Implementing Domain-Driven Design with Java

Premanand Chandrasekaran, Karthik Krishnan

Version 1.0.0-SNAPSHOT, May 02 2022 02:03 PM UTC

Table of Contents

Preface

Domain-Driven Design makes available a set of techniques and patterns that non-technical experts, architects and developers to work together and decompose complex systems into well-factored, collaborating, loosely coupled subsystems. Write more here… TODO.

Who This Book Is For

Developers working with Domain-Driven Design will be able to put their knowledge to work with this practical guide to create elegant software designs that are pleasant to work with and easy to work with and reason about. The book provides a hands-on approach to implementation and associated methodologies that will have you up-and-running, and productive in no time.

What You Will Learn

By the end of this book, you will be able to architect, design and implement robust, modern and loosely coupled distributed architectures employing domain-driven design.

Acknowledgements

TODO

Part 1: Foundations

While the IT industry prides itself on being at the very bleeding edge of technology, it also oversees a relatively high proportion of projects that fail outright or do not meet their originally intended goals for one reason or another. In Part 1, we will look at reasons for software projects not achieving their intended objectives and how practising Domain-Driven Design (DDD) can significantly help improve the odds of achieving success. We will also do a quick tour of the main concepts that Eric Evans elaborated in his seminal book by the same name and examine why/how it is extremely relevant in today's distributed systems age.

Chapter 1. The Rationale for Domain-Driven Design

> The being cannot be termed rational or virtuous, who obeys any authority, but that of reason.
>
> — Mary Wollstonecraft

1.1. Introduction

According to the Project Management Institute's (PMI) *Pulse of the Profession* report published in February 2020, only **77%** of all projects meet their intended goals — and even this is true only in the most mature organizations. For less mature organizations, this number falls to just **56%** i.e. approximately one in every two projects does not meet its intended goals. Furthermore, approximately one in every five projects is declared an outright failure. At the same time, we also seem to be embarking on our most ambitious and complex projects.

In this chapter, we will examine the main causes for project failure and look at how applying domain-driven design provides a set of guidelines and techniques to improve the odds of success in our favor. While Eric Evans wrote his classic book on the subject way back in 2003, we look at why that work is still extremely relevant in today's times.

1.2. Why do software projects fail?

> Failure is simply the opportunity to begin again, this time more intelligently.
>
> — Henry Ford

According to the project success report published in the Project Management Journal of the PMI, the following six factors need to be true for a project to be deemed successful:

Project Success Factors

Category	Criterion	Description
Project	Time	It meets the desired time schedules
	Cost	Its cost does not exceed budget
	Performance	It works as intended
Client	Use	Its intended clients use it
	Satisfaction	Its intended clients are happy
	Effectiveness	Its intended clients derive direct benefits through its implementation

With all of these criteria being applied to assess project success, a large percentage of projects fail for one reason or another. Let's examine some of the top reasons in more detail:

1.2.1. Inaccurate requirements

PMI's *Pulse of the Profession* report from 2017 highlights a very starking fact — a vast majority of projects fail due to inaccurate or misinterpreted requirements. It follows that it is impossible to build something that clients can use, are happy with and makes them more effective at their jobs if the wrong thing gets built — even much less for the project to be built on time, and under budget.

IT teams, especially in large organizations are staffed with mono-skilled roles such as UX designer, developer, tester, architect, business analyst, project manager, product owner, business sponsor, etc. In a lot of cases, these people are parts of distinct organization units/departments — each with its own set of priorities and motivations. To make matters even worse, the geographical separation between these people only keeps increasing. The need to keep costs down and the current COVID-19 ecosystem does not help matters either.

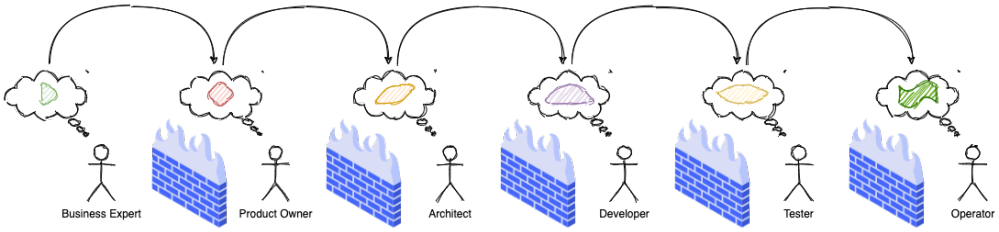


Figure 1- 1. Silo mentality and the loss of information fidelity

All this results in a loss in fidelity of information at every stage in the *assembly line*, which then results in misconceptions, inaccuracies, delays and eventually failure!

1.2.2. Too much architecture

Writing complex software is quite a task. One cannot just hope to sit down and start typing code — although that approach might work in some trivial cases. Before translating business ideas into working software, a thorough understanding of the problem at hand is necessary. For example, it is not possible (or at least extremely hard) to build credit card software without understanding how credit cards work in the first place. To communicate one's understanding of a problem, it is not uncommon to create software models of the problem, before writing code. This model or collection of models represents the understanding of the problem and the architecture of the solution.

Efforts to create a perfect model of the problem — one that is accurate in a very broad context, are not dissimilar to the proverbial holy grail quest. Those accountable to produce the architecture can get stuck in analysis paralysis and/or big design up front, producing artifacts that are one or more of too high level, wishful, gold-plated, buzzword-driven, disconnected from the real world — while not solving any real business problems. This kind of *lock-in* can be especially detrimental during the early phases of the project when knowledge levels of team members are still up and coming. Needless to say, projects adopting such approaches find it hard to meet with success consistently.

For a more comprehensive list of modeling anti-patterns, refer to Scott W. Ambler's website (http://agilemodeling.com) and book dedicated to the subject.

1.2.3. Too little architecture

Agile software delivery methods manifested themselves in the late 90s, early 2000s in response to heavyweight processes collectively known as *waterfall*. These processes seemed to favor big design up front and abstract ivory tower thinking based on wishful, ideal world scenarios. This was based on the premise that thinking things out well in advance ends up saving serious development headaches later on as the project progresses.

In contrast, agile methods seem to favor a much more nimble and iterative approach to software development with a high focus on working software over other artifacts such as documentation. Most teams these days claim to practice some form of iterative software development. However, this obsession to claim conformance to a specific family of agile methodologies as opposed to the underlying principles, a lot of teams misconstrue having just enough architecture with having no perceptible architecture. This results in a situation where adding new features or enhancing existing ones takes a lot longer than what it previously used to — which then accelerates the devolution of the solution to become the dreaded big ball of mud.

1.2.4. Excessive incidental complexity

Mike Cohn popularized the notion of the test pyramid where he talks about how a large number of unit tests should form the foundation of a sound testing strategy — with numbers decreasing significantly as one moves up the pyramid. The rationale here is that as one moves up the pyramid, the cost of upkeep goes up copiously while speed of execution slows down manifold. In reality though, a lot of teams seem to adopt a strategy that is the exact opposite of this — known as the testing ice cream cone as depicted below:

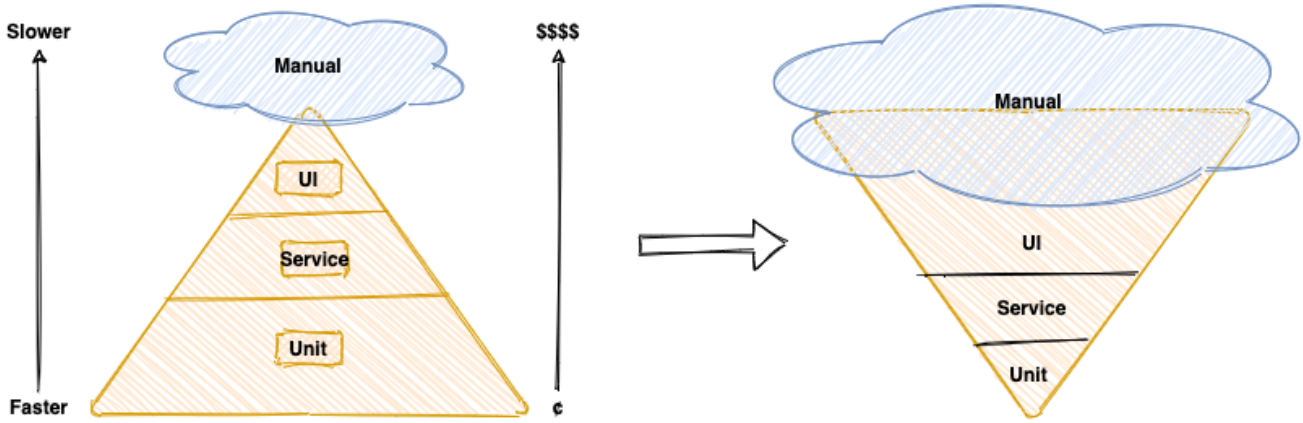


Figure 1- 2. Testing Strategy: Expectation vs. Reality

The testing ice cream cone is a classic case of what Fred Brooks calls incidental complexity in his seminal paper titled No Silver Bullet — Essence and Accident in Software Engineering. All software has some amount of essential complexity that is inherent to the problem being solved. This is especially true when creating solutions for non-trivial problems. However, incidental or accidental complexity is not directly attributable to the problem itself — but is caused by limitations of the people involved, their skill levels, the tools and/or abstractions being used. Not keeping tabs on incidental complexity causes teams to veer away from focusing on the real problems, solving which provide the most value. It naturally follows that such teams minimize their odds of success appreciably.

1.2.5. Uncontrolled technical debt

Financial debt is the act of borrowing money from an outside party to quickly finance the operations of a business — with the promise to repay the principal plus the agreed upon rate of interest in a timely manner. Under the right circumstances, this can accelerate the growth of a business considerably while allowing the owner to retain ownership, reduced taxes and lower interest rates. On the other hand, the inability to pay back this debt on time can adversely affect credit rating, result in higher interest rates, cash flow difficulties, and other restrictions.

Technical debt is what results when development teams take arguably sub-optimal actions to expedite the delivery of a set of features or projects. For a period of time, just like borrowed money allows you to do things sooner than you could otherwise, technical debt can result in short term speed. In the long term, however, software teams will have to dedicate a lot more time and effort towards simply managing complexity as opposed to thinking about producing architecturally sound solutions. This can result in a vicious negative cycle as illustrated in the diagram below:

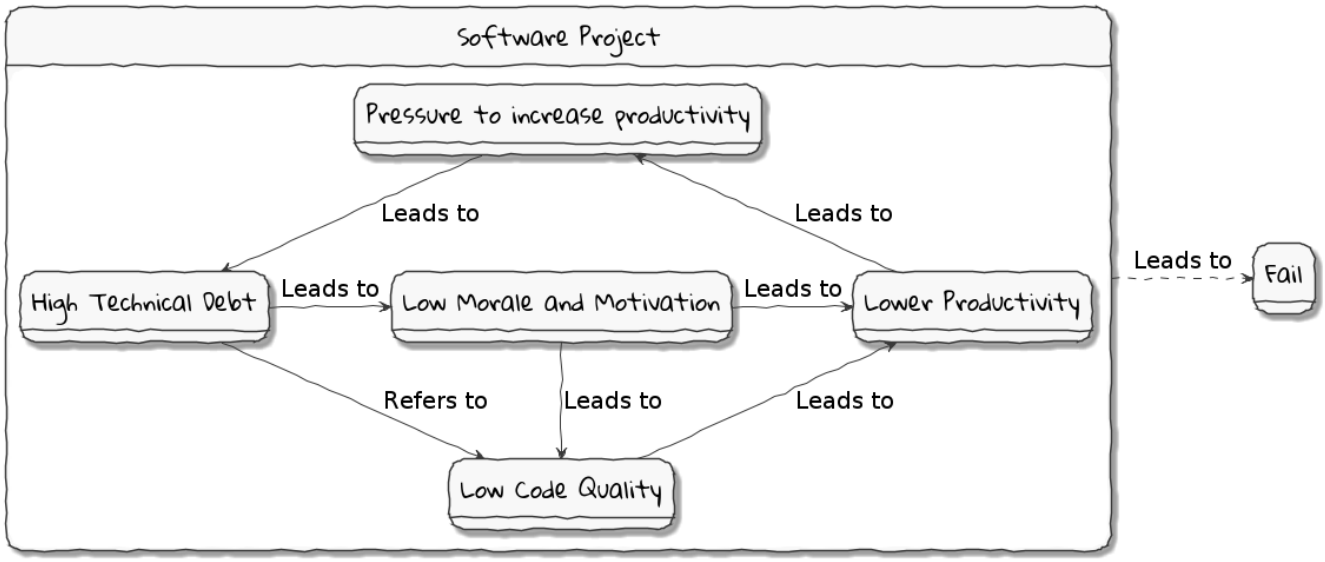


Figure 1- 3. Technical Debt — Implications

In a recent McKinsey survey sent out to CIOs, around 60% reported that the amount of tech debt increased over the past three years. At the same time, over 90% of CIOs allocated less than a fifth of their tech budget towards paying it off. Martin Fowler explores the deep correlation between high software quality (or the lack thereof) and the ability to enhance software predictably. While carrying a certain amount of tech debt is inevitable and part of doing business, not having a plan to systematically pay off this debt can have significantly detrimental effects on team productivity and ability to deliver value.

1.2.6. Ignoring Non-Functional Requirements (NFRs)

Stakeholders often want software teams to spend a majority (if not all) of their time working on features that provide enhanced functionality. This is understandable given that such features provide the highest ROI.These features are called functional requirements.

