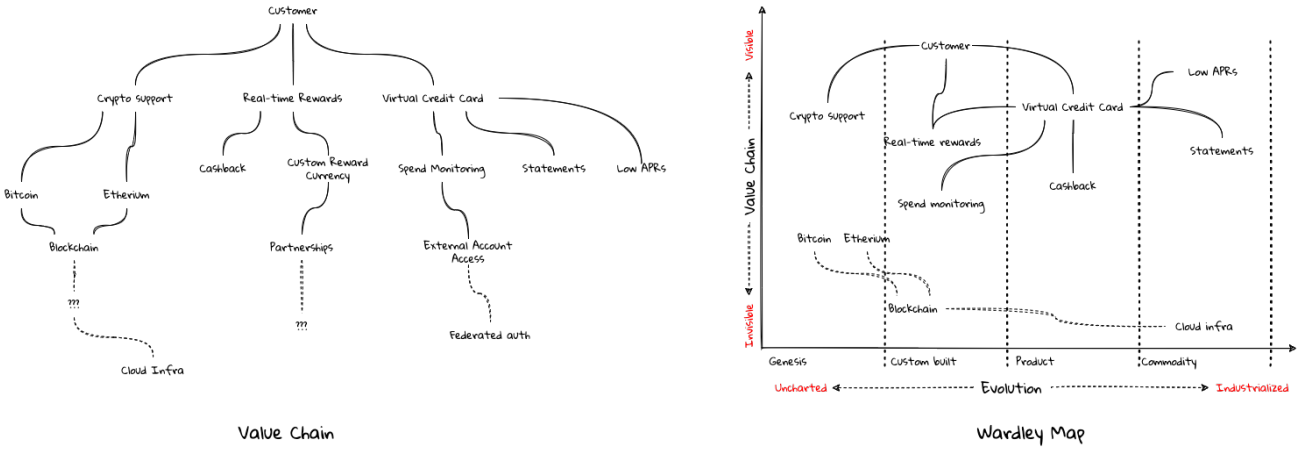


Figure 12. Value chain to a Wardley map

The Wardley map makes it easy to understand the capabilities provided by our solution, their dependencies and how value is derived. It also helps depict how these capabilities play out in comparison to those offered by competitors, allowing you to prioritize attention appropriately and make build versus buy decisions.

Impact maps

An impact map is a visualisation of scope and underlying assumptions, created collaboratively by senior technical and business people. It is a mind-map grown during a discussion facilitated by considering the following four aspects:

- *Goals*: **Why** are we doing this?
- *Actors*: **Who** are the consumers or users of our product?. In other words, who will be impacted by it.
- *Impacts*: **How** can the consumers' change in behavior help achieve our goals? In other words, the impacts that we're trying to create.
- *Deliverables*: **What** we can do, as an organisation or a delivery team, to support the required impacts? In other words, the software features or process changes required to be realized as

part of the solution.

