



# MASTERING STRATEGIC DDD

Step-by-step navigation through a  
real-world example

Maciej 'MJ' Jedrzejewski

# Author

---



Maciej "MJ" Jedrzejewski

Fractional architect, IT consultant, freelancer, and trainer. Adherent to a pragmatic and holistic approach to software architecture. I am here to help you avoid the unnecessary costs of over-engineering while maintaining quality standards. And all this without needing a permanent architect - I act as a resource-as-a-service solution - both from cooperation and workshops perspective.

You can find me on [Linkedin](#) or subscribe to a [YouTube channel](#) related to software architecture.

# From author

---

Computers and programming have been next to me for as long as I can remember. Fixing bugs, overclocking processors, and staying up late to solve another problem. Apart from the processors, nothing has changed in so many years. IT is not just my job. It is my life, my passion.

Throughout my career, I have worked with and spoken to many companies. Some of them were corporate and medium-sized, and others were start-ups. On the one hand, the problems they faced were completely different, but on the other hand, it was mainly about software architecture.

Poor application performance, long testing before production, high maintenance costs, and expensive feature enhancements.

This book is actually addressed to you to reduce the risk of the problems mentioned. Strategic Domain-Driven Design helps us focus on the importance of business processes and the domain itself. It all starts there - even performance problems or long (and expensive!) testing.

I aim to give you as many practical tips as possible based on real-world scenarios and show you how to approach building the application. Thanks to this book, I hope you can avoid the mistakes I have made myself.

Enjoy!

# Content

---

1. The Beginning	4
2. The Domain	11
3. Draft of Subdomains	18
4. Tuning of Subdomains	30
5. Bounded Contexts	37
6. Context Map	45
7. Next Steps	55

# 1. The beginning

---

Years ago, I wanted to start learning Domain-Driven Design. It was quite a popular topic, and many of my colleagues were discussing it. The problem, as always, was where to start.

I decided to ask one of my old mentors. He told me, "MJ, there is a great book about DDD called the Blue Book. Go read it, and everything will be clear. There is nothing better on the market".

"Well, it must be super easy. I can read it in the next two months and will be a master in this subject". - I thought.

I bought it. Eight years later, I can tell you it is a priceless book. But - there is always "a but" - it took me 8! years of practice and theory to get where I am now - and I am still learning. Thousands of bad decisions in projects I have been involved in, screwed-up concepts, and over-engineered architecture - this is all I have done.

It was a fantastic time, but I had to learn everything from my mistakes (and my team's).

That is why you are here. I want to help you, show you the way, and convince you to avoid repeating my mistakes. Only try to apply some things from books, tutorials, and other materials. Strategic Domain-Driven Design is not that complicated - it is not quantum physics, where our minds must step out of the framework imposed on them from childhood. But (again!) it does require you to step out of your comfort zone and stop thinking in terms

of technicalities - providers, controllers, managers, and handlers. **The most important thing is business.**

And that is where this story begins.

Much of the material you can find online and in various books is often scattered. It isn't easy to find the knowledge in the form of a recipe - when, what, and how to apply it.

When building IT systems, there is **no such thing as a silver bullet**, a solution that will always work, no matter what environment you are in or the system's state. However, I will give you a step-by-step guide showing you the path that worked for most projects I was involved in. In the end, you will have to decide which of these steps you want to repeat, extend, and which you want to skip - it is your choice.

All I give you is my knowledge and experience.

## **Understand your domain first**

Each company has its **business domain** - sometimes there are two, sometimes dozens, sometimes hundreds - in which it operates and earns money.

Joining a new company or project is often an extremely stressful time - you must prove you are the right person for the job or have many decisions to make. It is not uncommon for you to feel like you are on a rollercoaster filled with architecture diagrams, books, or tons of

requirements, new teammates, organizational things, and so on. Even if you are experienced, you may need help with all the information coming your way.

Because of the above, you may not have time to go deep into the business domain in which you operate. You may be told that all the requirements are there, they are clear, and **we need more time** to focus on the business analysis.

That's where most problems and legacy systems start.

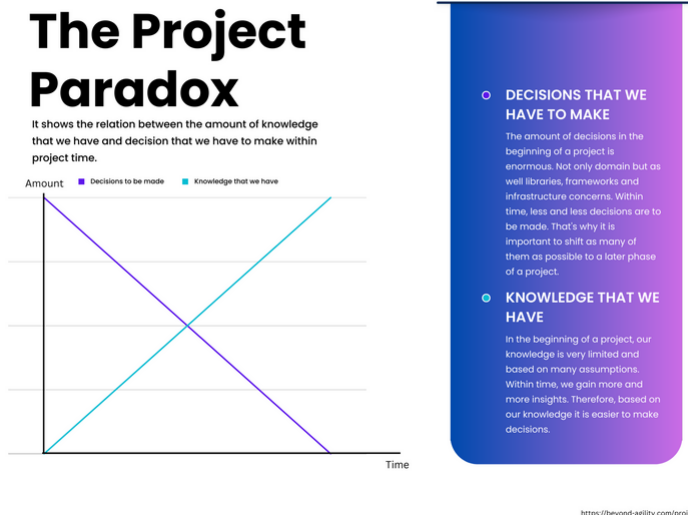
It is crucial that you communicate clearly and loudly that there is a need to go through the domain and its processes again - you are new, and you will only understand a few of them just by reading tons of documents. You will need to organize some workshops - usually, you will need at least a few days to understand simple businesses and up to several weeks to analyze larger ones.

Often, it is challenging to convince people to do this when trust has yet to be established. So you usually only get one chance. And this one chance has to become a success - if your workshop is full of quality, people are involved, and - ideally - it needs much clarification, the stage is set for future workshops.

## The Project Paradox

You must understand the domain well to ensure you succeed in the long run. At the beginning of any project - when our knowledge is minimal - we have to make a lot of

different decisions. Then, over time, our knowledge grows, but the number of decisions decreases:



Pic. 1 - Project paradox - lots of decisions to be made when we know nothing about the domain

Due to the lack of knowledge and the fact that we are exposed to many decisions, we can expect many of them to be wrong.

Why not spend more time initially to understand what and why we want to build something and improve the quality of our initial decisions? Ultimately, we will create a safer environment around us and avoid many problems that would have arisen had we skipped the initial steps.



# Organize workshops

The time has come - you must organize your first workshop. The most important thing is to set yourself a goal, a result that you can work on and build on.

In most cases, I want to discover and describe the most critical business process. The one for which the system will be mainly built - fully or partially, sometimes parts of the business processes happen outside the system - an example can be a process where, in one step, the request is sent to the national legal system. We must wait a couple of weeks for the answer before continuing the process in our company.

For this opener - the first workshop - you will ideally have decision-makers - people who can make such decisions - and domain experts - people who have much knowledge about different parts of the process, e.g., logisticians, accountants, sales, lawyers, etc. If there is an existing legacy system, these may be the software developers who have worked with the system the most, as they usually know much about the business.