Non-functional requirements, on the other hand, are those aspects of the system that do not affect functionality directly, but have a profound effect on the efficacy of those using these using and maintaining these systems. There are many kinds of NFRs. A partial list of common NFRs is

depicted below:

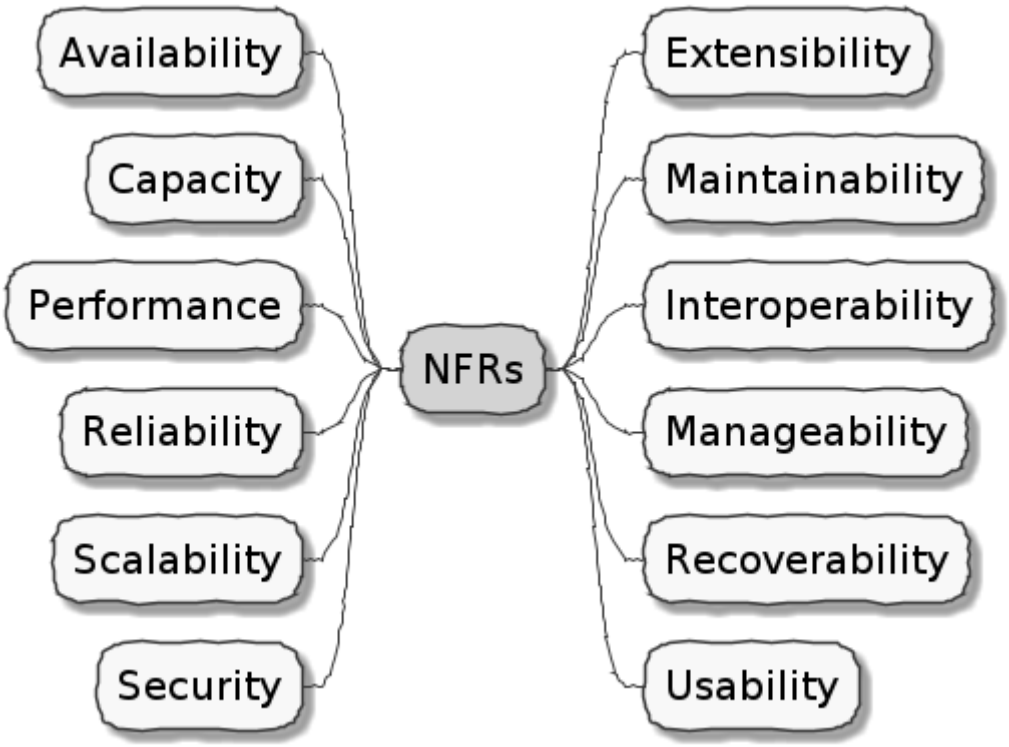


Figure 1- 4. Non-Functional Requirements

Very rarely do users explicitly request non-functional requirements, but almost always expect these features to be part of any system they use. Oftentimes, systems may continue to function without NFRs being met, but not without having an adverse impact on the *quality* of the user experience. For example, the home page of a web site that loads in under 1 second under low load and takes upwards of 30 seconds under higher loads may not be usable during those times of stress. Needless to say, not treating non-functional requirements with the same amount of rigor as explicit, value-adding functional features, can lead to unusable systems — and subsequently failure.

In this section we examined some common reasons that cause software projects to fail. Is it possible to improve our odds? Before we do that, let's look at the nature of modern software systems and how we can deal with the ensuing complexity.

1.3. Modern systems and dealing with complexity

> We can not solve our problems with the same level of thinking that created them.
>
> — Albert Einstein

As we have seen in the previous section, there are several reasons that cause software endeavors to fail. In this section, we will look to understand how software gets built, what the currently prevailing realities are and what adjustments we need to make in order to cope.

1.3.1. How software gets built

Building successful software is an iterative process of constantly refining knowledge and expressing it in the form of models. We have attempted to capture the essence of the process at a high level here:

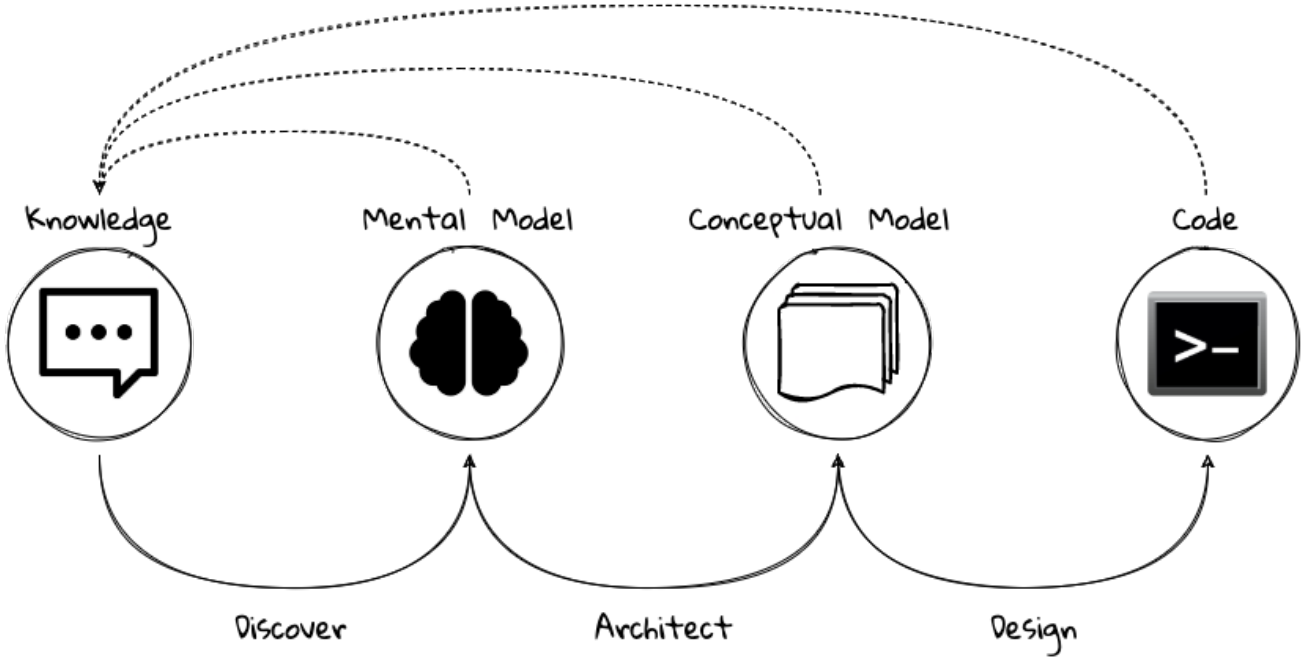


Figure 1- 5. Building software is a continuous refinement of knowledge and models

Before we express a solution in working code, it is necessary to understand **what** the problem entails, **why** the problem is important to solve, and finally, **how** it can be solved. Irrespective of the methodology used (waterfall, agile, and/or anything in between), the process of building software is one where we need to constantly use our knowledge to refine mental/conceptual models to be able to create valuable solutions.

1.3.2. Complexity is inevitable

We find ourselves in the midst of the fourth industrial revolution where the world is becoming more and more digital — with technology being a significant driver of value for businesses. Exponential advances in computing technology as illustrated by Moore's Law below,

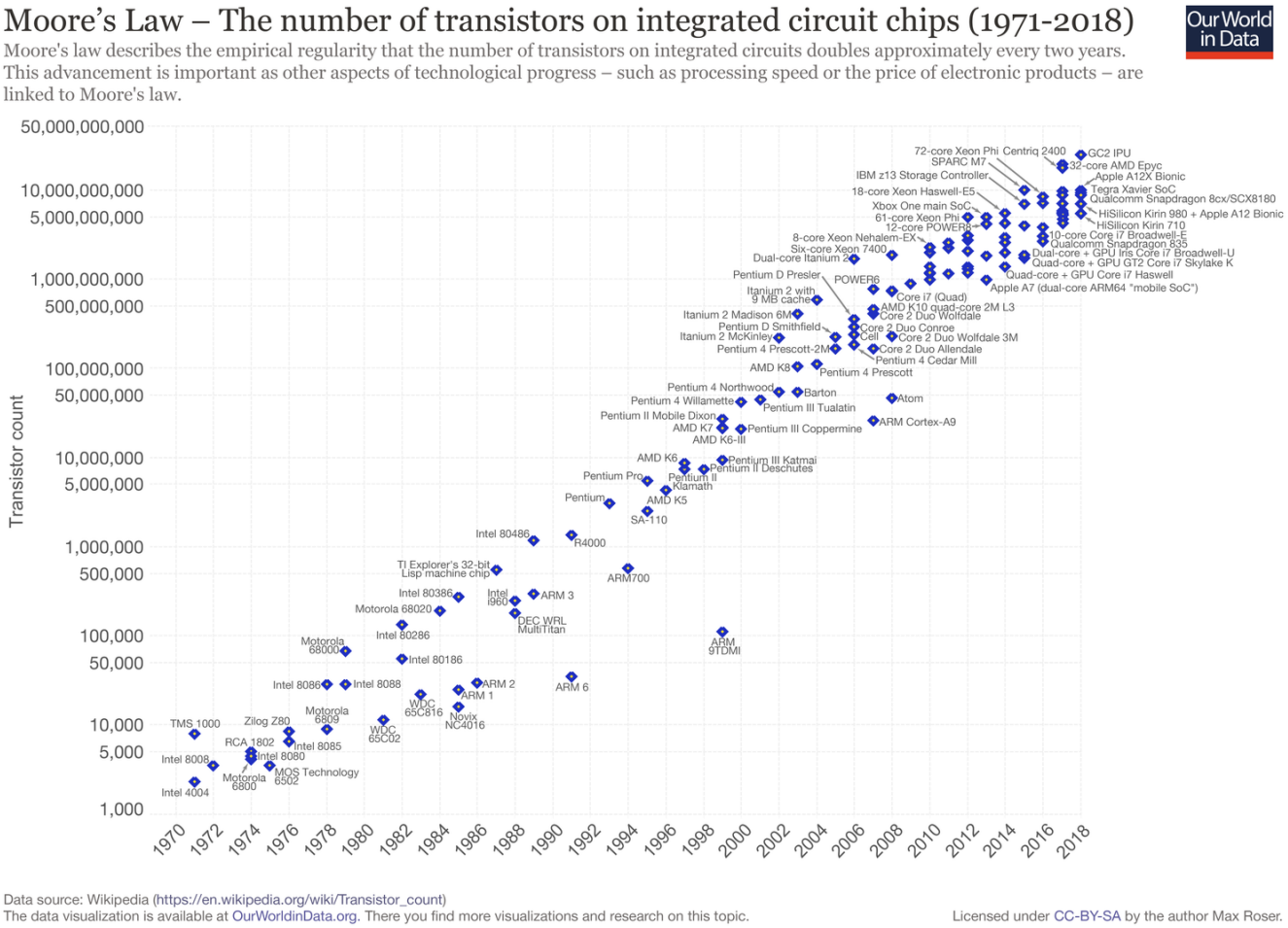


Figure 1- 6. Moore's Law

along with the rise of the internet as illustrated below,

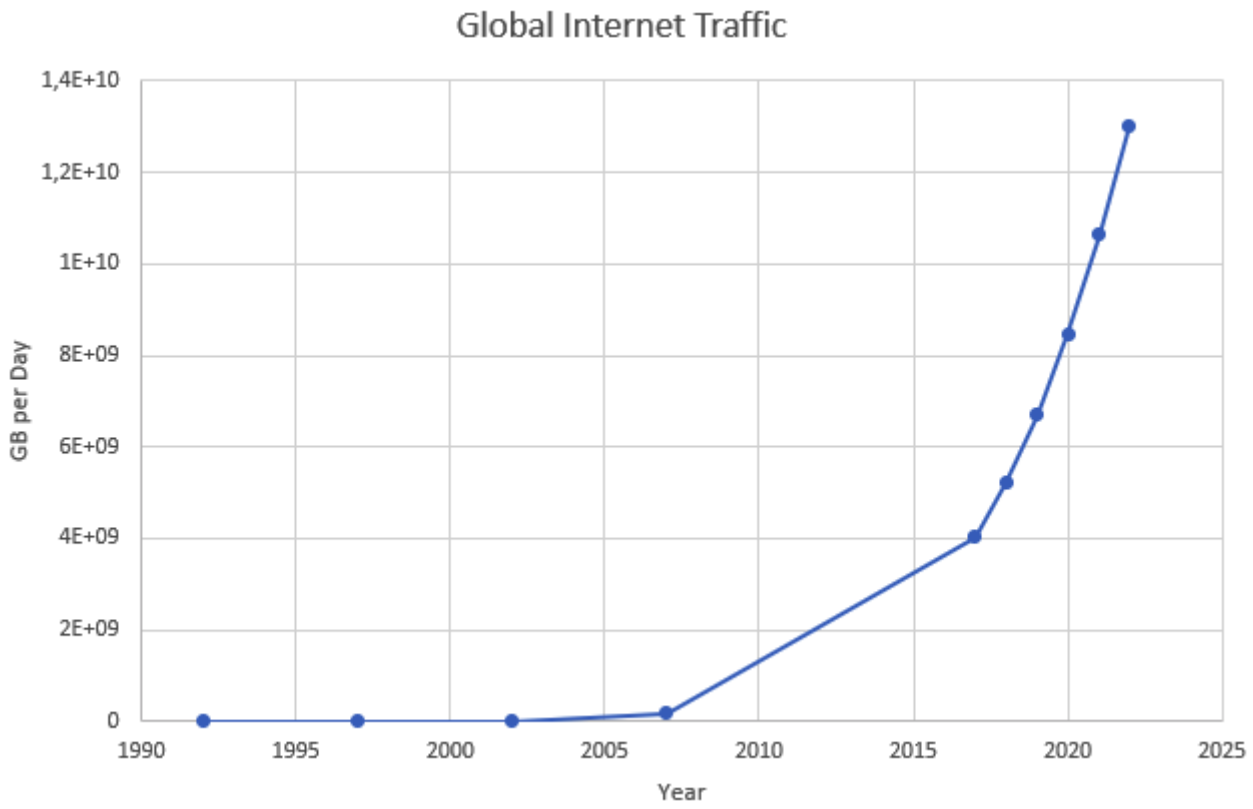


Figure 1- 7. Global Internet Traffic

has meant that companies are being required to modernize their software systems much more rapidly than they ever have. Along with all this, the onset of commodity computing services such as the public cloud has led to a move away from expensive centralized computing systems to more distributed computing ecosystems. As we attempt building our most complex solutions, monoliths are being replaced by an environ of distributed, collaborating microservices. Modern philosophies and practices such as automated testing, architecture fitness functions, continuous integration, continuous delivery, devops, security automation, infrastructure as code, to name a few, are disrupting the way we deliver software solutions.

All these advances introduce their own share of complexity. Instead of attempting to control the amount of complexity, there is a need to embrace and cope with it.

1.3.3. Optimizing the feedback loop

As we enter an age of encountering our most complex business problems, we need to embrace new ways of thinking, a development philosophy and an arsenal of techniques to iteratively evolve mature software solutions that will stand the test of time. We need better ways of communicating, analyzing problems, arriving at a collective understanding, creating and modeling abstractions, and then implementing, enhancing the solution.