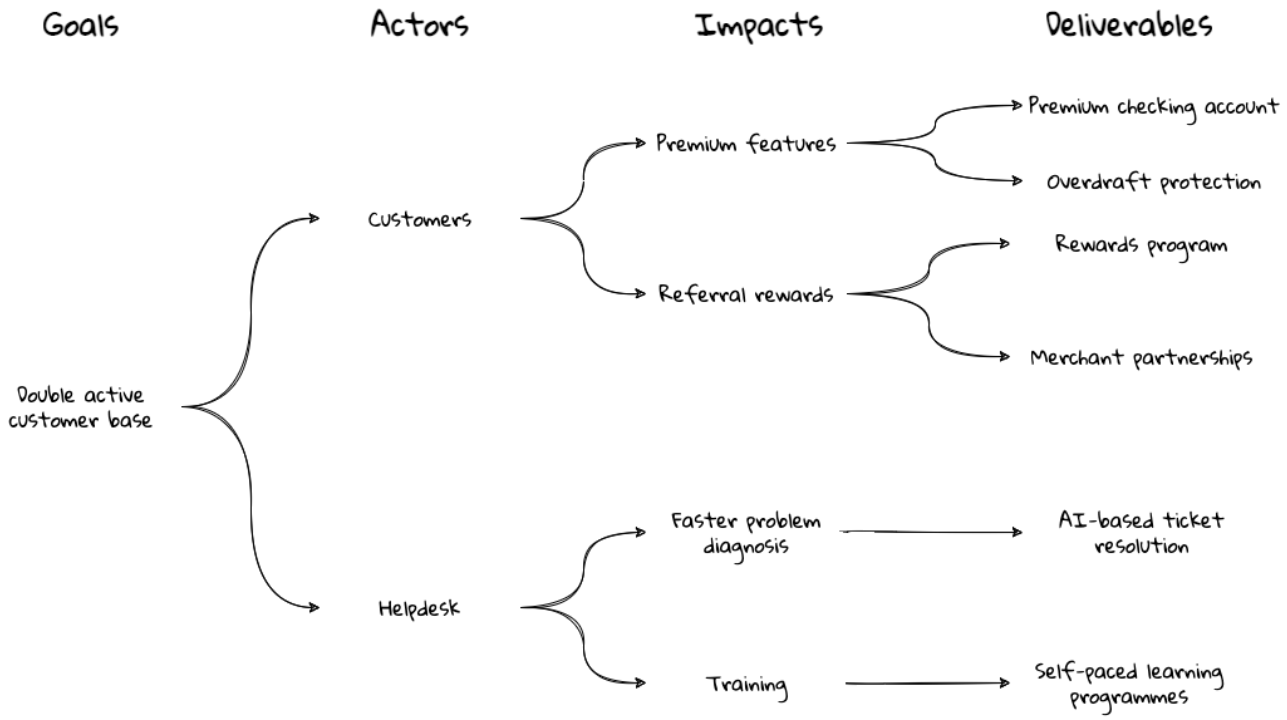


Figure 13. A simple impact map for a retail bank

Impact mapping provides an easy to understand visual representation of the relationship between the goals, the users and the impacts to the deliverables.

Implementing the solution using tactical design

In the previous section, we have seen how we can arrive at a shared understanding of the problem using the strategic design tools. We need to use this understanding to create a solution. DDD's tactical design aspects, tools and techniques help translate this understanding into working software. Let's look at these aspects in detail. In part 2 of the book, we will apply these to solve a real-world problem.

It is convenient to think of the tactical design aspects as depicted in this picture:

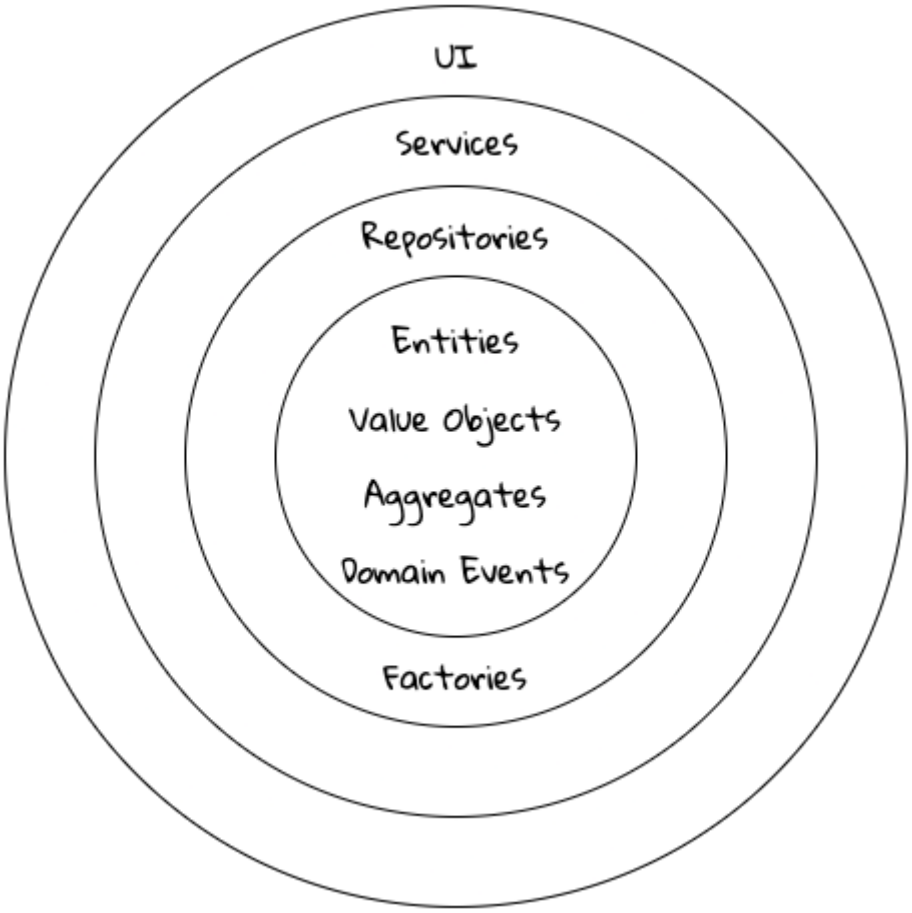


Figure 14. The elements of DDD's tactical design

Let's look at the definitions of these elements.