When everything is defined:

- **participants** - my preference is not to have more than eight because I cannot give my full attention to each person
- **place** - on-site or remote - I prefer the on-site option as I have more interaction with the participants, and people tend to be more active (based on my experience)

- **time** - how much time do we need for the workshop? I usually plan 3 \* 1.5 hours spots for two days for the first one. If I see that we need more time (which happens quite often), then I plan for follow-ups

You can start thinking about methods and techniques you want to use. There are different ways of doing it:

- Event Storming
- Domain Storytelling
- User Story Mapping
- Story Storming
- Business Model Canvas
- Wardley Maps

and many other alternatives or combinations of the above. I usually opt for one of the first two - Event Storming or Domain Storytelling - and use them with Wardley Maps and Business Model Canvas.

## Summary

This chapter introduced you to the world of strategic thinking. You might not have seen anything about subdomains, their types, bounded contexts, domain and architecture patterns, or context maps.

If you are surprised, that's exactly what I wanted to hear. Before jumping into the deep end, we must go through a particular domain's business analysis process. Otherwise, we rely on theory, which I want to avoid.

In the next chapter, we will work on a business domain called **Fitness Studio**. There, we will discover business processes using one of the mentioned techniques - **Event Storming**. Then, when our processes are on the whiteboard, we will try to distill subdomains - finally, something directly related to Domain-Driven Design - based on concrete heuristics.

Buckle up, and let us get to work on our domain!

## 2. The domain

---

Every time we build software, we operate in an area called a domain. It is something that describes our business. It can be:

- **E-commerce** - this domain can include online shopping, product catalog, order processing, payments, and many other smaller parts
- **Taxes** - in this domain, we can find areas like income tax, corporate tax, tax compliance, or tax policy
- **Healthcare** can include patient management, appointment scheduling, medical records, prescription management, etc.
- **Education** - assignment tracking, learning platforms, student enrolment, grading, etc.
- **Logistics** - things like warehousing, transportation, inventory management, route optimization, or delivery schedule

As you can see, it all depends on your operating sector. Often, businesses operating in the same industry are similar - that is, they are represented by standard, basic business models or types that represent core aspects of any business or industry. Such representations are called archetypes. This does not mean, however, that all these parts work precisely the same.

And that's something we must address in our business domain analysis every time we build - or, in the case of legacy apps, rebuild - the system.

# Our business domain - Fitness Studio

We are hired to build an application responsible for handling all possible processes you can imagine when you think about a fitness studio.

Of course, it is impossible to close the door, implement every possible business process we can think of, and release our application in 2 years (**such an approach usually ends in disaster** - quite often, we run out of money or no one wants to use it). Instead, we should consider the MVP that would be valuable for our customers and cover the minimum critical processes.

Ok, great. But... how do we find these processes? Surely, techniques like Event Storming or Domain Storytelling will help. We gather different people for a series of workshops, and there, we brainstorm ideas and think about the flow.

Yes, this is a step you will need sooner or later. But in practice, it all starts somewhere else. It begins with some ideas presented to you before you even think of planning any workshops. Usually, you are invited to a meeting or a conversation where the idea is offered to you in a form similar to the one below:

*We want to build a SaaS application that any fitness studio can use. It will cover everything you can think of, from the initial offer to diet planning.*

*We want to prepare and promote offers to our customers. The contract should be signed if someone is interested in joining or renewing their membership.*

*Once the contract has been signed, the client will be issued a pass allowing access to the studio.*

*Of course, when the existing pass expires, we need to be able to offer our customers a new contract.*

*It also needs to be able to generate some reports; at the beginning, it may just be the number of passes sold.*

When you hear all this, you might feel overwhelmed by the information. But I have to tell you that you are fortunate - I have personally had experiences where the development team was told to do the same as application XYZ, which was the entire description of what we had to build. It is called **pathology**.

OK. Why am I so happy when I get much information that scratches the surface but not the details? Well, the time for details will come in the workshops that you will organize later.

Now let's look at what information you have received and how you can extract what you need for the first workshops:

*We want to build a SaaS application that any fitness studio can use. It will cover everything you can think of, from the initial offer to diet planning.*

The first important thing you can take away from this conversation is a high chance of building a **SaaS application** for fitness studios. This is technical but shows a business model - how you want to profit from it. Please write it down; you'll need to talk about this with your colleagues after the workshops.

Why? Quite often, there may be better solutions for the business, and you are there to start discussions if you think there are better ways of doing things.

Let's look at the next part of the conversation:

*We want to prepare and promote offers to our customers. The contract should be signed if someone is interested in joining or renewing their membership.*

We have the first requirements, which we will examine in more detail during our workshops. **Someone will prepare offers and advertise them to new and existing customers.** If a person decides to accept it, the contract will be prepared. This provides you with a framework so you can plan your workshops and define the objective on which you will focus. As you can probably see, different questions might come at that point:

- Will the offer be fully prepared in our application?
- How will it be promoted? Newsletter? Within the application?
- What if the person is interested but wants a discount?
- Is the contract signed within the application or at the fitness studio?

These questions should be answered in your sessions with domain experts and decision-makers. But it's great that we already have them. Write them down so you remember to mention them later.

Looking further ahead, we can see some other important information:

*Once the contract has been signed, the client will be issued a pass allowing access to the studio.*

If you look back, you can see that there were already four business processes, and the next one has just appeared:

- Offers preparation
- Offers promotion
- Contract preparation
- Contract signing
- **Pass registration**

The final step - pass registration - is the result of signing the contract and **triggers the receipt of a pass by the customer**. Again, the first questions that may come to mind are:

- How is the pass given to a customer?
- Is there only one way to do this or many, e.g., on-site, by email, or by post?

And so on. Again, these concerns should be addressed and used in future discussions.

We have analyzed almost the whole statement. Two parts need to be included. First, let's close the area of customer management:

*Of course, when the existing pass expires, we need to be able to offer our customers a new contract.*

Here's how it works! **When an existing pass expires, we send a new contract offer to the customer**. The following questions may be asked:



- What does it mean when the pass expires?
- Should we prepare an offer before the pass expires?
- Should the discount be included in the contract offer?
- Is there a case where the contract should be automatically renewed?

Look at how much information about core business processes we have extracted from such a short statement! And this is just the beginning because these are just your thoughts. You can imagine what will happen in brainstorming sessions with your colleagues.

The last part of the statement is not directly related to customers, but it can help fitness studios compare sales from month to month and prepare some goals to achieve in the coming months or year:

*It also needs to be able to generate some reports; at the beginning, it may just be the number of passes sold.*

We want to be able to **generate a report on the total number of passes sold**. Ok, another area we need to focus on. The decision on what kind of report to offer must be consulted with the fitness studios that will use our application. If not, you have a perfect question during workshops: *Why was it not consulted with people who will potentially use our application?*

**IMPORTANT:** The above steps are not required; you can start directly in your workshops, but they always helped me dig deeper into the topic before and be well prepared. Often, it is easy to forget to ask some questions, and this way, you increase the chance of remembering them.

## Summary

So, this is the time to focus on business analysis from a workshop perspective. In the next chapter, you will learn how to start your domain analysis with Event Storming and distill the first subdomains based on its outcome.

Let's start with the draft of subdomains!

# 3. Draft of Subdomains

---

After all of the previous steps, we are ready to continue our journey and try to prepare the first draft for splitting the **Fitness Studio domain** into several subdomains. But there is still a long way to go.