To state the obvious — we're all building software with seemingly brilliant business ideas on one side and our ever-demanding customers on the other, as shown here:

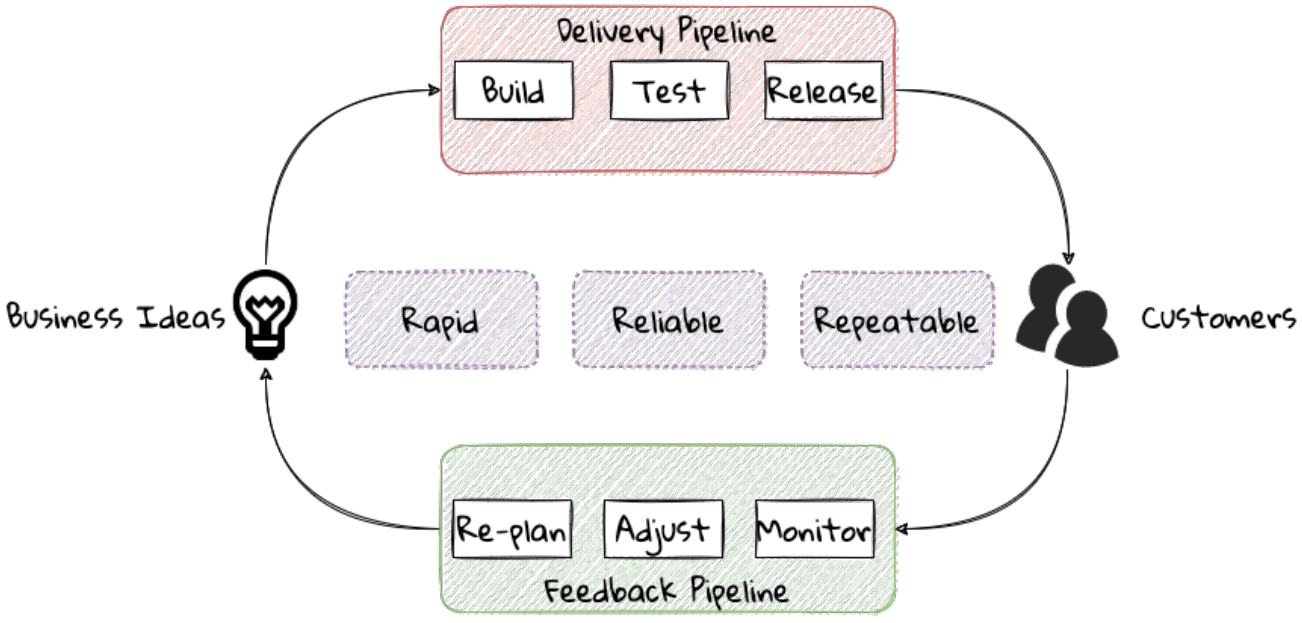


Figure 1- 8. The software delivery continuum

In between, we have two chasms to cross — the *delivery pipeline* and the *feedback pipeline*. The delivery pipeline enables us to put software in the hands of our customers, whereas the feedback pipeline allows us to adjust and adapt. As we can see, this is a continuum. And if we are to build better, more valuable software, this continuum, this potentially infinite loop has to be optimized!

To optimize this loop, we need three characteristics to be present: we need to be fast, we need to be reliable, and we need to do this over and over again. In other words, we need to be rapid, reliable and repeatable — all at the same time!! Take any one of these away, and it just won't sustain.

Domain-driven design promises to provide answers on how to do this in a systematic manner. In the upcoming section, and indeed the rest of this book, we will examine what DDD is and why it is indispensable when working to provide solutions for non-trivial problems in today's world of massively distributed teams and applications.

1.4. What is Domain-Driven Design?

> Life is really simple, but we insist on making it complicated.
>
> — Confucius

In the previous section, we saw how a myriad of reasons coupled with system complexity get in the way of software project success. The idea of domain-driven design, originally conceived by Eric Evans in his 2003 book, is an approach to software development that focuses on expressing software solutions in the form of a model that closely embodies the core of the problem being solved. It provides a set of principles and systematic techniques to analyze, architect and implement software solutions in a manner that enhances chances of success.

While Evans' work is indeed seminal, ground-breaking, and way ahead of its time, it is not prescriptive at all. This is a strength in that it has enabled evolution of DDD beyond what Evans had originally conceived at the time. On the other hand, it also makes it extremely hard to define what DDD actually encompasses, making practical application a challenge. In this section, we will look at some foundational terms and concepts behind domain-driven design. Elaboration and practical application of these concepts will happen in upcoming chapters of this book.

When encountered with a complex business problem:

1. **Understand the problem**: To have a deep, shared understanding of the problem, it is necessary for business experts and technology experts to collaborate closely. Here we collectively understand what the problem is and why it is valuable to solve. This is termed as the **domain** for the problem.
2. **Break down the problem** into more manageable parts: To keep complexity at manageable levels, break down complex problems into smaller, independently solvable parts. These parts are termed as **subdomains**. It may be necessary to further break down subdomains where the subdomain is still too complex. Assign explicit boundaries to limit the functionality of each subdomain. This boundary is termed as the **bounded context** for that subdomain. It may also be convenient to think of the subdomain as a concept that makes more sense to the domain experts (in the problem space), whereas the bounded context is a concept that makes more sense to the technology experts (in the solution space).
3. For each of these bounded contexts:
    a. **Agree on a shared language**: Formalize the understanding by establishing a shared language that is applicable unambiguously within the bounds of the subdomain. This shared language is termed as the ubiquitous language of the domain.
    b. **Express understanding in shared models**: In order to produce working software, express the ubiquitous language in the form of shared models. This model is termed as the **domain model**. There may exist multiple variations of this model, each meant to clarify a specific aspect of the solution. For example, a process model, a sequence diagram, working code, a

deployment topology, etc.

4. **Embrace incidental complexity** of the problem: It is important to note that it is not possible to shy away from the essential complexity of a given problem. By breaking down the problem into subdomains and bounded contexts, we are attempting to distribute it (more or less) evenly across more manageable parts.
5. **Continuously evolve** for greater insight: It is important to understand that the above steps are not a one-time activity. Businesses, technologies, processes and our understanding of these evolve, it is important for our shared understanding to remain in sync with these models through continuous refactoring.

A pictorial representation of the essence of domain-driven design is expressed here:

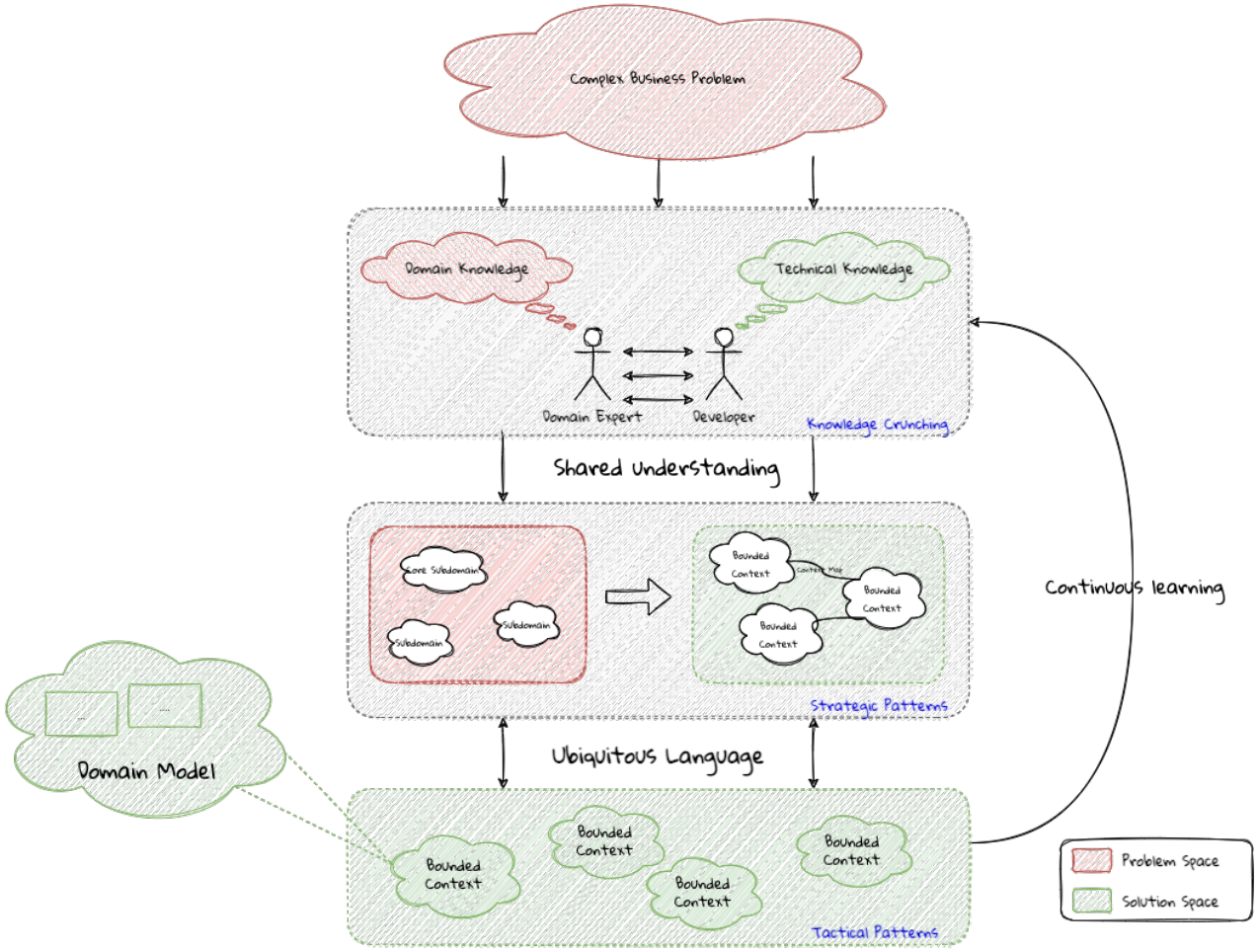


Figure 1- 9. Essence of DDD

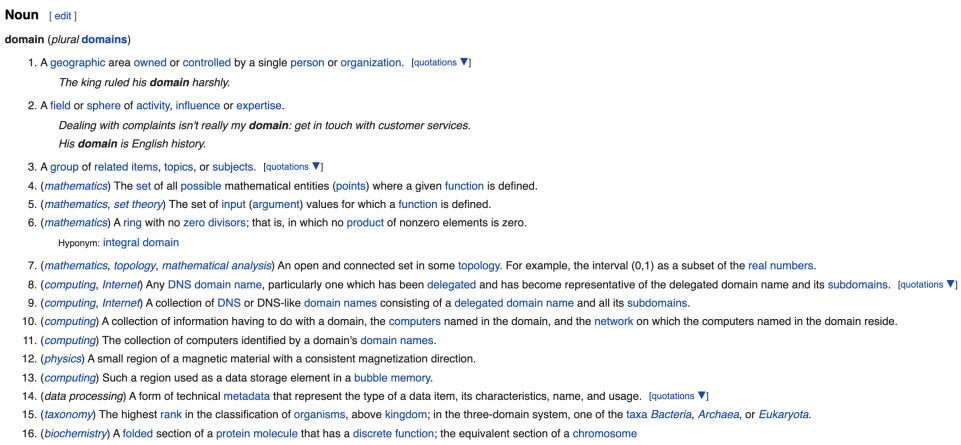
We appreciate that this is quite a whirlwind introduction to the subject of domain-driven design.

1.4.1. 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- 10. **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 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:

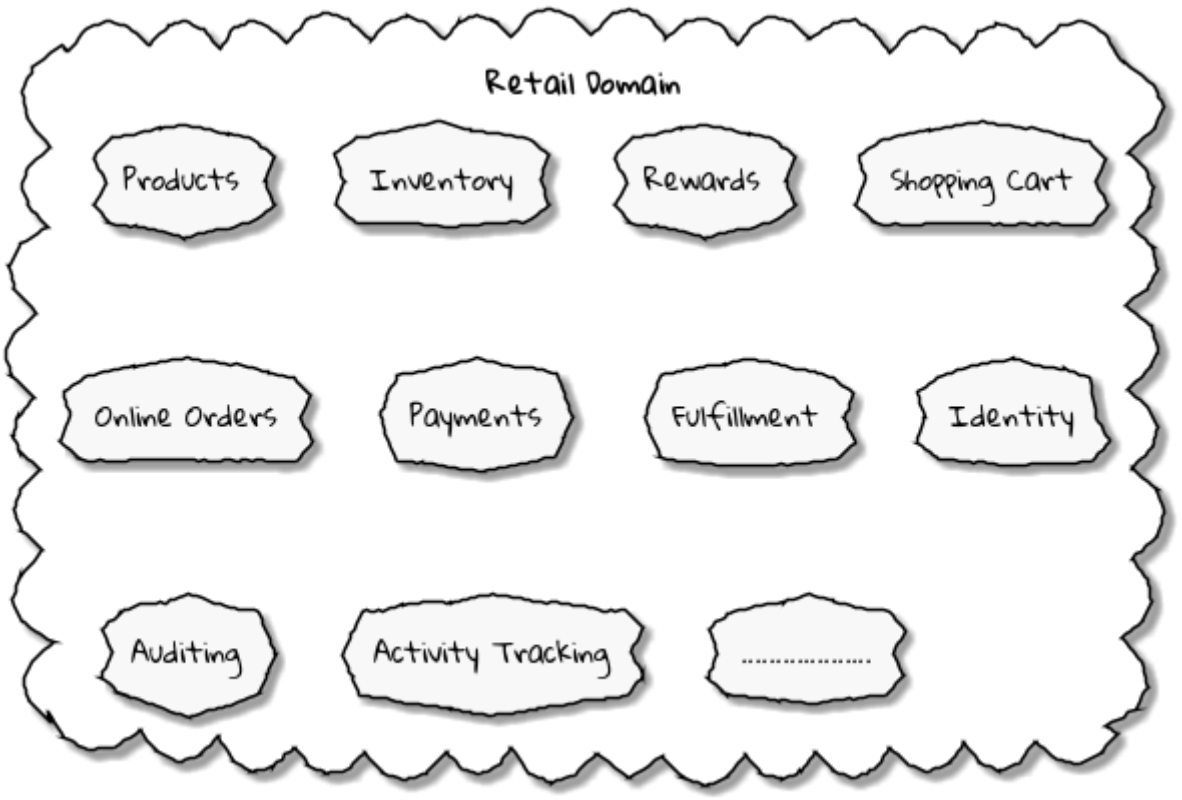


Figure 1- 11. Subdomains in the Retail domain

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:

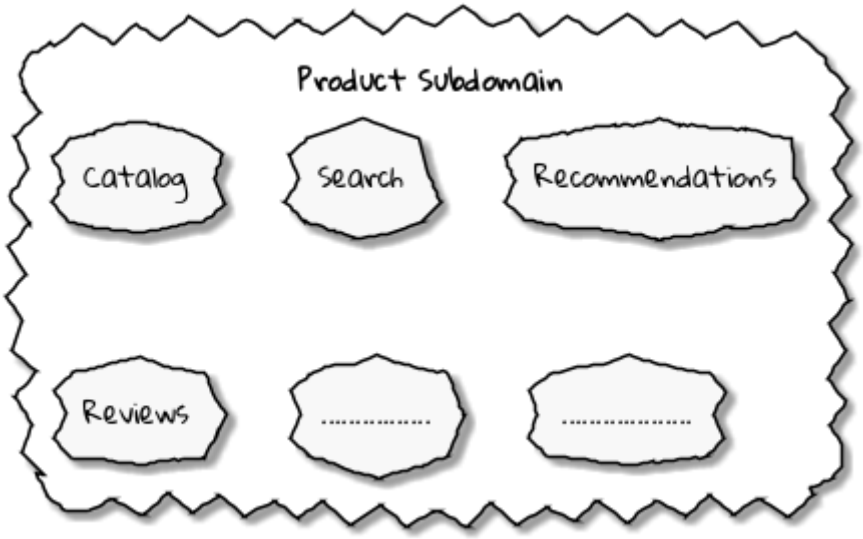


Figure 1- 12. Subdomains in the Products subdomain

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.