Value objects

Value objects are immutable objects that encapsulate the data and behavior of one or more related attributes. It may be convenient to think of value objects as named primitives. For example, consider a `MonetaryAmount` value object. A simple implementation can contain two attributes — an *amount* and a *currency code*. This allows encapsulation of behavior such as adding two `MonetaryAmount` objects safely as shown here:

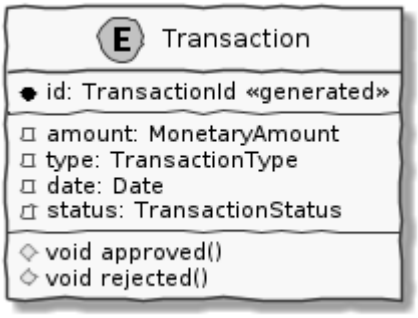


Figure 15. A simple `MonetaryAmount` value object

The effective use of value objects helps protect from the primitive obsession[2] antipattern, while increasing clarity. It also allows composing higher level abstractions using one or more value objects. It is important to note that value objects do not have the notion of identity. That is, two value having the same value are treated equal. So two `MonetaryAmount` objects having the same

`amount` and `currency code` will be considered equal. Also, it is important to make value objects immutable. That is, a need to change any of the attributes should result in the creation of a new attribute.

It is easy to dismiss the use of value objects as a mere engineering technique, but the consequences of (not) using them can be far-reaching. In the `MonetaryAmount` example above, it is possible for the `amount` and `currency code` to exist as independent attributes. However, the use of the `MonetaryAmount` enforces the notion of the *ubiquitous language*. Hence, we recommend the use of value objects as a default instead of using primitives.

Critics may be quick to point out problems such as class explosion and performance issues. But in our experience, the benefits usually outweigh the costs. But it may be necessary to re-examine this approach if problems occur.

Entities

An entity is an object with a **unique identity** and **encapsulates** the data and behaviour of its attributes. It may be convenient to view entities as a collection of other entities and value objects that need to be grouped together. A very simple example of an entity is shown here:

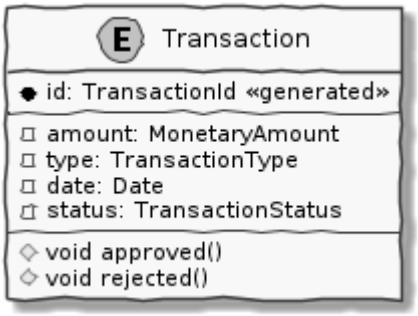


Figure 16. A simple depiction of `Transaction` entity

In contrast to a value object, entities have the notion of a unique identifier. This means that two `Transaction` entities having the same underlying values, but having a different identifier (`id`) value, will be considered different. On the other hand, two entity instances having the same value for the identifier are considered equal. Furthermore, unlike value objects, entities are mutable. That is, their attributes can and will change over time.

The concept of value objects and entities depends on the context within which they are used. In an order management system, the `Address` may be implemented as a value object in the *E-Commerce* bounded context, whereas it may be needed to be implemented as an entity in the *Order Fulfillment* bounded context.

It is common to collectively refer to entities and value objects as *domain objects*.

Aggregates

As seen above, entities are hierarchical, in that they can be composed of one more children. Fundamentally, an aggregate:

- Is an entity usually composed of other child entities and value objects.
- Encapsulates access to child entities by exposing behavior (usually referred to as *commands*).
- Is a boundary that is used to enforce business invariants (rules) in a consistent manner.
- Is an entry point to get things done within a bounded context.