First, let's try to define what a subdomain is.

Every business operates in a business domain - we already know that. To keep it efficient, it is usually divided into different areas such as warehousing, deliveries, accounting, etc. In addition, there are many other processes within each business unit or - in larger companies - department. Through these processes, the business can serve its customers and generate revenue.

Let's look at an example business domain of an educational system. This includes financial management, human resources, facilities, student services, staff recruitment, and marketing. Let's assume that each of these can be treated as a subdomain.

Can human resources and staff recruitment be part of the same department? Yes, of course they can! Can part of student services be part of more than one department? Yes, it can. This is why **saying that a subdomain is equal to a department is wrong** and will bring you many problems in the future.

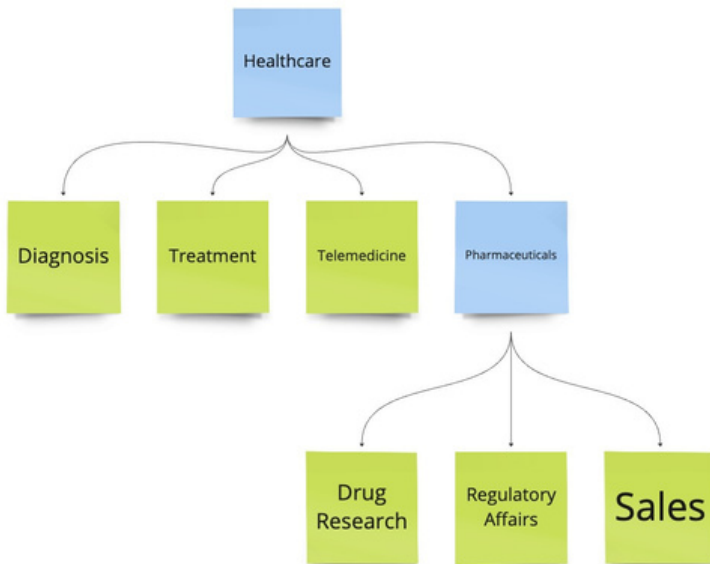
Let's consider another practical example to illustrate the concept of subdomains in DDD.

Imagine you are building an online marketplace similar to Amazon. Within this marketplace, there are several distinct areas of functionality, each serving a specific purpose:

- **Product Catalog:** it manages everything related to products: adding new products, updating product information, and categorizing products into different sections (e.g., electronics, clothing, books)
- **Order Management:** it handles activities related to customer orders: placing orders, tracking orders, etc.
- **Review and Rating:** here, users can leave reviews and ratings for products they have purchased, helping others make informed decisions

Each area represents a subdomain in the context of the online marketplace application. By defining these subdomains, developers can focus on the specific complexities and rules within each area without getting overwhelmed by the entire system (**divide and conquer**). This approach allows for more transparent communication among team members and ensures that each application part is developed and maintained effectively.

There is another crucial thing to remember. Sometimes, a subdomain can represent another domain and contain other subdomains - this can happen in large-scale organizations. So, when you are modeling a domain and running the workshops where you will break it down, you may find that one or more subdomains are too big to fit into one system, and you need another. To illustrate this, let's look at the image below:



Pic. 2 - Complex domain of a healthcare where a subdomain becomes a separate business domain

It shows a complex domain of healthcare services. Initially, we have split it into four subdomains. However, further analysis revealed complex processes inside the Pharmaceuticals subdomain that we must investigate further. This is a perfect example of how **a subdomain can become a domain**.

Ok, great. Now you know what a subdomain is. But how do you find these subdomains?

That's where modeling techniques come in. My two favorites are:

- Event Storming
- Domain Storytelling

But you might also use other techniques like User Story Mapping, Impact Mapping, Story Storming, or others. I want to focus on one method and show you the path from the beginning till the end of strategic DDD. Let's go with Event Storming.

I am going to scratch the surface of it without a detailed explanation. If you want to see how it works in detail, there is no better source than the [following resources](#).

Event Storming is divided into three levels:

- Big Picture
- Process
- Design

In this series, we will focus on the first two.

Each will be part of a different focus area. When discussing creating subdomains, we need to start at the big-picture level.

How do you organize such a workshop?

I invite people who are decision-makers and domain experts. Sometimes, these are business people, sometimes developers (who - in legacy systems - usually have the most knowledge of the business processes), or a mix of both.

I prefer sessions with less than ten people (including me as the facilitator).

**On-site or remote?** Both are possible.

If you go on-site, you will need cards - instead of sticky ones, get static ones - as you can quickly move them around. And a long wall to stick the cards on.

If you decide to run the workshop remotely, you can use software like Miro - IMO there is no better option for **remote Event Storming sessions**.

Try to split the workshop into several parts. I usually plan a short break after each hour.

There are 2 types of cards that we usually use during this kind of workshop.

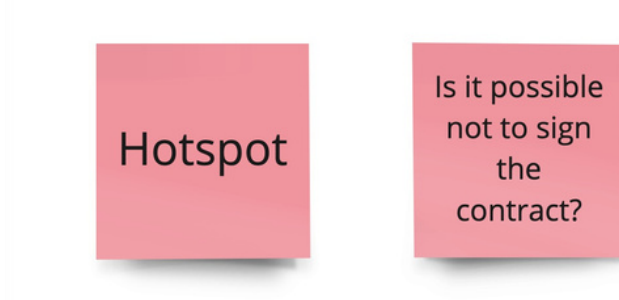
#### **Events:**



Pic. 3 - An example of an event

This primary type represents all the business events in our domain. The rule of thumb is to name it in the past tense (something happened) because it is an event that has already occurred and cannot be changed.

## Hot Spots:



Pic. 4 - An example of a hotspot

which are questions and inputs that cannot be answered during the workshop or are questionable - there is little knowledge about it - or which we want to shift.

It is time now to start the **workshop around our Fitness Studio** domain. Participants get as many orange notes as needed and begin sticking business events around. Then, there is no focus on any timeline and no good or wrong ideas. After some time, your wall will look similar to this:





Pic. 5 - First result of Big Picture Event Storming

As you can see, there are both:

- Events that are **related to business**, e.g. Contract signed or Pass expired
- Events that are **technical**, e.g. Request sent to Offer API or Button clicked

Remove the technical ones, as they are not necessary in the context of business and subdomain distillation.