The terms domain and subdomains tend to get used interchangeably quite often.This can be confusing to the casual onlooker.Given that sub(domains) tend to be quite complex and hierarchical, a subdomain can be a domain in its own right.

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 subdomain.
- **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:

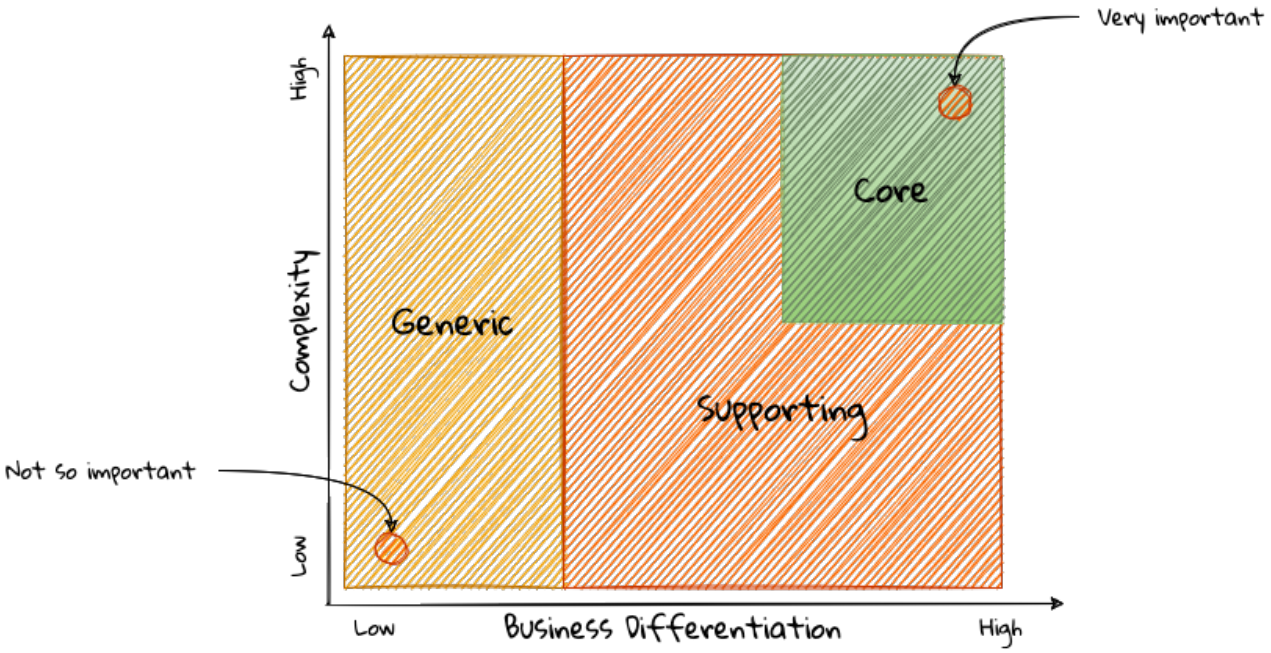


Figure 1- 13. Importance of subdomains

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:

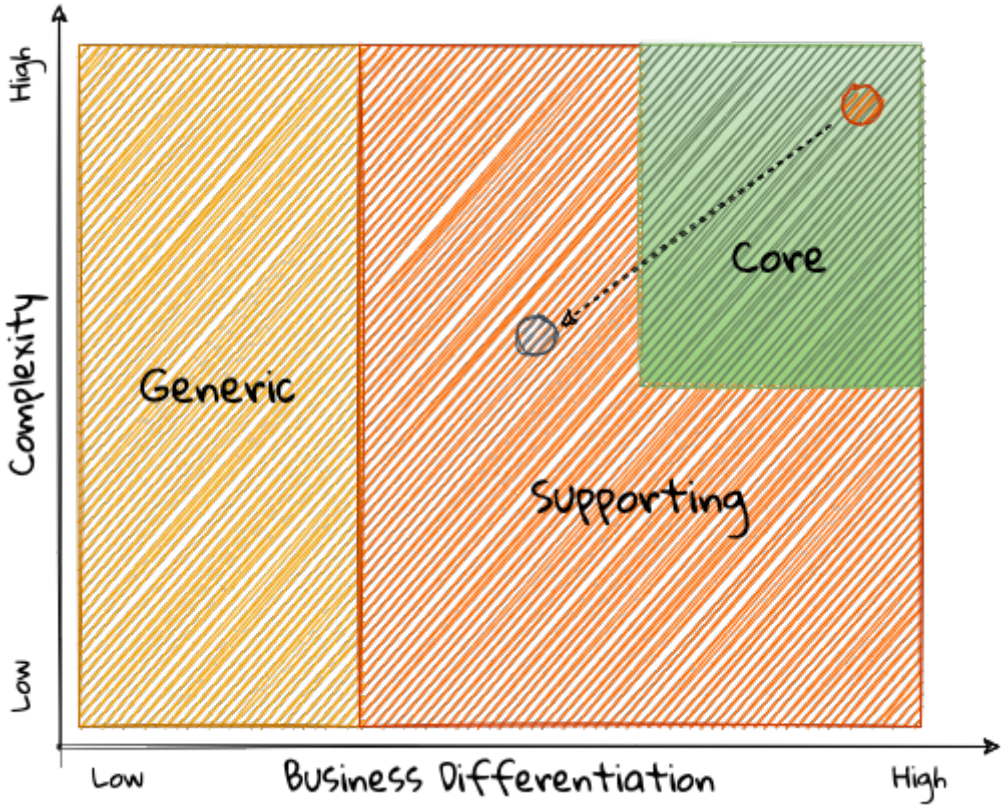


Figure 1- 14. Core domain erosion

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:

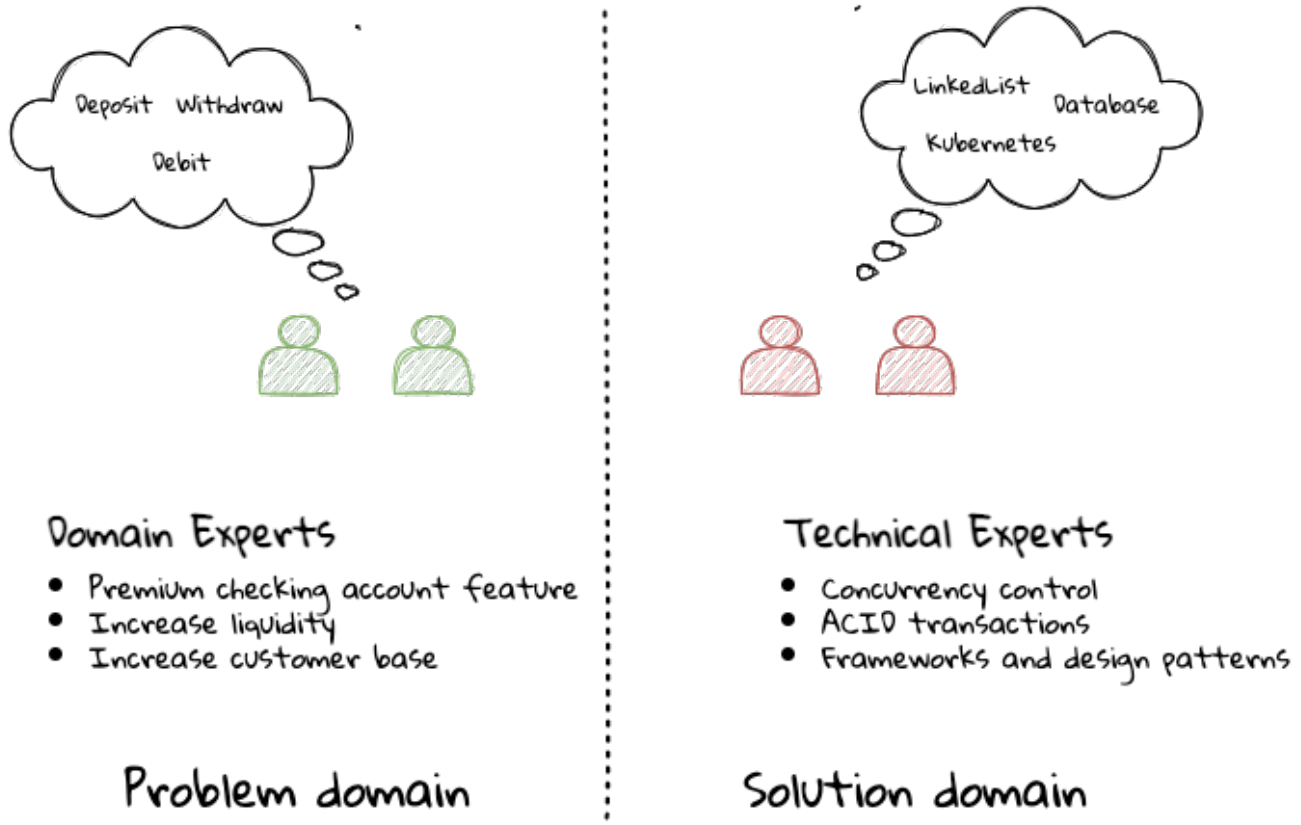


Figure 1- 15. Divide originating in language

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:

The use of the word domain in this context is made in an abstract sense — not to be confused with the concept of the business domain introduced earlier.

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:

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 from a software implementation perspective. 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 (for example, one that is sound from an engineering, software architecture standpoint), 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:

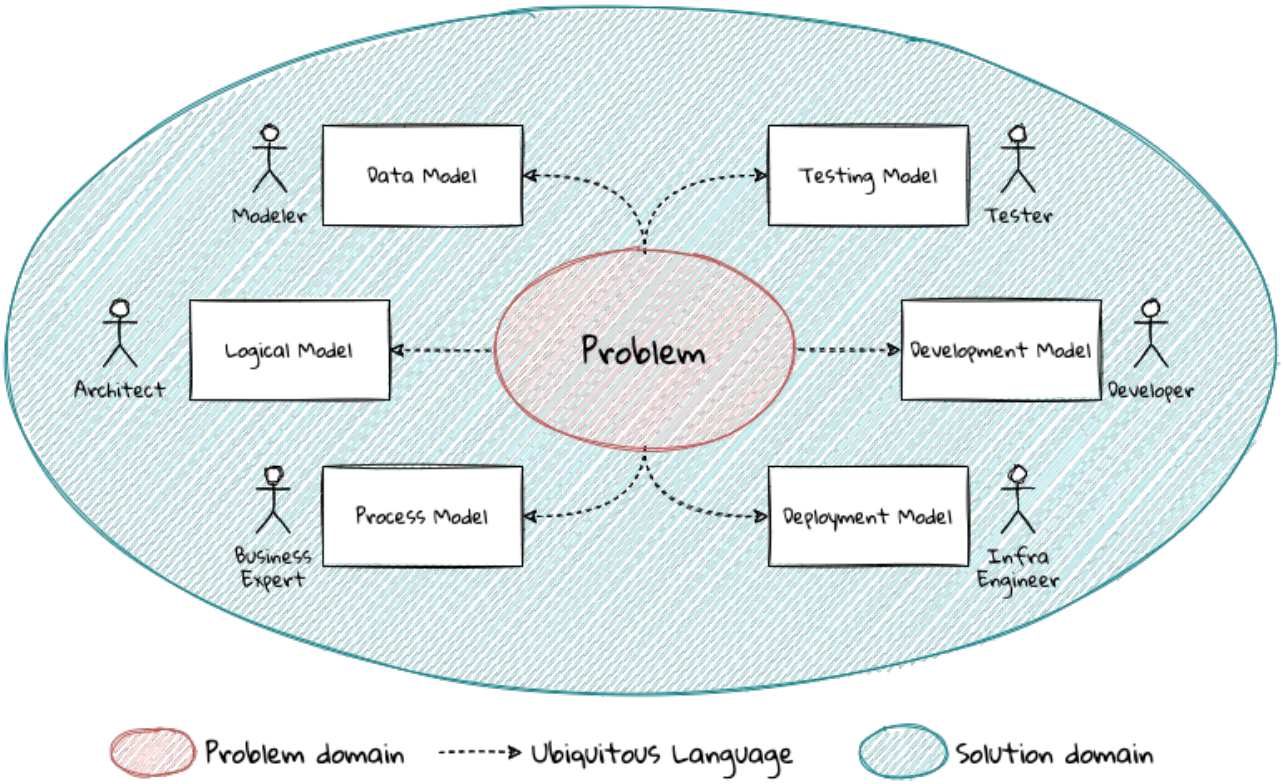


Figure 1- 16. Multiple models to represent the solution to the problem using the ubiquitous language

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:

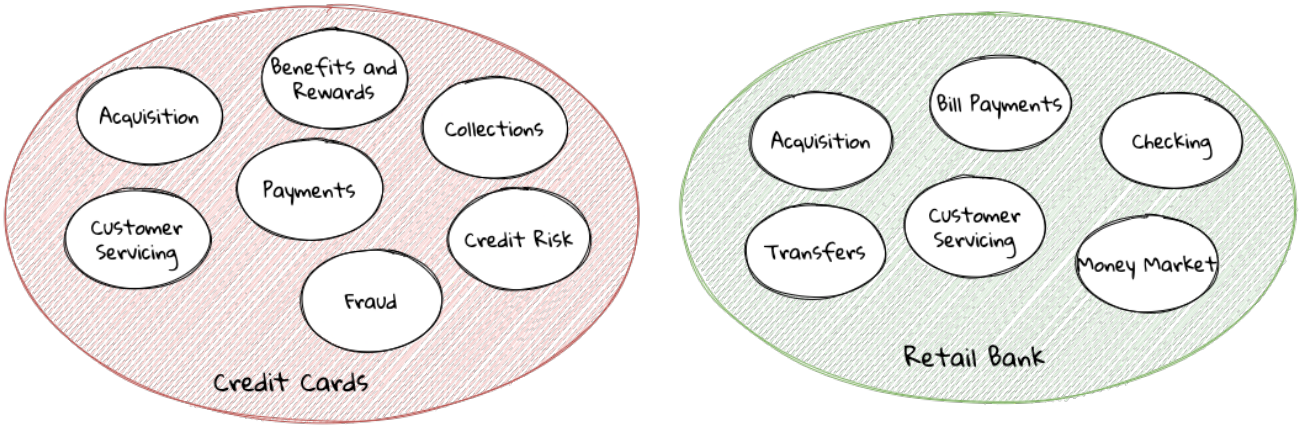


Figure 1- 17. Banking subdomains at Acme bank

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

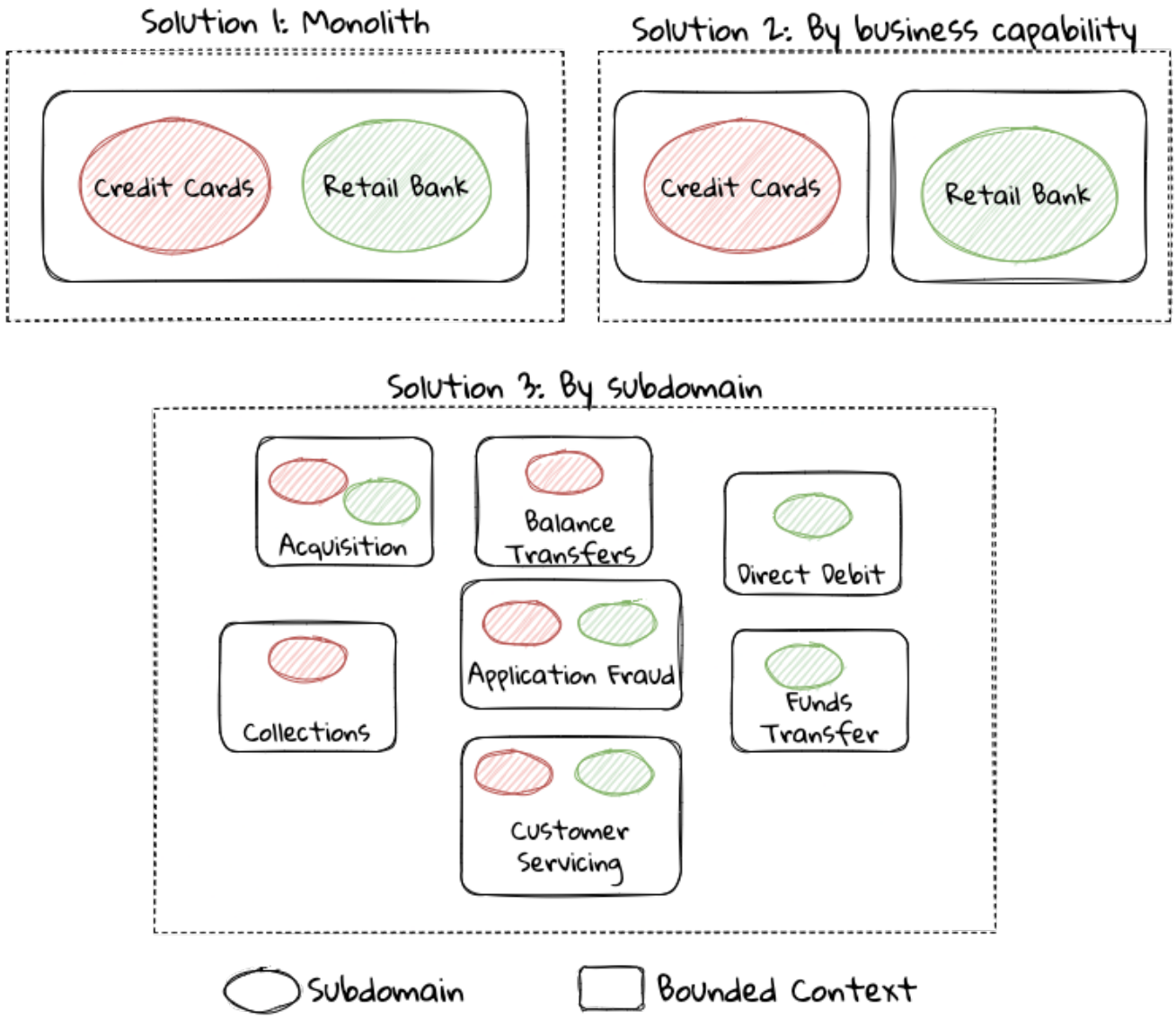


Figure 1- 18. Bounded contexts options at Acme bank

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.

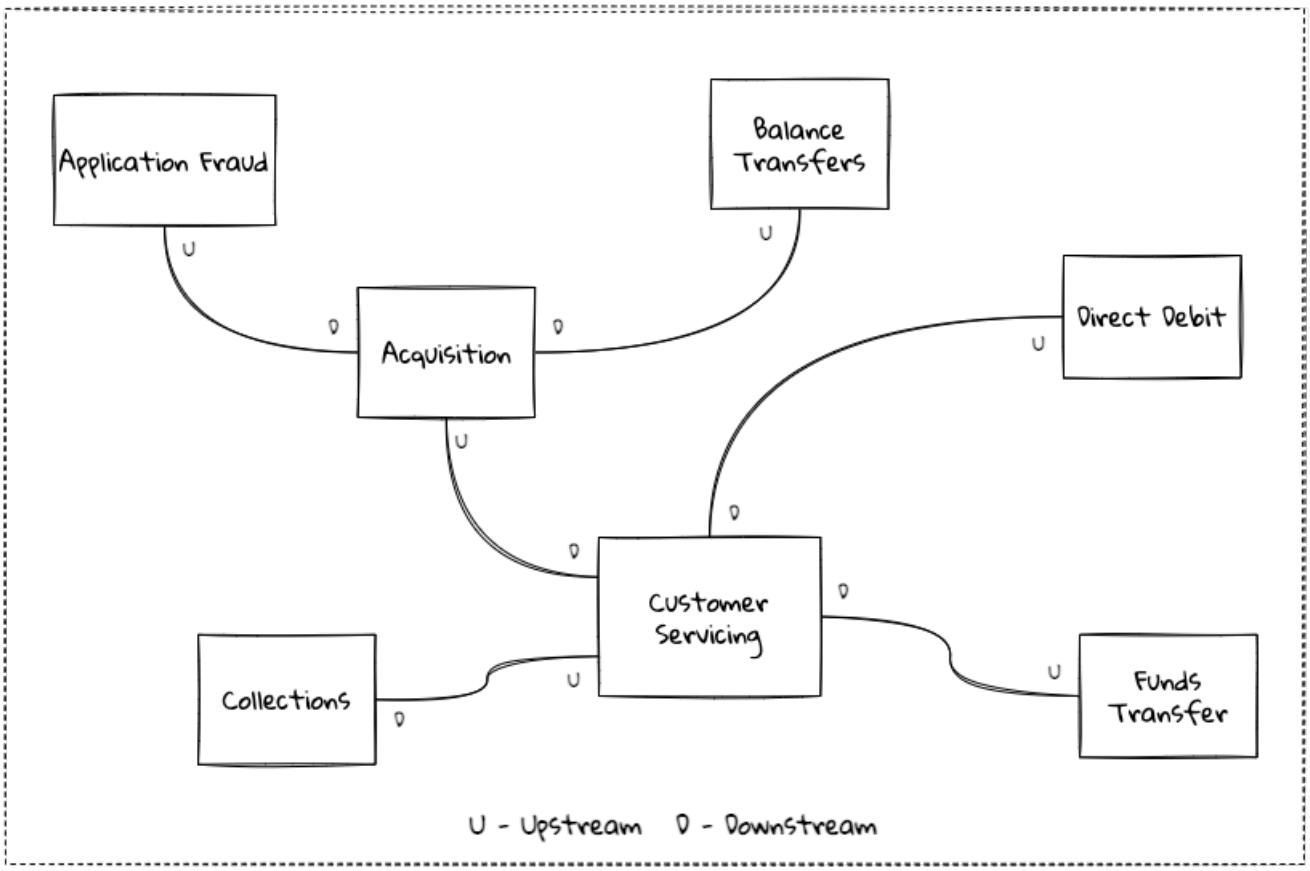


Figure 1- 19. Sample context map for Acme bank

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.

In subsequent chapters we will reinforce all the concepts introduced here in a lot more detail.

In the next section, we will look at why the ideas of DDD, introduced all those years ago, are still very relevant. If anything, we will look at why they are becoming even more relevant now than ever.

1.4.2. 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:

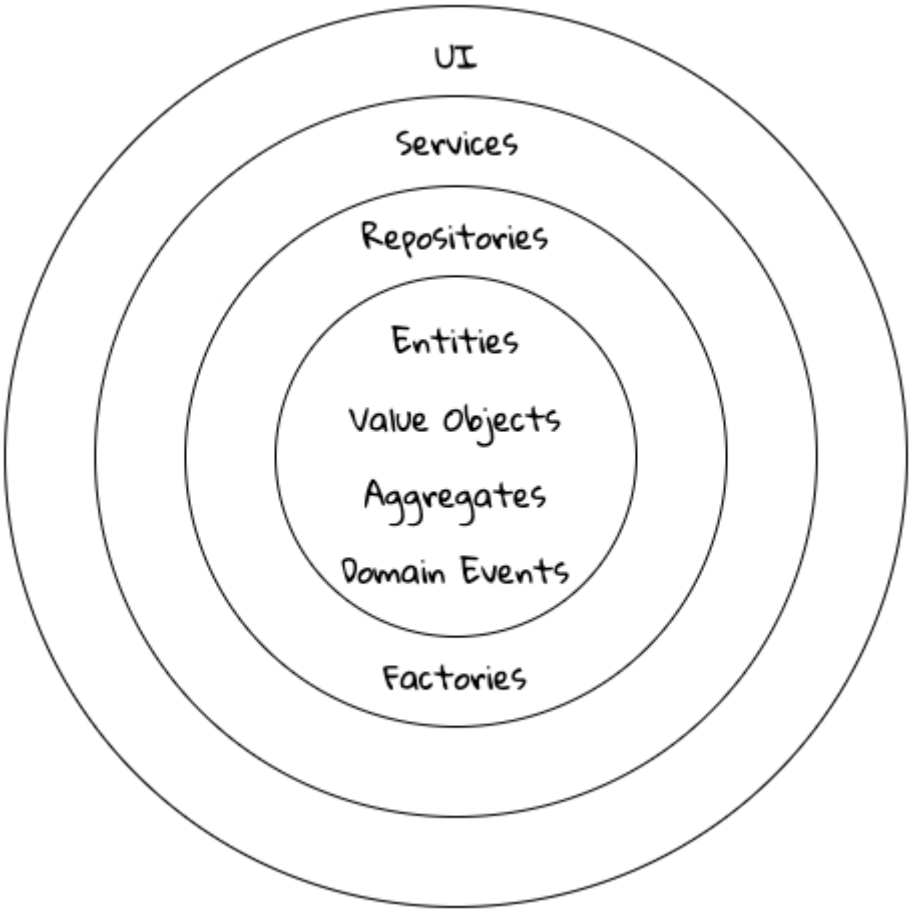


Figure 1- 20. The elements of DDD's tactical design

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:

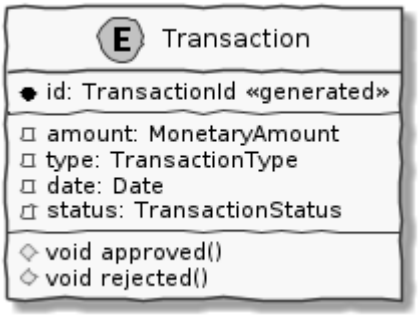


Figure 1- 21. A simple `MonetaryAmount` value object

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:

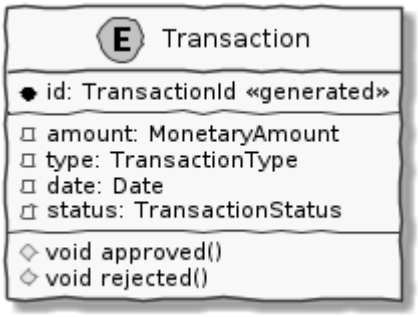


Figure 1- 22. A simple depiction of `Transaction` entity

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:

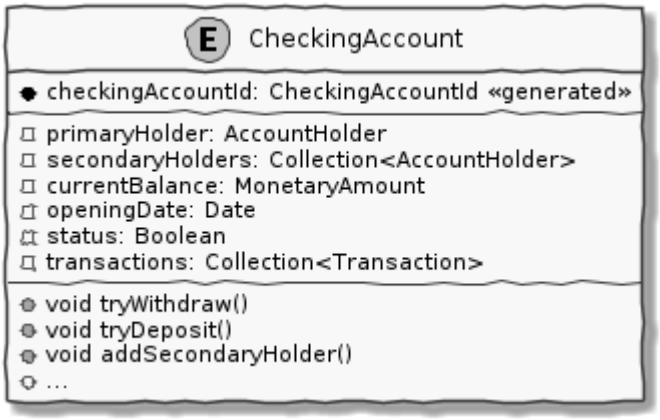


Figure 1- 23. A simple depiction of a `CheckingAccount` aggregate

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. could be classified as domain events. 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.

1.5. Why is DDD Relevant? Why Now?

> He who has a why to live for can bear almost anyhow.
>
> — Friedrich Nietzsche

In a lot of ways, domain-driven design was way ahead of its time when Eric Evans introduced the concepts and principles back in 2003. DDD seems to have gone from strength to strength. In this section, we will examine why DDD is even more relevant today, than it was when Eric Evans wrote his book on the subject way back in 2003.