Consider the example of a `CheckingAccount` aggregate:

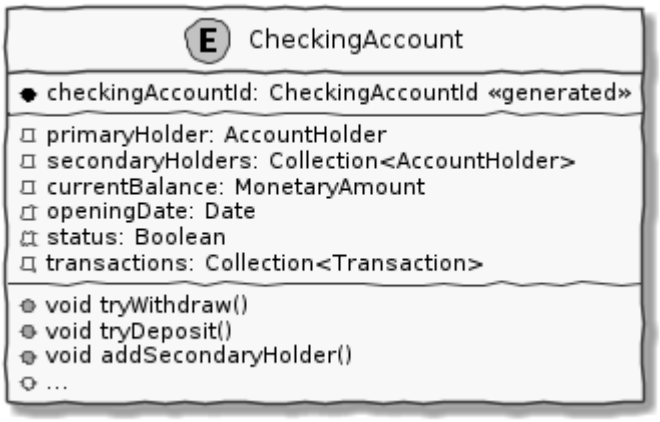


Figure 17. A simple depiction of a `CheckingAccount` aggregate

Note how the `CheckingAccount` is composed using the `AccountHolder` and Transaction` entities among other things. In this example, let's assume that the overdraft feature (ability to hold a negative account balance) is only available for high net-worth individuals (HNI). Any attempt to change the `currentBalance` needs to occur in the form of a unique `Transaction` for audit purposes — irrespective of its outcome. For this reason, the `CheckingAccount` aggregate makes use of the `Transaction` entity. Although the `Transaction` has `approve` and `reject` methods as part of its interface, only the aggregate has access to these methods. In this way, the aggregate enforces the business invariant while maintaining high levels of encapsulation. A potential implementation of the `tryWithdraw` method is shown here:

```
class CheckingAccount {
    private AccountHolder primaryHolder;                          ①
private Collection<Transaction> transactions;                     ①
    private MonetaryAmount currentBalance;                        ①
    // Other code omitted for brevity

    void tryWithdraw(MonetaryAmount amount) {                     ②
        MonetaryAmount newBalance = this.currentBalance.subtract(amount);
        Transaction transaction = add(Transaction.withdrawal(this.id, amount));
        if (primaryHolder.isNotHNI() && newBalance.isOverdrawn()) { ③
        transaction.rejected();
        } else {
            transaction.approved();
            currentBalance = newBalance;
        }
    }
}
```

① The `CheckingAccount` aggregate is composed of child entities and value objects.

② The `tryWithdraw` method acts as a consistency boundary for the operation. Irrespective of the outcome (approved or rejected), the system will remain in a consistent state. In other words, the `currentBalance` can change only within the confines of the `CheckingAccount` aggregate.

③ The aggregate enforces the appropriate business invariant (rule) to allow overdrafts only for HNIs.

Aggregates are also referred to as **aggregate roots**. That is, the object that is at the root of the entity hierarchy. We use these terms synonymously in this book.

Domain events

As mentioned above, aggregates dictate how and when state changes occur. Other parts of the system may be interested in knowing about the occurrence of changes that are significant to the business. For example, an order being placed or a payment being received, etc. *Domain events* are the means to convey that something business significant has occurred. It is important to differentiate between system events and domain events. For example, in the context of a retail bank, a *row was saved* in the database, or a *server ran out of disk space*, etc. may classify as system events, whereas a *deposit was made* to a checking account, *fraudulent activity was detected* on a transaction, etc. In other words, domain events are things that **domain experts care about**.

It may be prudent to make use of domain events to reduce the amount of coupling between bounded contexts, making it a critical building block of domain-driven design.

Repositories

Most businesses require durability of data. For this reason, aggregate state needs to be persisted and retrieved when needed. Repositories are objects that enable persisting and loading *aggregate* instances. This is well documented in Martin Fowler's *Patterns of Enterprise Application Architecture* book as part of the repository[3] pattern. It is pertinent to note that we are referring to aggregate repositories here, not just any entity repository. The singular purpose of this repository is to load a **single instance** of an aggregate using its identifier. It is important to note that this repository does not support finding aggregate instances using any other means. This is because, business operations happen as part of manipulating a single instance of the aggregate within its bounded context.

Factories

In order to work with aggregates and value objects, instances of these need to be constructed. In simple cases, it might suffice to use a constructor to do so. However, aggregate and value object instances can become quite complex depending on amount the state they encapsulate. In such cases, it may be prudent to consider delegating object construction responsibilities to a *factory* external to the aggregate/value object. We make use of the static factory method, builder, and dependency injection quite commonly in our day-to-day. Joshua Bloch discusses several variations of this pattern in *Chapter 2: Creating and destroying objects* in his *Effective Java* book.