**NOTE:** there might be duplicated events, so just throw away all duplicates.

After cleanup, our whiteboard looks like th whiteboard should look like this:is:



Pic. 6 - Board after removing technical events

The time has come to group the various events. Try to group it based on the following:

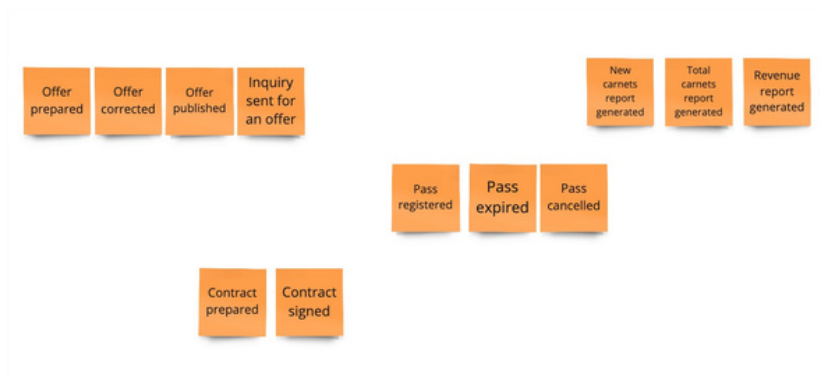
- **Similar wording**, e.g., Offer with Offer, Pass with Pass, etc. **NOTE:** Be careful - it is treacherous. Sometimes, Offer in one context can mean something else in another context
- **Timeline** - what has to happen before the next event occurs
- **Actors** - who is responsible for the execution of the process
- **End of process** - to execute a process, we need several events. If the next event does not fit the process, it might be an indicator that it can be a part of a separate subdomain

Such a grouping can be handled by drawing circles of one color or just by moving cards around:



Pic. 7 - Events grouped by colors

After marking it, we are ready to group it:



Pic. 8 - Events grouped by processes

In theory, everything is fine. But I made a mistake. One of the events does not match the process. Could you look at the diagram again and try to find it before you read on?

We want to publish the offer to our potential customers. Several steps are required:

- Someone has to prepare it
- Then it has to be reviewed and corrected
- When it is ready, it has to be published

Voila, our offer is published. So, what does **Inquiry sent for an offer** do here? This is the mistake.

I accidentally put it next to other offer events because it contained the **well-known word - Offer**. But there is another one - **Inquiry**, which has yet to appear before, and this event is the result of an action taken by a customer (another actor) in a completely different process, not by someone who prepared the offer (probably a salesman).

If this needs to be more convincing, another way to spot such a mistake is to focus on **pivotal events**.

## What is a pivotal event?

You can think of it as a "it is completely different!"; **a key event** in your process. An event that is most interesting to the majority of people. It usually ends a (sub)process. And there can be many pivotal events:

- ContractSigned
- PassRegistered
- OfferSent
- InvoiceDelivered
- PaymentReceived
- RiskCalculated

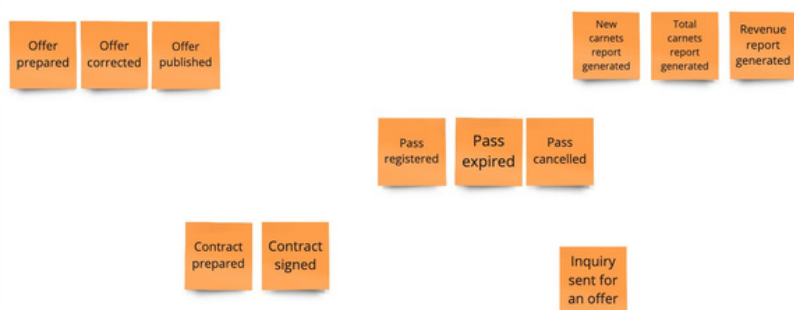
These are events that are essential to your business. They are also good candidates to set the boundaries of your subdomains.

Thanks to them, you can prepare the first draft of your subdomains. In this case, these could be:

- Contracts
- Passes
- Offers
- Invoices
- Payments
- RiskAssessment

subdomains. When you are ready, review them again, go through each process, and decide if there is a better candidate or if you can leave it as it is.

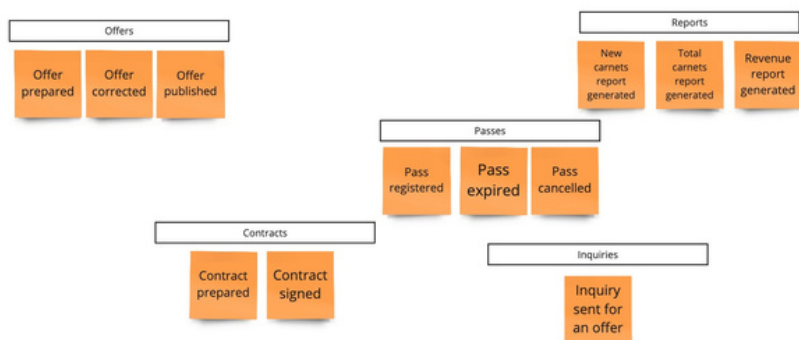
Let's fix our diagram:



Pic. 9 - Inquiry is moved from Offers to a separate process

Now, our events are grouped. They may change in the future - during the **Process Level Event Storming** or when your domain changes but it is an excellent draft.

Now it is time to name them - remember to do this together with everyone in the workshop:



Pic. 10 - Processes wrapped into subdomains

**IMPORTANT:** As you see on the diagram, reports are wrapped in a separate subdomain. Often, it makes sense to start this way and, with time, observe the behavior and changes done in specific subdomains. It might make sense to have reports per subdomain - reports in contracts, reports in passes boundaries, and so on.

Ok. We are done for now. After heavy workshops, we got it - we can continue to model the details of each process. Thanks to that, we can rework our draft division and select **types of subdomains - Core, Supporting, and Generic.**

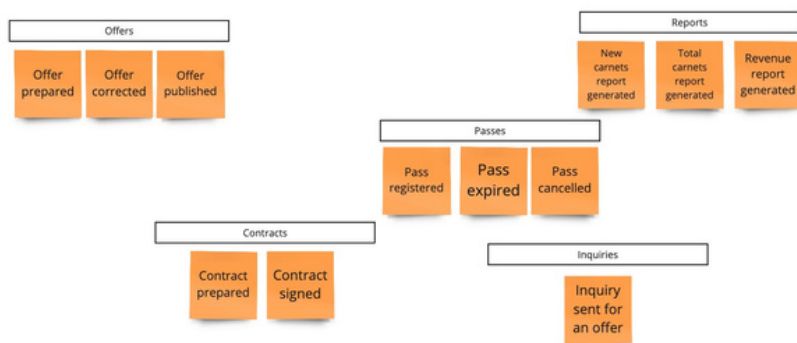
Let's get down to work and tune our subdomains!

## 4. Tuning of Subdomains

We have done much great work so far. We have analyzed our business domain from a big-picture perspective, recorded events, wrapped them into processes, and grouped them into subdomains. That's fine, but not enough to start coding.

It is time to move forward and tune our subdomains. How do we do this? The answer to this is **Process Level Event Storming**. This technique allows us to focus and extend concrete processes. Thanks to it, we may have discovered - and certainly will - that our previous work does not reflect reality. After all, the devil is often in the details, and a small detail can turn everything upside down.