1.5.1. Rise of Open Source

Eric Evans, during his keynote address at the Explore DDD conference in 2017, lamented about how difficult it was to implement even the simplest concepts like immutability in value objects when his book had released. In contrast though, nowadays, it's simply a matter of importing a mature, well documented, tested library like Project Lombok or Immutables to be productive, literally in a matter of minutes. To say that open source software has revolutionized the software industry would be an understatement! At the time of this writing, the public maven repository (https://mvnrepository.com) indexes no less than a staggering **18.3 million artifacts** in a large assortment of popular categories ranging from databases, language runtimes to test frameworks and many many more as shown in the chart below:

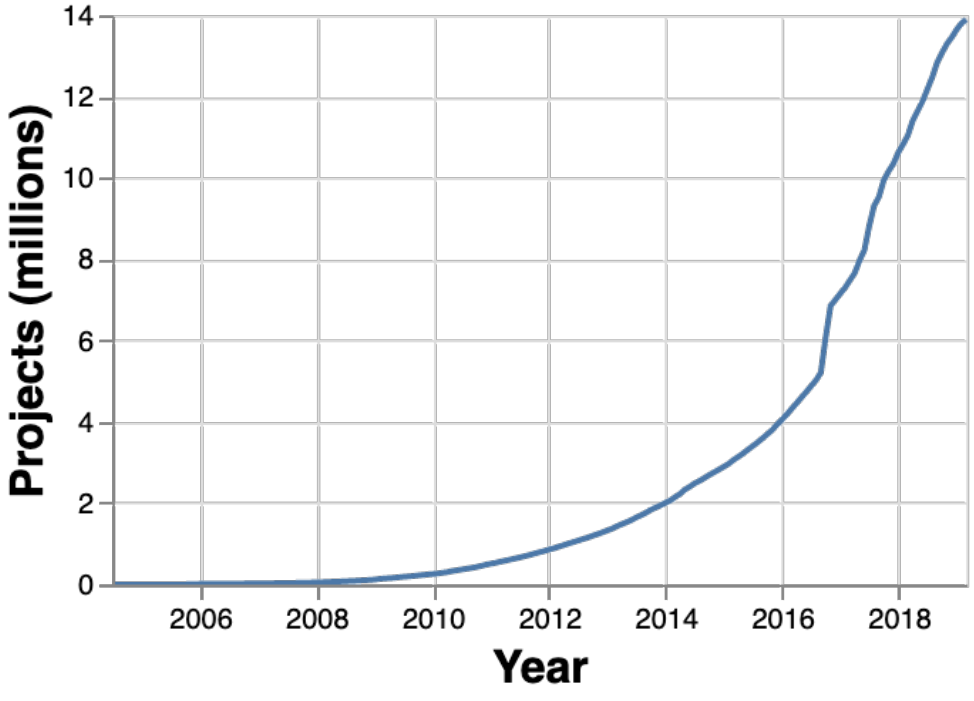


Figure 1- 24. Open source Java over the years. Source: https://mvnrepository.com/

Java stalwarts like the spring framework and more recent innovations like spring boot, quarkus, etc. make it a no-brainer to create production grade applications, literally in a matter of minutes. Furtheremore, frameworks like Axon, Lagom, etc. make it relatively simple to implement advanced architecture patterns such are CQRS, event sourcing, that are very complementary to implementing DDD-based solutions.

1.5.2. Advances in Technology

DDD by no means is just about technology, it could not be completely agnostic to the choices available at the time. 2003 was the heyday of heavyweight and ceremony-heavy frameworks like J2EE (Java 2 Enterprise Edition), EJBs (Enterprise JavaBeans), SQL databases, ORMs (Object Relational Mappers) and the like — with not much choice beyond that when it came to enterprise tools and patterns to build complex software — at least out in the public domain. The software world has evolved and come a very long way from there. In fact, modern game changers like Ruby on Rails and the public cloud were just getting released. In contrast though, we now have no shortage of application frameworks, NoSQL databases, programmatic APIs to create infrastructure components with a lot more releasing with monotonous regularity.

All these innovations allow for rapid experimentation, continuous learning and iteration at pace. These game changing advances in technology have also coincided with the exponential rise of the internet and ecommerce as viable means to carry out successful businesses. In fact the impact of the internet is so pervasive that it is almost inconceivable to launch businesses without a digital component being an integral component. Finally, the consumerization and wide scale penetration of smartphones, IoT devices and social media has meant that data is being produced at rates inconceivable as recent as a decade ago. This means that we are building for and solving the most complicated problems by several orders of magnitude.

1.5.3. Rise of Distributed Computing

There was a time when building large monoliths was very much the default. But an exponential rise in computing technology, public cloud, (IaaS, PaaS, SaaS, FaaS), big data storage and processing volumes, which has coincided with an arguable slowdown in the ability to continue creating faster CPUs, have all meant a turn towards more decentralized methods of solving problems.

[Hilbert InfoGrowth] |

https://upload.wikimedia.org/wikipedia/commons/7/7c/Hilbert_InfoGrowth.png

Figure 1- 25. Global Information Storage Capacity

Domain-driven design with its emphasis on dealing with complexity by breaking unwieldy monoliths into more manageable units in the form of subdomains and bounded contexts, fits naturally to this style of programming. Hence, it is no surprise to see a renewed interest in adopting DDD principles and techniques when crafting modern solutions. To quote Eric Evans, it is no surprise that Domain-Driven Design is even more relevant now than when it was originally conceived!

1.6. Summary

In this chapter we examined some common reasons for why software projects fail. We saw how inaccurate or misinterpreted requirements, architecture (or the lack thereof), excessive technical debt, etc. can get in the way of meeting business goals and success.

We looked at the basic building blocks of domain-driven design such as domains, subdomains, ubiquitous language, domain models, bounded contexts and context maps. We also examined why the principles and techniques of domain-driven design are still very much relevant in the modern age of microservices and serverless. You should now be able to appreciate the basic terms of DDD and understand why it is important in today's context.

In the next chapter we will take a closer look at the real-world mechanics of domain-driven design. We will delve deeper into the strategic and tactical design elements of DDD and look at how using these can help form the basis for better communication and create more robust designs.

1.7. Questions

1. What are the most common reasons for software projects to fail?
2. Why is DDD relevant in today's context?

1.8. Further Reading

Title	Author	Location
Pulse of the Profession - 2017	PMI	https://www.pmi.org/-/media/pmi/documents/public/pdf/learning/thought-leadership/pulse/pulse-of-the-profession-2017.pdf
Pulse of the Profession - 2020	PMI	https://www.pmi.org/learning/library/forging-future-focused-culture-11908
Project success: Definitions and Measurement Techniques	PMI	https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460
Project success: definitions and measurement techniques	JK Pinto, DP Slevin	https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460

Title	Author	Location
Analysis Paralysis	Ward Cunningham	https://proxy.c2.com/cgi/wiki?AnalysisParalysis
Big Design Upfront	Ward Cunningham	https://wiki.c2.com/?BigDesignUpFront
Enterprise Modeling Anti-Patterns	Scott W. Ambler	http://agilemodeling.com/essays/enterpriseModelingAntiPatterns.htm
A Project Manager's Guide To 42 Agile Methodologies	Henny Portman	https://thedigitalprojectmanager.com/agile-methodologies
Domain-Driven Design Even More Relevant Now	Eric Evans	https://www.youtube.com/watch?v=kIKwPNKXaLU
Introducing Deliberate Discovery	Dan North	https://dannorth.net/2010/08/30/introducing-deliberate-discovery/
No Silver Bullet — Essence and Accident in Software Engineering	Fred Brooks	http://faculty.salisbury.edu/~xswang/Research/Papers/SERelated/no-silver-bullet.pdf
Mastering Non-Functional Requirements	Sameer Paradkar	https://www.packtpub.com/product/mastering-non-functional-requirements/9781788299237
Big Ball Of Mud	Brian Foote & Joseph Yoder	http://www.laputan.org/mud/
The Forgotten Layer of the Test Automation Pyramid	Mike Cohn	https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid
Tech debt: Reclaiming tech equity	Vishal Dalal et al	https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/tech-debt-reclaiming-tech-equity
Is High Quality Software Worth the Cost	Martin Fowler	https://martinfowler.com/articles/is-quality-worth-cost.html#WeAreUsedToATrade-offBetweenQualityAndCost

[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

Chapter 2. Where and How Does DDD Fit?

> We won't be distracted by comparison if we are captivated with purpose.
>
> — Bob Goff

Software architecture refers to the fundamental structures of a software system and the discipline of creating such structures and systems. Over the years, we have accumulated a series of architecture styles and programming paradigms to help us deal with system complexity. In this chapter we will examine how DDD can be applied in a manner that is complementary to these architecture styles and programming paradigms. We will also look at how/where it fits in the overall scheme of things when crafting a software solution.

At the end of this chapter, you will gain an appreciation of a variety of architecture style and programming paradigms, along with some pitfalls to watch out for, when applying them. You will also understand the role that DDD plays in augmenting each of these.

2.1. Architecture Styles

Domain-driven design presents a set of architecture tenets in the form of the strategic and tactical design elements. This enables decomposing large, potentially unwieldy business subdomains into well-factored, independent bounded contexts. One of the great advantages of DDD is that it does not require the use of any specific architecture. However, the software industry has been using a plethora of architecture styles over a period of the last several years. Let's look at how DDD can be used in conjunction with a set of popular architecture styles to arrive at better solutions.

2.1.1. Layered Architecture

The layered architecture is one of the most common architecture styles where the solution is typically organized into four broad categories: **presentation**, **application**, **domain** and **persistence**. Each of the layers provides a solution to a particular concern it represents as shown here:

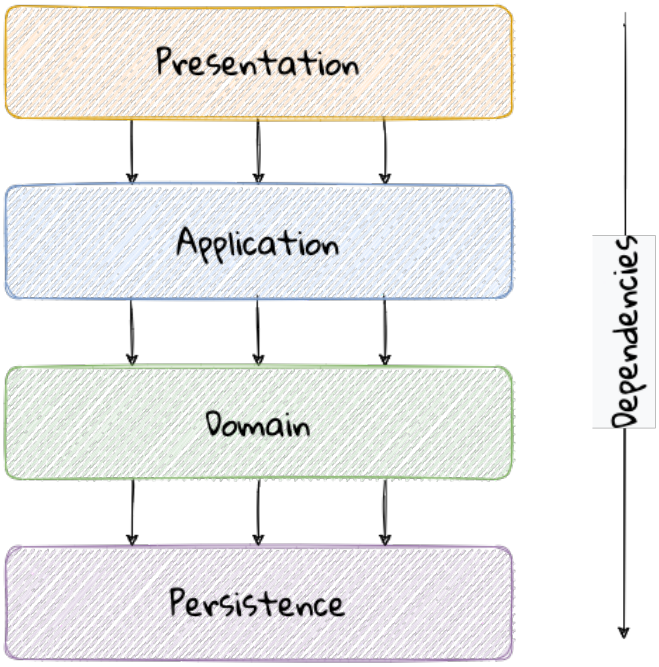


Figure 1- 26. Essence of a layered architecture.

The main idea behind the layered architecture is a separation of concerns — where the dependencies between layers are unidirectional (from the top to the bottom). For example, the domain layer can depend on the persistence layer, not the other way round. In addition, any given layer typically accesses the layer immediately beneath it without bypassing layers in between. For example, the presentation layer may access the domain layer only through the application layer.

This structure enables looser coupling between layers and allows them to evolve independently of each other. The idea of the layered architecture fits very well with domain-driven design's tactical design elements as depicted here:

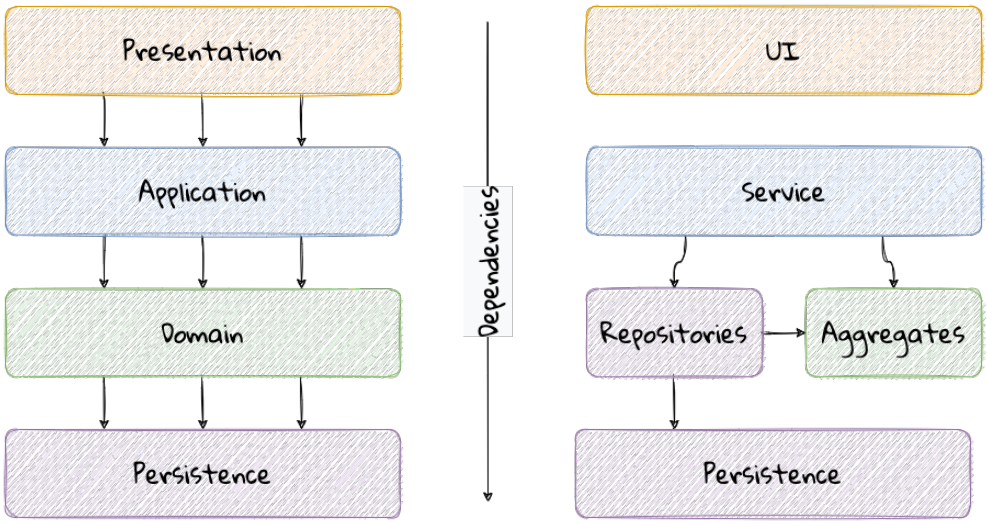


Figure 1- 27. Layered architecture mapped to DDD's tactical design elements.

DDD actively promotes the use of a layered architecture, primarily because it makes it possible to focus on the domain layer in isolation of other concerns like how to information gets displayed, how end-to-end flows are managed, how data is stored and retrieved, etc. From that perspective, solutions that apply DDD tend to naturally be layered as well.