Services

When working within the confines of a single bounded context, the public interface (commands) of the aggregate provides a natural API. However, more complex business operations may require interacting with multiple bounded contexts and aggregates. In other words, we may find ourselves in situations where certain business operations do not fit naturally with any single aggregate. Even if interactions are limited to a single bounded context, there may be a need to expose that functionality in an implementation-neutral manner. In such cases, one may consider the use of objects termed as *services*. Services come in at least 3 flavors:

1. **Domain services**: To enable coordinating operations among more than one aggregate. For example, transferring money between two checking accounts at a retail bank.
2. **Infrastructure services**: To enable interactions with a utility that is not core to the business. For example, logging, sending emails, etc. at the retail bank.
3. **Application services**: Enable coordination between domain services, infrastructure services and other application services. For example, sending email notifications after a successful inter-account money transfer.

Services can also be stateful or stateless. It is best to allow aggregates to manage state making use of repositories, while allowing services to coordinate and/or orchestrate business flows. In complex cases, there may be a need to manage the state of the flow itself. We will look at more concrete examples in part 2 of this book.

It may become tempting to implement business logic almost exclusively using services — inadvertently leading to the anemic domain model[4] anti-pattern. It is worthwhile striving to encapsulate business logic within the confines of aggregates as a default.

Tactical design tools

As we have mentioned a few times, DDD is about making sure that we build the right thing and then build it right. While strategic design tools help on the former, tactical design tools help on the latter. Let us look at set of tools and techniques we have found to be useful.

BDD and TDD

Test-Driven Development (TDD) was conceived by Kent Beck as a means to encourage simple designs and inspire confidence by writing a **test before writing the production code** required to satisfy that requirement. The intent behind this was to arrive at an optimal design iteratively, guided by a set of executable test cases. Unfortunately, it got misconstrued as a unit-testing technique as opposed to a design technique, leading to it being employed in a manner where teams were unable to derive the benefits of the practice. Behavior-Driven Development (BDD) was conceived by Chris Matts and Dan North as a means to practice TDD the right way by using better terminology (for example, using *specifications* instead of *test suites*, *scenarios* instead of *tests*, *confirming behavior* instead of *testing code*, etc). Consider the example of the overdraft facility for premium customers. A potential test for this feature using **JUnit** may look something like this:

```
class CheckingAccountOverdraftTests {

    @Test
    public void shouldAllowOverdraftForPremiumCustomers() {
        Account account = Account.checking("ABC123")
                              .withCustomer(premiumCustomer())
                              .withBalance(Money.usd(100));

        account.withdraw(Money.usd(150));

        assertThat(account.balance()).isEqualTo(Money.usd(-50));
    }
}
```

The BDD specification for this same feature using **Cucumber** and **Gherkin** looks something like this:

```
Feature: Overdraft capability for premium customers

Scenario: Should allow overdraft facility for premium customers.

Given I am a premium customer with account number "ABC123"
  And I have 100 dollars in my account
 When I attempt to withdraw 150 dollars
 Then I should have a negative balance of 50 dollars
```

Conceptually, there is no difference between the two versions, although it is arguable that less technical domain experts will likely prefer the BDD specification over the TDD test case because it uses the language of the problem domain in an implementation-neutral manner. This has led to BDD being used for more coarse-grained acceptance testing, while TDD gets used for more fine-grained unit testing. In our experience, both these techniques complement each other in achieving better software design. Because BDD tools (like Cucumber[5], JBehave[6]) allow the use of implementation-neutral specification languages like Gherkin, they tend to be more approachable for less technical stakeholders. However, a plethora of tools like easyb[7], Mockito[8], AssertJ[9] etc. make it fairly natural to adopt the BDD style even in Java, proving that they are very complementary as shown here:

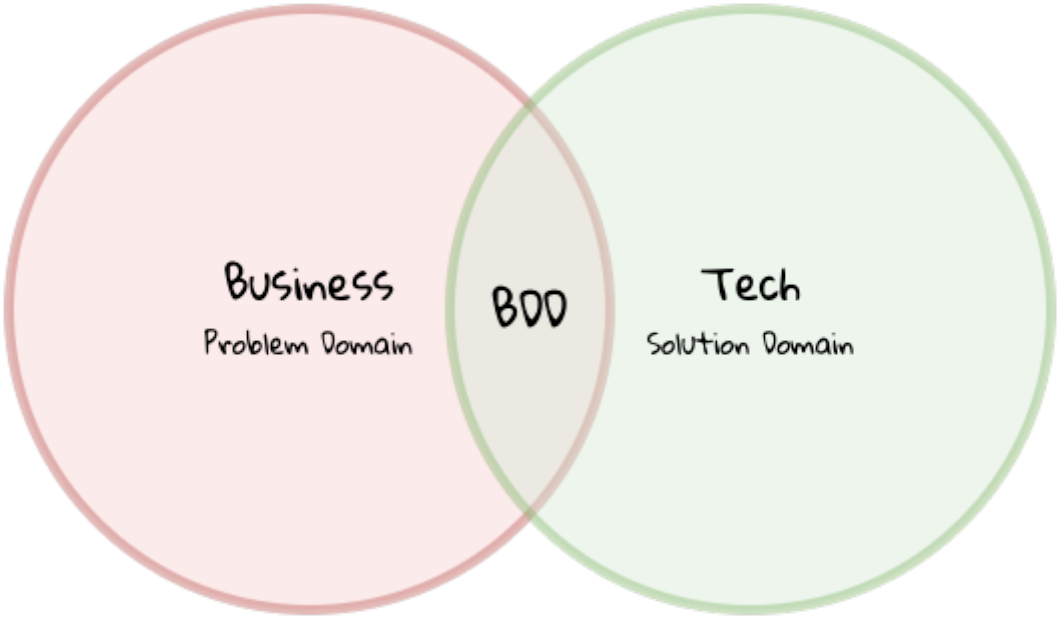


Figure 18. TDD and BDD are complementary concepts.

Both BDD and TDD, when used in close conjunction with DDD's ubiquitous language, can promote closer synergy between the problem and solution domain and in our experience is an indispensable tool when building software solutions.

Contract testing

When implementing a sound test strategy, one encounters two broad classes of tests: **unit** and **end-to-end** (and everything in between). Unit tests tend to be fairly fine-grained, especially when they mimic the behavior of collaborating components using mocks/stubs. This allows us to run a large suite of such tests while consuming very little time. In an ideal world, we would prefer restricting ourselves to running just unit tests. However, unit tests do have a limitation in that the assumptions made while mocking/stubbing collaborator behavior may become inaccurate when the owners of these components make changes to their respective solutions. This may lead to a situation where unit tests work just fine, but the solution as a whole does not work in a formal end-to-end environment, resulting in non-technical stakeholders losing faith in these tests.

To restore confidence, teams then resort to verifying functionality by mostly running time-consuming, error-prone manual end-to-end tests, occasionally throwing in some automation. In our experience, running a stable, non-trivial suite of automated end-to-end tests remains quite a challenge, due to the computational and cognitive expense involved in setup and ongoing upkeep. Because these tests require large portions of the real solution stack to be in place, they tend to often happen very close to the end, causing them to be rushed and coarse-grained.

What we need are tests that both run rapidly, are easy to maintain (like unit tests), and provide a high degree of confidence that the system is functionally correct when they pass (like end-to-end tests). Contract tests can act as the missing link between unit and end-to-end tests by affording a set of *blessed* mocks (those that are always compatible with the real implementation). We will cover contract testing in more detail in Chapter 10.

In addition to the tools and techniques discussed above, we would like to call out domain story telling[10] and eventstorming[11] as two valuable techniques that cover aspects of both strategic and

tactical design. We cover both these techniques with more concrete examples in Chapter 4.

Summary

In this chapter we looked at the differences in perspective that arise due to problem domain and solution domain thinking. We examined how to arrive at a shared understanding of the problem domain using DDD's strategic design elements. We introduced a few tools and techniques that can aid in accelerating the strategic design process.

We also looked at how to build robust solutions using DDD's tactical design elements along with tools and techniques that can enhance our journey of building solutions that can evolve durably.

In the next chapter we will take a closer look at where DDD fits in the scheme of various architecture approaches, patterns and how it is applicable in a wide array of scenarios.

Questions

1. Can you draw a business model canvas for the current ecosystem you are working with? Did this exercise help in enhancing your understanding of the big picture?
2. Are you able to identify the different subdomains in your problem domain? Do your solutions (bounded context) align with these subdomains?
3. Are you able to draw a simple context map of your current ecosystem?
4. Do your bounded contexts align along specific aggregate roots?

[1] https://www.thoughtworks.com/en-us/radar/techniques/inverse-conway-maneuver

[2] https://wiki.c2.com/?PrimitiveObsession

[3] https://martinfowler.com/eaaCatalog/repository.html

[4] https://martinfowler.com/bliki/AnemicDomainModel.html

[5] https://cucumber.io/docs/installation/java/

[6] https://jbehave.org/

[7] https://easyb.io/

[8] https://site.mockito.org

[9] https://joel-costigliola.github.io/assertj/

[10] https://domainstorytelling.org/

[11] https://www.eventstorming.com/