Let's take a final look at the results of the Big Picture Event Storming workshop:

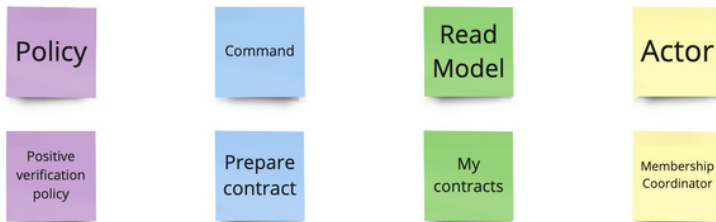


Pic. 11 - The view of drafted subdomains

It clearly states that we have at least five subdomains to work on. We will focus on two of them as the analysis of others will be a repeatable work - thanks to it, you can try to practice it on your own. The selection will cover:

- Offers
- Contracts

Before we start, I would like to introduce you to four new types of cards:



Pic. 12 - New types of cards that we will use during Process Level Event Storming

## Policy Card

- **Color:** Usually purple
- **Description:** Describes a decision rule or a business policy that dictates what action should be taken under certain circumstances. Policies are typically derived from business rules or legal requirements
- **Purpose:** Policies guide the decision-making process within the domain and often lead to the generation of commands based on certain conditions



## Command Card

- **Color:** Usually blue
- **Description:** Represents an action taken by a user or system which changes the state of the domain. It's usually a verb or a verb phrase, like *Sign Contract* or *Cancel Subscription*
- **Purpose:** Commands are crucial for understanding what actions can be performed in the system and what triggers these actions

## Read Model Card

- **Color:** Usually green
- **Description:** Represents the data that is read or viewed by users or systems. Read Models are the way the system exposes information after processing events
- **Purpose:** They help in understanding how data is presented to users and what information is available for making decisions or performing actions

## Actor Card

- **Color:** Usually small, yellow
- **Description:** Represents a person or role that interacts with the domain. Actors can be the initiators of commands or recipients of information
- **Purpose:** Actors help identify who or what is interacting with the system, providing insights into user roles or other entities that play a part in the domain

So this set of cards will allow us to show who interacts with the process, what data is given to the person

interacting with it, who triggers commands that lead to events, and so on.

After the Process Level Event Storming workshop, our diagram will look something like this:



Pic. 13 - Results of Process Level Event Storming

There are many more details compared to what we had before, especially in the Contracts subdomain.

During workshops, you will see that some of your subdomains are very simple. There are no business rules, and the processes are straightforward. This should tell you that it will probably be a very simple subdomain of either **Supporting** or **Generic type**. In our example, Offers will be one of these.

On the contrary, Contracts is a good candidate for a **Core** subdomain - there are 3 policies, based on verification, will one or the other action be handled. We discovered more processes during workshops than in the big picture.

There is *Customer Verification* and *Contract Rejection* that has not been present before.

Ok, let's take a step back and define all the types of subdomains mentioned:

## Core

- **Description:** The core subdomain is the central part of your business that differentiates it from competitors. It is where the primary business value is created and is often the reason why customers choose your product or service over others
- **Characteristics:** This subdomain is unique to your business and requires internal development to meet specific business needs. It is complex, frequently evolving, and needs continuous refinement and innovation
- **Focus:** Highest priority in terms of resources and attention

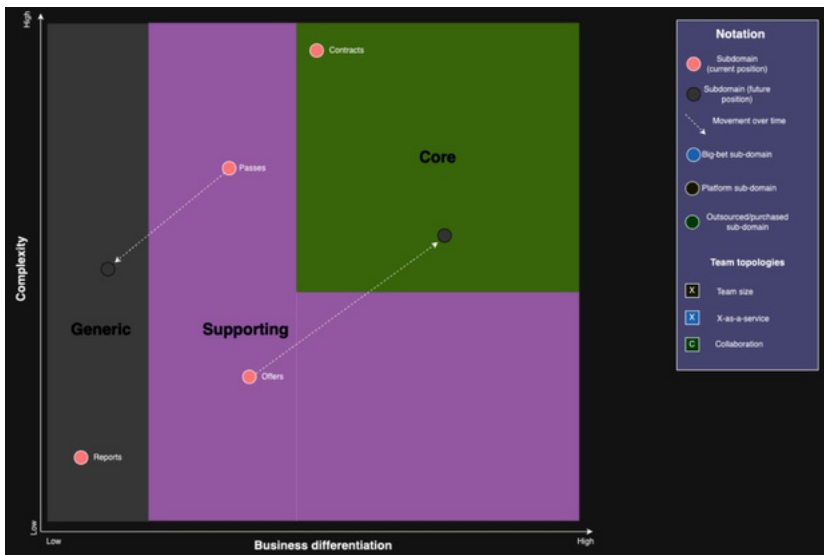
## Supporting

- **Description:** Supporting subdomains are necessary for the business but do not constitute its competitive advantage. They support the core ones but aren't the primary focus of the business
- **Characteristics:** Less complex than the core subdomain, these areas still require specific business logic but are not as critical for the unique selling proposition of the product or service
- **Focus:** Often developed in-house but can be as well outsourced

## Generic

- **Description:** Generic subdomains are the standard functionalities that are common across many businesses and industries. They don't provide competitive advantage and are not specific to the business
- **Characteristics:** These are often standardized processes or features with well-established solutions in the market
- **Focus:** Businesses opt for off-the-shelf solutions

Before you decide which direction your subdomain might take, it's very important to talk to the business about their forecast of how often this subdomain might change. Look at historical data to see how often processes changed in previous years. This will give you a structured way to look at potential variation. I can recommend the use of DDD Crew's Core Domain Charts:



Pic. 14 - Domain chart with map of subdomains

From the diagram, you can see that the Reports module has low complexity and low business differentiation, so there is a high chance that you will find an off-the-shelf solution you can buy.

On the other hand, there are Contracts that have high complexity and above-average business differentiation.

This means that you are likely to have to build them yourself. There are also Offers and Passes that fall into the supporting type of subdomain, but both have the potential to change their nature soon:

- **Offers** may become more complex due to the increased number of policies or regulations that may come into play. The more it is based on your custom internal rules, the more differentiation it will need from a business perspective - you will not find off-the-shelf solutions for it, and additionally, you might want to keep the knowledge about it inside your company instead of outsourcing it
- **Passes** are not so complex today - they still have some business rules - that they could become a generic solution that you and other companies can use. Not today, but in a few months, there may be a player who will sell it as a ready-to-integrate (off-the-shelf) solution

This way, you can create a map to document your thinking and software-building strategy.

Now that we have fine-tuned our subdomains, we are ready for one of the final steps in strategic Domain-Driven Design. It is called **Bounded Contexts**.

## 5. Bounded Contexts

---

One of the most essential steps in strategic Domain-Driven Design is organizing your subdomains within boundaries called **Bounded Contexts**. This is one of the most misunderstood concepts in DDD, and I aim to make it as straightforward as possible for you. To do this, we will start with the simplest possible example of a bounded context and go deeper and more profound:

*Imagine a city. This city has different areas, like a shopping district, a residential area, and an industrial zone. Each area has its own rules and purposes. For example, you can buy clothes and toys in the shopping district, but you can't do that in the industrial zone, where factories make things.*

*Now, think of each of these areas as **a bounded context**. It is like having an invisible fence around each area that sets the rules and activities that can happen inside it. In a shopping district, the language and actions are all about shopping. In the industrial zone, the language and actions are about manufacturing, etc.*

Each bounded context has its own rules, ubiquitous language, and purpose. So, in general, they help people and systems keep things organized and make sense of all the information and ideas they have to deal with.

In the last chapter, we distilled subdomains that represent parts of a concrete business domain - in our case, Fitness Studio.

The following subdomains were discovered:

- Offers
- Contracts
- Passes
- Reports

Using domain charts, each subdomain was assigned to a specific type - core, supporting, or generic. Thanks to this, we are ready to start thinking about the boundaries around it.