Notable variations

A variation of the layered architecture was invented by Alistair Cockburn, which he originally called the *hexagonal architecture*[5] (alternatively called the ports and adapters architecture). The idea behind this style was to avoid inadvertent dependencies between layers (as could occur in the layered architecture), specifically between the core of the system and the peripheral layers. The main idea here is to make use of interfaces (*ports*) exclusively within the core to enable modern drivers such as testing and looser coupling. This allows the core to be developed and evolved independently of the non-core parts and the external dependencies. Integration with real-world components such as a database, file systems, web services, etc. is achieved through concrete implementations of the *ports* termed as *adapters*. The use of interfaces within the core enables much easier testing of the core in isolation of the rest of the system using mocks and stubs. It is also common to use dependency injection frameworks to dynamically swap out implementations of these interfaces when working with the real system in an end-to-end environment. A visual representation of the hexagonal architecture is shown here:

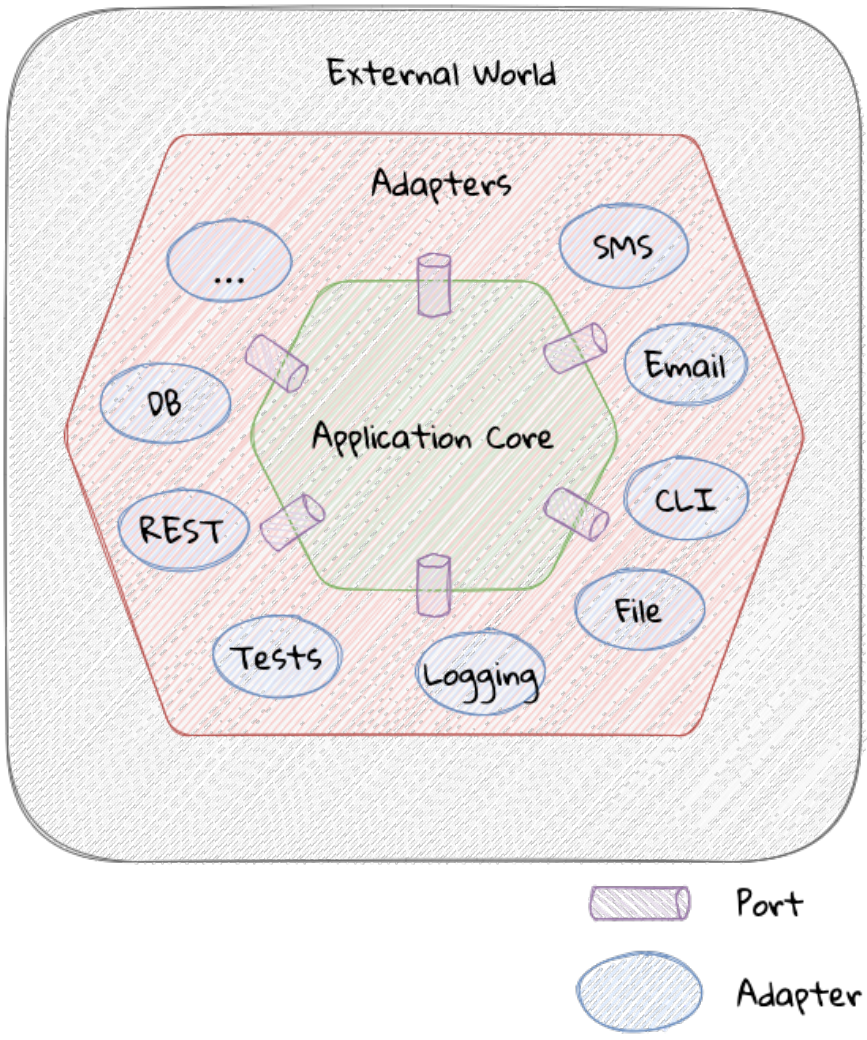


Figure 1- 28. Hexagonal architecture

It turns out that the use of the term hexagon in this context was purely for visual purposes — not to limit the system to exactly six types of ports.

Similar to the hexagonal architecture, the onion architecture[6], conceived by Jeffrey Palermo is based on creating an application based on an independent object model within the core that can be

compiled and run separately from the outer layers. This is done by defining interfaces (called ports in the hexagonal architecture) in the core and implementing (called adapters in the hexagonal architecture) them in the outer layers. From our perspective, the hexagonal and onion architecture styles have no perceptible differences that we could identify.

A visual representation of the onion architecture is shown here:

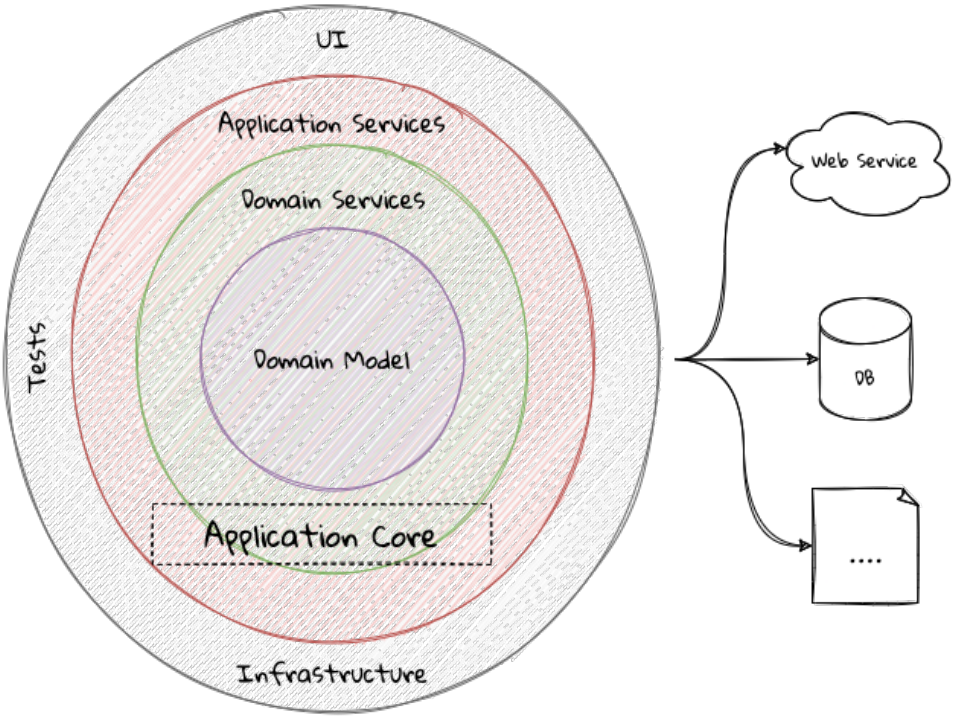


Figure 1- 29. Onion architecture

Yet another variation of the layered architecture, popularized by Robert C. Martin (known endearingly as Uncle Bob) is the clean architecture. This is based on adhering to the SOLID principles[7] also perpetrated by him. The fundamental message here (just like in the case of hexagonal and onion architecture) is to avoid dependencies between the core — the one that houses business logic and other layers that tend to be volatile (like frameworks, third-party libraries, UIs, databases, etc).

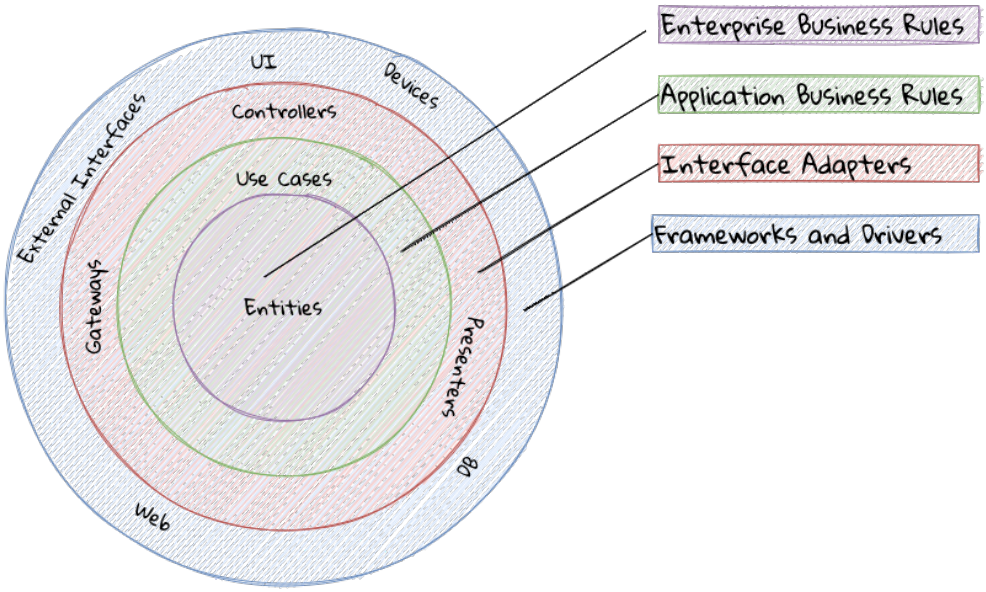


Figure 1- 30. Clean architecture

All these architecture styles are synergistic with DDD's idea of developing the domain model for the core subdomain (and by extension its bounded context) independently of the rest of the system.

While each of these architecture styles provide additional guidance in terms of how to structure a layered architecture, any architecture approach we choose, comes with its set of tradeoffs and limitations you will need to be cognizant of. We discuss some of these considerations here.

Considerations

Layer cake anti-pattern

Sticking to a fixed set of layers provides a level of isolation, but in simpler cases, it may prove overkill without adding any perceptible benefit other than adherence to an agreed on architectural guidelines. In the layer cake anti-pattern, each layer merely proxies the call to the layer beneath it without adding any value. The example below illustrates this scenario that is fairly common:

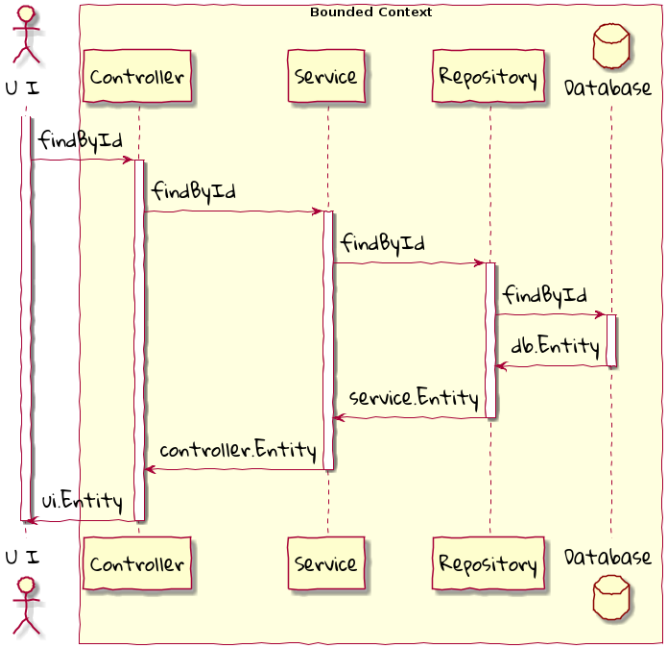


Figure 1- 31. Example of the **layer cake** *anti-pattern to find an entity representation by ID*

Here the `findById` method is replicated in every layer and simply calls the method with the same name in the layer below with no additional logic. This introduces a level of accidental complexity to the solution. Some amount of redundancy in the layering may be unavoidable for the purposes of standardization. It may be best to re-examine the layering guidelines if the *layer cake* occurs prominently in the codebase.

Anemic translation

Another variation of the layer cake we see commonly is one where layers refuse to share input and output types in the name of higher isolation and looser coupling. This makes it necessary to perform translations at the boundary of each layer. If the objects being translated are more or less structurally identical, we have an *anemic translation*. Let's look at a variation of the `findById` example we discussed above.

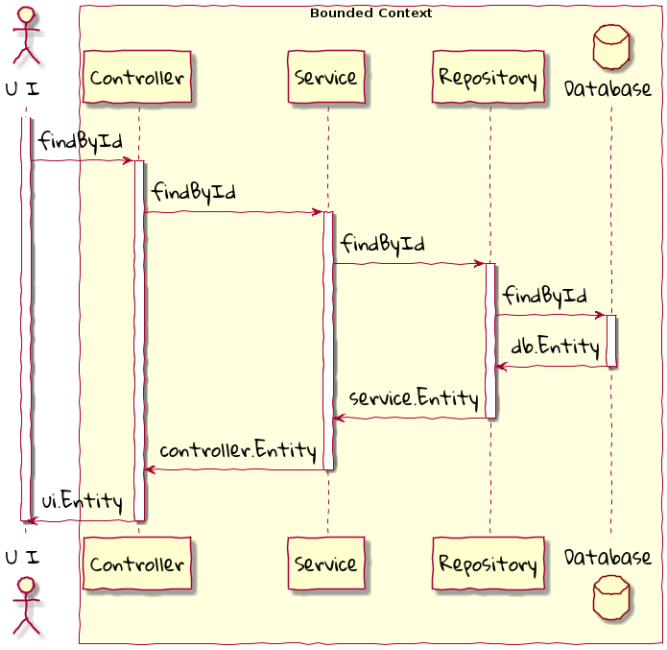


Figure 1- 32. Example of the **anemic translation** *anti-pattern to find an entity representation by ID*

In this case, each layer defines a `Entity` type of its own, requiring a translation between types at each layer. To make matters worse, the structure of the `Entity` type may have seemingly minor variations (for example, `lastName` being referred to as `surname`). While such translations may be necessary across bounded contexts, teams should strive to avoid the need for variations in names and structures of the same concept within a single bounded context. The intentional use of the **ubiquitous language** helps avoid such scenarios.

Layer bypass

When working with a layered architecture, it is reasonable to start by being strict about layers only interacting with the layer immediately beneath it. As we have seen above, such rigid enforcements may lead to an intolerable degree of accidental complexity, especially when applied generically to a large number of use-cases. In such scenarios, it may be worth considering consciously allowing one or more layers to be bypassed. For example, the `controller` layer may be allowed to work directly with the `repository` without using the `service` layer. In many cases, we have found it useful to use a separate set of rules for *commands* versus *queries* as a starting point.

This can be a slippery slope. To continue maintaining a level of sanity, teams should consider the use of a lightweight architecture governance tool like **ArchUnit**[8] to make agreements explicit and afford quick feedback. A simple example of how to use ArchUnit for this purpose is shown here:

```
class LayeredArchitectureTests {
    @ArchTest
    static final ArchRule layer_dependencies_are_respected_with_exception =
layeredArchitecture()

            .layer("Controllers").definedBy("..controller..")
            .layer("Services").definedBy("..service..")
            .layer("Domain").definedBy("..domain..")
            .layer("Repository").definedBy("..repository..")

            .whereLayer("Controllers").mayNotBeAccessedByAnyLayer()
            .whereLayer("Services").mayOnlyBeAccessedByLayers("Controllers")
            .whereLayer("Domain").mayOnlyBeAccessedByLayers("Services", "Repository",
"Controllers")
            .whereLayer("Repository")
                .mayOnlyBeAccessedByLayers("Services", "Controllers"); ①
}
```

① The Repository layer can be accessed by both the Services and Controllers layers — effectively allowing Controllers to bypass the use of the Services layer.