At the beginning of a project, there are usually two situations:

- **Each subdomain is treated as a separate bounded context** because they have nothing in common. This means that the Offers subdomain will become the Offers bounded context, the Contracts subdomain will become the Contracts bounded context, etc.
- **A single bounded context wraps multiple subdomains.** This can happen when subdomains are so close together that wrapping them in separate bounded contexts makes no sense. Imagine there are three subdomains: *Assessments*, *Progress Tracking*, and *Virtual Coaching*. You have discovered this:
  - terms and concepts such as trainee, coach, assessment, and progress have the same meaning in all subdomains
  - the rules, workflows, and processes are consistent across the subdomains. Progress that is measured in one subdomain will be measured in a similar way in others unless there is a reason to differ

- It is common for these subdomains to communicate with each other

And that is an indicator that perhaps it makes sense to put them in a **single bounded context called Personalised Training**.

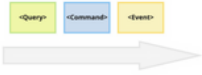



Perhaps.

Remember how, during the process-level Event Storming, we looked deeply into the draft subdomains and made specific changes based on policies, actors, and processes to create the final version of our subdomains? You can do the same with your bounded context drafts using these methods:

- Bounded Context Canvas prepared by the great DDD Crew (what would the world be without them?)
- Bounded Context Event Storming proposed by Radek Maziarka as a step between process-level and design-level event storming. The main problem it addresses is that at the process level, we focus on the whole process and map the workflow of the entire system - we do not want to focus on the details of a concrete bounded context. At the design level, our focus is on aggregates and data modeling. So, if we want to show the workflows and policies within each bounded context, we need a step between the two. **IMPORTANT:** You can still handle it inside the process level, but this way, there will be much work coupled with just one level of ES

In this chapter, I want to show you how to use the first one - the **Bounded Context Canvas**. Let's have a look at it:



<b>Name:</b>		VS github.com/ddd-crew/bounded-context-canvas	
<b>Purpose</b> What benefits does this context provide, and how does it provide them? Describe the purpose from a business perspective	<b>Strategic Classification</b> Domain - core - supporting - generic - other? Business Model - revenue - engagement - compliance - cost reduction Evolution - genesis - custom built - product - commodity		<b>Domain Roles</b> Role Types - draft context - evolution context - analysis context - gateway context - other
<b>Inbound Communication</b> Collaborator      Messages 		<b>Ubiquitous Language</b> Context-specific domain terminology  <b>Business Decisions</b> Key business rules, policies, and decisions 	<b>Outbound Communication</b> Messages      Collaborator 
<b>Assumptions</b> Describe which currently unverified assumptions went into this bounded context design. Make those assumptions explicit by documenting them here		<b>Verification Metrics</b> Describe metrics which can be used to (in)validate the current structure of this bounded context?	
		<b>Open Questions</b>	

Pic. 15 - Bounded Context Canvas template by DDD Crew

The **Name** is clear and represents the name of your bounded context. Next is a field that often provokes thoughts like *Do I need it?* Or *What if we combine it with something else?* called the **Purpose**. Then, the **Strategic Classification** which combines 3 areas:

- **Domain Type:** the same as for subdomains. If there is a 1-1 mapping of a subdomain to a bounded context, it is simple - a subdomain of a core type becomes a bounded context of the same type. If you have several subdomains of different types, e.g., core and supporting, the bounded context will be core; if supporting and generic, it will be Supporting, etc.
- **Business model:** does this bounded context bring you revenue directly? Or maybe users often interact with it and use your application because of it? Or does it protect your business reputation? Or maybe it reduces costs?

- **Evolution:** is your bounded context already explored, and are there off-the-shelf solutions you can buy? Or is it in an emerging market, and you need to build it yourself? You can use Wardley Maps to help you evaluate such kinds of questions

When you are ready with the strategic classification, the next important step is to define the **Domain Roles** of your bounded context. Would it receive much data to be analyzed to provide context to your business or users? Or will it be treated as an execution context, triggering workflows and other bounded contexts?

One of the most essential parts is defining the **Inbound and Outbound Communication**. Starting with the former, you must figure out who will communicate within this bounded context. It can be:

- another bounded context
- external system
- user
- frontend application, etc.

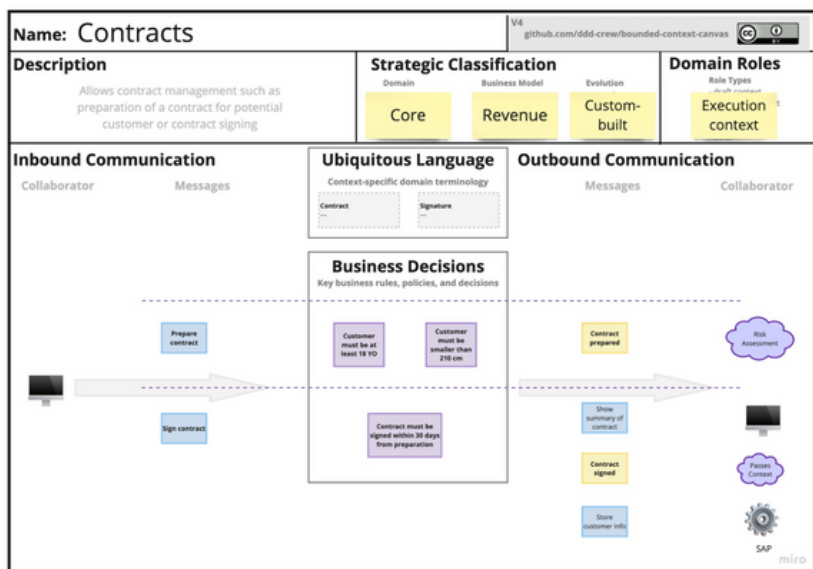
Any of the above can send **Query, Command, or Event**. Then, based on policies and business rules, your bounded context processes the information and can send it to:

- another bounded context
- external system
- user
- frontend application, etc.

again by using **Query, Command, or Event**. You define this type of action in the outbound communication area.

**Assumptions and open questions** about your bounded context can be written in the last section of the bounded context canvas. It is also good behavior to define some verification metrics. This allows you to validate the structure of the selected bounded context.

Enough theory; let's see what it looks like in practice. We experiment with one of our subdomains - the most complicated one - Contracts:



Pic. 16 - Bounded Context Canvas defined for Contracts bounded context

It is a core bounded context that is used for revenue. We decided to make it custom-built as the market is forming and it is seen as a competitive advantage over the others. It will be used for the execution and triggering of other workflows.

What you can see as well is that this bounded context is triggered from the outside by a frontend application with 2 commands:

- Prepare contract
- Sign contract

where each is an entry point to a business process. Each process is restricted by a **swimming lane (-----)** to make it clear which business rules, inbound and outbound communication belong to which business process.

There are policies like the **Customer must be at least 18 YO** that must be passed to finish the process. In the end, 2 events are sent to other bounded contexts (Risk Assessment and Passes) and 2 commands to the frontend application and external system (SAP).

Thanks to this approach, you can validate your ideas about bounded contexts. When working on them, you often realize they don't make sense. Especially at the beginning of the DDD adventure - sometimes, this is very demotivating because we have to invest a lot of time and work. As time passes, you will see that bounded contexts are repeated in different systems - you will spot archetypes.

When it comes to representing bounded contexts in code, using .NET as an example, I recommend that you map them directly to modules.

This means that **1 bounded context = 1 module** in a modular monolith. You can quickly extract one module into a separate deployment unit. It also does not block you from extracting just one process from the module.

If you are interested in this approach, here are links to 2 repositories worth checking out:

- **Evolutionary Architecture By Example**, where I am a co-author - it shows the evolutionary approach of modular monolith where modules represent bounded contexts
- **Modular Monolith with DDD** by Kamil Grzybek - a great place to learn how to build modular monoliths in .NET

**NOTE:** This is just my recommendation. There are other approaches - 1 subdomain per 1 module (or microservice), 1 business process per microservice, etc. The approach I mentioned worked best based on my experience and the products I have worked on, but that doesn't mean it can't be different for you. Let us always remember to respect other approaches and each other! :)

Once we have established the boundaries around our subdomains and wrapped them in bounded contexts, it is time for the final step of strategic DDD - establishing communication between multiple bounded contexts and external systems.

This is called Context Map and is our final focus in this book.

## 6. Context Map

---

The final part of strategic Domain-Driven Design is to define the integration and relationships between different bounded contexts, third-party systems, and teams.

In Chapter 5, I showed you how to define a bounded context using the Bounded Context Canvas. This allowed us to plan a single bounded context's inbound and outbound communication, types, patterns, and business rules. But almost every system is built on many contexts, third-party systems, etc. Furthermore, each needs to communicate with others in one way or another - the same as the teams responsible for them.

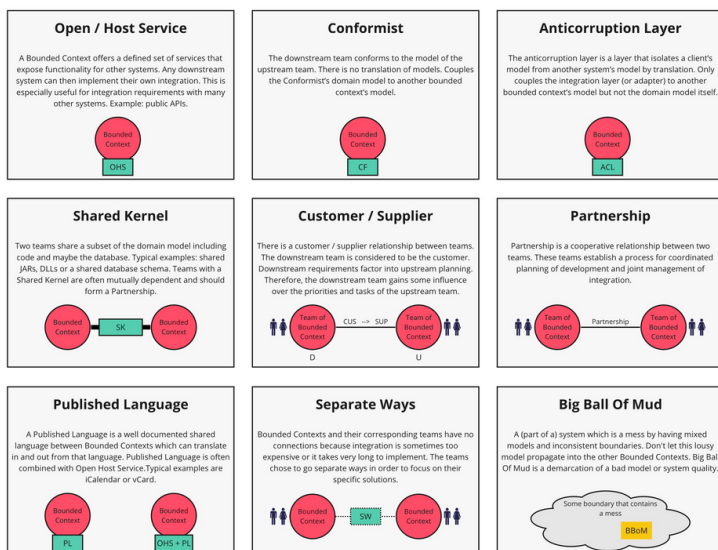
In other words, we need to build interaction and relationships.

This is where a concept called **Context Mapping** comes into play. And there is nothing to be afraid of, although I know from the past that there is much fear of it because it seems complicated. Despite appearances, it is a relatively simple concept, but you must approach it from the right side.

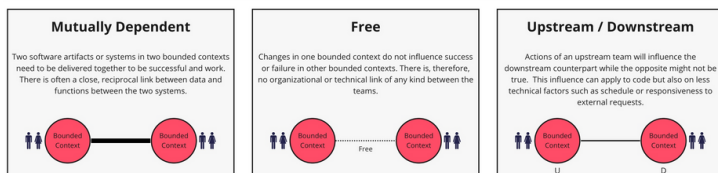
However, before I explain how I approach this, I would like to introduce you to the various context-mapping patterns. Here, as usual, we are supported by materials from the DDD Crew:

# Context Map Cheat Sheet

## Context Map Patterns



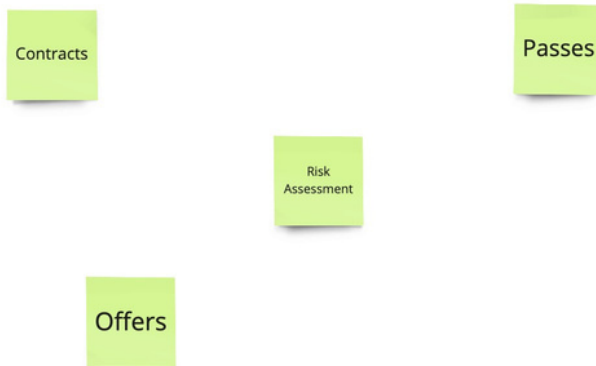
## Team Relationships



Context Map Cheat Sheet v2: <https://github.com/dddcontext-mapping> | License: Creative Commons Attribution-ShareAlike 4.0 | Context Mapping System for DDD Reference by Eric Evans: <https://github.com/ericniebler/ddd-context-mapping> | Reference: 2015-03.pdf

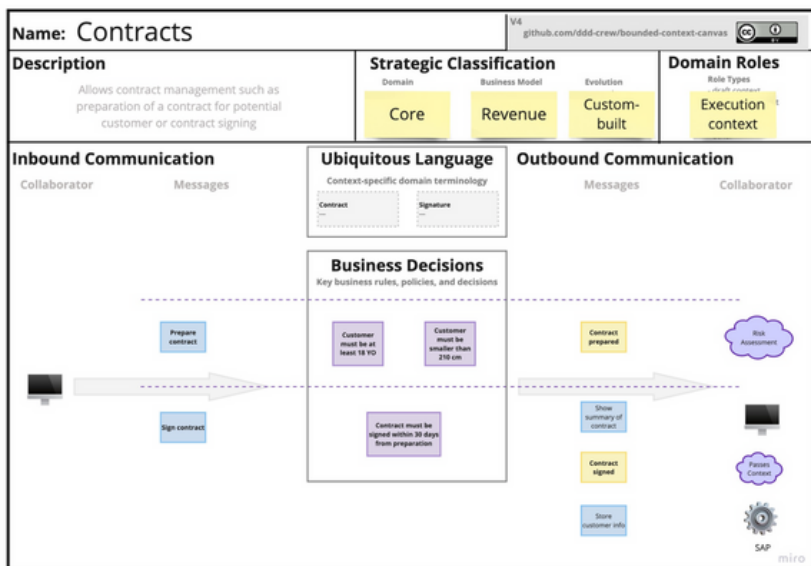
Pic. 17 - A useful cheatsheet for context mapping prepared by DDD Crew

As you can see above, there are many patterns to choose from. Which one to choose? My first step is to write down all the bounded contexts I have. In our case, it will be all that belongs to the **Fitness Studio** domain:



Pic. 18 - All bounded contexts that we want to focus on

Now, we are ready to look back at the Bounded Context Canvas that we defined for Contracts in the previous chapter:



Pic. 19 - Bounded Context Canvas defined for Contracts bounded context



All the inbound communication comes directly from the front end, so no other bounded context is involved. But there is outbound communication that goes to Risk Assessments and Passes bounded contexts. This means that there is interaction between them and the Contracts. Let's focus first on the communication between Contracts and Passes. Two actions are being handled:

- When the contract is signed, this bounded context informs Passes about that fact
- Passes registers a new pass for the customer, and the customer is ready to enter the fitness studio

One of the context mapping concepts is to define upstream and downstream systems:

- **Upstream** is the provider or source of data, services, or functionality. It often acts as the sender in a relationship
- **Downstream** is the receiver or consumer of the upstream system's data, services, or functionality. It relies on the upstream system for certain elements of its operation

We can clearly define that Contracts will be the upstream system - as it delivers the triggering information about contract signature, while Passes will be downstream - as it reacts to a triggered event. Let's mark this on the diagram:



Pic. 20 - Upstream and downstream relation between Contracts and Passes

Now that we have the definition of which context is upstream and which is downstream, we can try to find the best pattern for their interaction. Let's say we want to communicate between them using simple messaging:

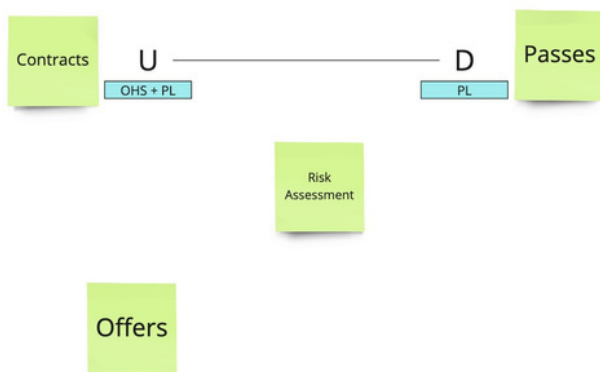
- **Contracts** will send the event to a message broker when a contract is signed
- **Passes** will subscribe to this message broker, and when the event arrives, it will consume it and register the pass

There are at least 2 things to consider:

- It would be nice if we could define a common language between the two bounded contexts
- Changes that happen inside upstream bounded context should not directly affect downstream - it would be cool to have a public interface exposed by **Contracts**

The first requirement can be addressed by using a pattern called **Published Language**. It allows us to define a common language between both contexts.

The second can be handled with **Open-Host Service** - any internal changes in Contracts will not affect Passes as long as the public interface remains unchanged. Our diagram should now look as follows:

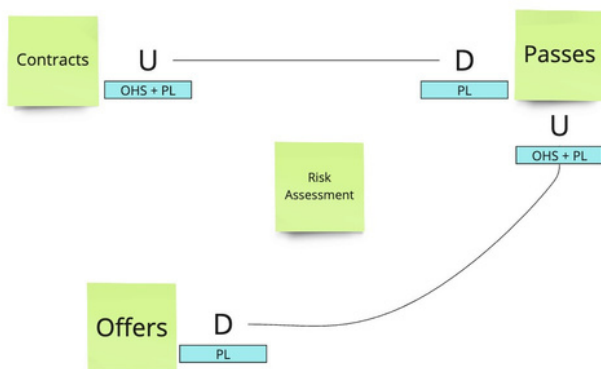


Pic. 21 - Patterns of published language and open-host service

We are ready with the context map between Contracts and Passes. Another interaction that will be very similar in our business domain is the one between Passes and Offers:

- **Passes** will send the event to a message broker when a pass is marked as expired
- **Offers** will subscribe to this message broker, and when the event arrives, it will consume it and prepare a new offer

However, this time Passes will be the upstream bounded context, and Offers will consume it (downstream):



Pic. 22 - Communication patterns between Passes and Offers

Next, we take a look at **Risk Assessment**. In our case, a third party - you can imagine a national credit rating system - provides a public API. It acts as an upstream and also uses the Open-Host Service pattern. Our Contracts bounded context will call it directly (temporary tight coupling) and get a response. However, this response contains a poorly structured model, and we do not want to consume it as it is and act as a Conformist.

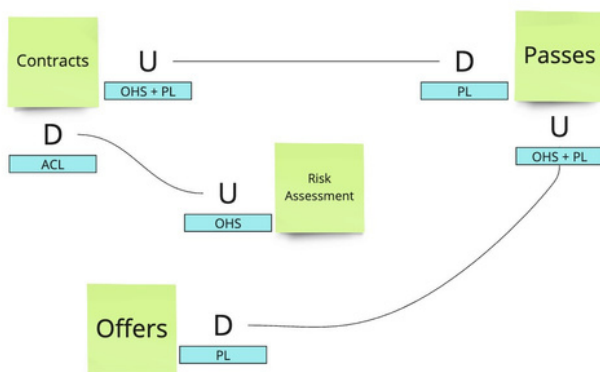
We chose another pattern on the downstream (Contracts) called the **Anti-Corruption Layer**.

It is advantageous when working with legacy or third-party systems - we usually do not want to let the model from the upstream system (or bounded context) flow into ours. Often, such models are messy, contain too much information, or their language does not fit our rules. This is where you use this pattern, which translates the model

into your model.

You might ask - **why not use it always?** Sometimes we have to be conformists. This is usually due to legal requirements, where data from the public system must always be stored in the same format as in the original system. In this case, we can do nothing about it (as much as we would like to), and we have to accept it as Conformists.

What does our Context Map look like now after these changes?



Pic. 23 - Communication between all bounded contexts

**NOTE:** For the downstream calling the Open-Host Service upstream, there may be a Customer-Supplier relationship - imagine your company is the trendsetter whose ideas are often defined as standards. It affects the whole industry and can act as a customer in a Customer-Supplier relationship - as the public API provider desires. This is a rare situation, but it can happen.

Then, you repeat the process for each bounded context. In the end, you have a ready-to-use Context Map.

Congratulations, that's it! You did the strategic analysis of the entire domain.

## 7. Next Steps

---

I hope you learned a lot while reading this book - there is no better thing for the author to hear. Still, this is just the beginning of much more to explore.

Applying the knowledge you gained in a few practical cases would be a good idea. If it is impossible to do that in your company or a team, you can start your open-source project - find a business domain around which you can build the system and document it there. It will help others as well, and at the same time, it may give you the opportunity to promote yourself.

There is no better combination than the theory you read, watched, and practiced. And it is extremely important to have as much practice as possible - then, DDD will become easier and more accessible.

I can recommend the following materials to continue your adventure with DDD:

- [Learning Domain-Driven Design](#) by Vlad Khononov
- [All materials](#) by DDD Crew
- [The Blue Book](#) by Eric Evans
- [Implementing Domain-Driven Design](#) by Vaughn Vernon

Each of the above will allow you to make the next step.

One last thing - do not fall into a trap called *I have to apply all I learned about DDD*. You don't. **Use only what you need.**

# Share this book with others!

By sharing this free ebook on your social media, more people will read it - it will also be a *thank you*, for which I will be very grateful!

Please let me know if you enjoyed it - you can find me on [Linkedin](#) and on [YouTube](#). If you find anything that could be improved in this book, please let me know as well.

## Feel free to reach me out

If your company struggles with legacy systems, wants to migrate them, does not know how to build the MVP, or feels it would be great to have a workshop with me, you can check one of my pages:

[Cooperation](#)

[Workshops](#)

If you need help with your business in the area of IT, please do not hesitate to book a free consultation with me.

**Book it now**