2.1.2. Vertical slice architecture

The layered architecture and its variants described above, provide reasonably good guidance on how to structure complex applications.The vertical slice architecture championed by Jimmy Boggard recognizes that it may be too rigid to adopt a standard layering strategy for all use cases across the entire application.Furthermore, it is important to note that business value cannot be derived by implementing any of these horizontal layers in isolation.Doing so will only result in unusable inventory and lots of unnecessary context switching until all these layers are connected.Therefore, the vertical slice architecture proposes *minimizing coupling between slices, and maximizing coupling in a slice*[9] as shown here:

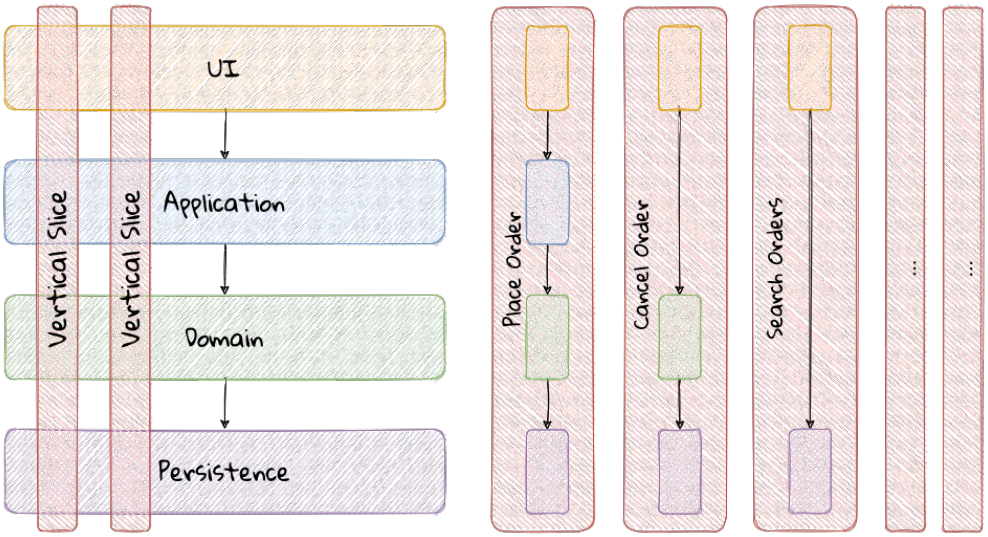


Figure 1- 33. Vertical slice architecture

In the example above, *place order* might require us to coordinate with other components through

the application layer, apply complex business invariants while operating within the purview of an ACID transaction.Similarly, *cancel order* might require applying business invariants within an ACID transaction without any additional coordination — obviating the need for the application layer in this case.However, *search orders* might require us to simply fetch existing data from a query optimized view.This style makes use of a horses for courses approach to layering that may help alleviate some anti-patterns listed above when implementing a plain vanilla layered architecture.

Considerations

The vertical slice architecture affords a lot of flexibility when implementing a solution — taking into consideration the specific needs of the use-case being implemented. However, without some level of governance, this may quickly devolve to the big ball of mud with layering decisions being made seemingly arbitrarily based on personal preferences and experiences (or lack thereof). As a sensible default, you may want to consider using a distinct layering strategy for commands and queries. Beyond that, non-functional requirements may dictate how you may need to deviate from here. For example, you may need to bypass layers to meet performance SLAs for certain use cases.

When used pragmatically, the vertical slice architecture does enable applying DDD very effectively within each or a group of related vertical slices — allowing them to be treated as bounded contexts. We show two possibilities using the *place order* and *cancel order* examples here:

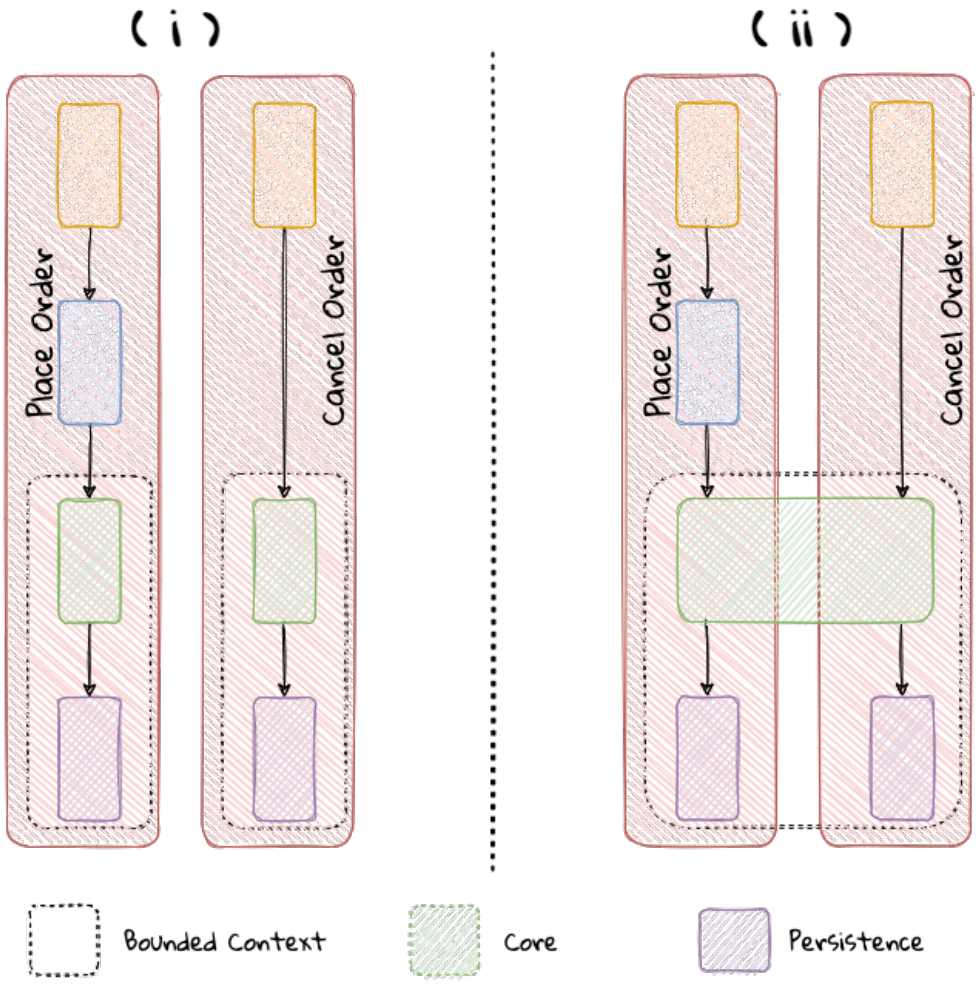


Figure 1- 34. Vertical slices used to evolve bounded contexts

In example (i) above, *place order* and *cancel order*, each use a distinct domain model, whereas in

example (ii), both use cases share a common domain model and by extension become part of the same bounded context. This does pave the way to slice functionality when looking to adopt the serverless architecture along use case boundaries.

2.1.3. Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is an architectural style where software components expose (potentially) reusable functionality over standardized interfaces. The use of standardized interfaces (such as SOAP, REST, gRPC, etc. to name a few) enables easier interoperability when integrating heterogeneous solutions as shown here:

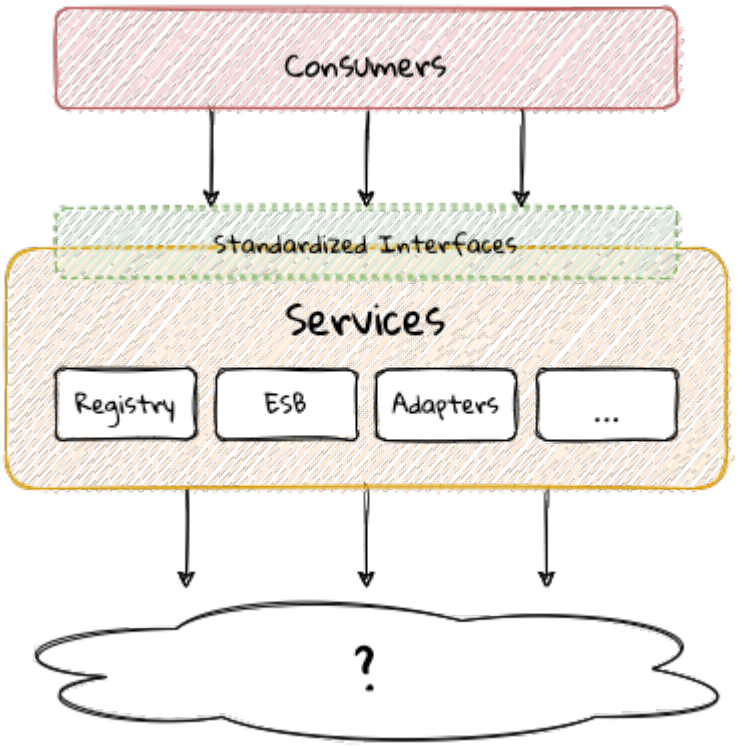


Figure 1- 35. SOA: Expose reusable functionality over standard interfaces.

Previously, the use of non-standard, proprietary interfaces made this kind of integration a lot more challenging. For example, a retail bank may expose inter-account transfer functionality in the form of SOAP web services. While SOA prescribes exposing functionality over standardized interfaces, the focus is more on integrating heterogeneous applications than on implementing them.

Considerations

At one of the banks we worked at, we exposed a set of over 500 service interfaces over SOAP. Under the covers, we implemented these services using EJB 2.x (a combination of stateless session beans and message-driven beans) hosted on a commercial J2EE application server which also did double duty as an enterprise service bus (ESB). These services largely delegated most if not all the logic to a set of underlying stored procedures within a single monolithic Oracle database using a canonical data model for the entire enterprise! To the outside world, these services were *location transparent*, stateless, *composable* and *discoverable*. Indeed, we advertised this implementation as an example of SOA, and it would be hard to argue that it was not.

This suite of services had evolved organically over the years with no explicit boundaries, concepts

from various parts of the organization and generations of people mixed in, each adding their own interpretation of how business functionality needed to be implemented. In essence, the implementation resembled the dreaded big ball of mud which was extremely hard to enhance and maintain.

The intentions behind SOA are noble. However, the promises of reuse, loose coupling are hard to achieve in practice given the lack of concrete implementation guidance on component granularity. It is also true that SOA means many things[10] to different people. This ambiguity leads to most SOA implementations becoming complex, unmaintainable monoliths, centered around technology components like a service bus or the persistence store or both. This is where using DDD to solve a complex problem by breaking it down into subdomains and bounded contexts can be invaluable.

2.1.4. Microservices architecture

In the last decade or so, microservices have gained quite a lot of popularity with lots of organizations wanting to adopt this style of architecture. In a lot of ways, microservices are an extension of service-oriented architectures — one where a lot of emphasis is placed on creating focused components that deal with doing a limited number of things and doing them right. Sam Newman, the author of the *Building Microservices* book defines microservices as *small*-sized, independently deployable components that maintain their own state and are **modeled around a business domain**. This affords benefits such as adopting a horses for courses approach when modeling solutions, limiting the blast radius, improved productivity and speed, autonomous cross-functional teams, etc. Microservices usually exist as a collective, working collaboratively to achieve the desired business outcomes, as depicted here:

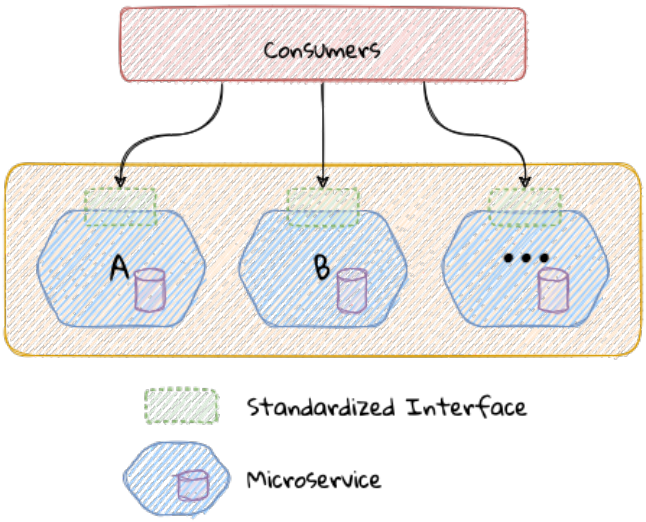


Figure 1- 36. A microservices ecosystem

As we can see, SOA and microservices are very similar from the perspective of the consumers in that they access functionality through a set of standardized interfaces. The microservices approach is an evolution of SOA in that the focus now is on building smaller, self-sufficient, independently deployable components with the intent of avoiding single points of failure (like an enterprise database or service bus), which was fairly common with a number of SOA-based implementations.

Considerations

While microservices have definitely helped, there still exists quite a lot of ambiguity when it comes

to answering how big or small[11] a microservice should be. Indeed, a lot of teams seem to struggle to get this balance right, resulting in a distributed monolith[12] — which in a lot of ways can be much worse than even the single process monolith from the SOA days. Again, applying the strategic design concepts of DDD can help create independent, loosely coupled components, making it an ideal companion for the microservices style of architecture.

2.1.5. Event-Driven Architecture (EDA)

Irrespective of the granularity of components (monolith or microservices or something in between), most non-trivial solutions have a boundary, beyond which there may be a need to communicate with external system(s). This communication usually happens through the exchange of messages between systems, causing them to become coupled with each other. Coupling comes in two broad flavors: *afferent* — who depends on you and *efferent* — who you depend on. Excessive amounts of efferent coupling can make systems very brittle and hard to work with.

Event-driven systems enable authoring solutions that have a relatively low amount of efferent coupling by emitting events when they attain a certain state without caring about who consumes those events. In this regard, it is important to differentiate between message-driven and event-driven systems as mentioned in the *Reactive Manifesto*:

Message-driven versus Event-driven

> A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted. This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.
>
> — Reactive Manifesto

In simpler terms, event-driven systems do not care who the downstream consumers are, whereas in a message-driven system that may not necessarily be true. When we say event-driven in the context of this book, we mean the former.

Typically, event-driven systems eliminate the need for point-to-point messaging with the ultimate consumers by making use of an intermediary infrastructure component usually known as a message broker, event bus, etc. This effectively reduces the efferent coupling from *n* consumers to 1. There are a few variations on how event-driven systems can be implemented. In the context of publishing events, Martin Fowler talks about two broad styles (among other things) — event notifications and event-carried state transfer in his What do you mean by "event-driven"?[13] article.

Considerations

One of the main trade-offs when building an event-driven system is to decide the amount of state (payload) that should be embedded in each event. It may be prudent to consider embedding just enough state indicating changes that occurred as a result of the emitted event to keep the various opposing forces such as producer scaling, encapsulation, consumer complexity, resiliency, etc. We will discuss the related implications in more detail when we cover implementing events in Chapter 5.

Domain-driven design is all about keeping complexity in check by creating these independent bounded contexts. However, independent does not mean isolated. Bounded contexts may still need to communicate with each other. One way to do that is through the use of a fundamental DDD building block — domain events. Event-driven architecture and DDD are thus complementary. It is typical to make use of an event-driven architecture to allow bounded contexts to communicate while continuing to loosely coupled with each other.

2.1.6. Command Query Responsibility Segregation (CQRS)

In traditional applications, a single domain, data/persistence model is used to handle all kinds of operations. With CQRS, we create distinct models to handle updates (commands) and enquiries. This is depicted in the following diagram:

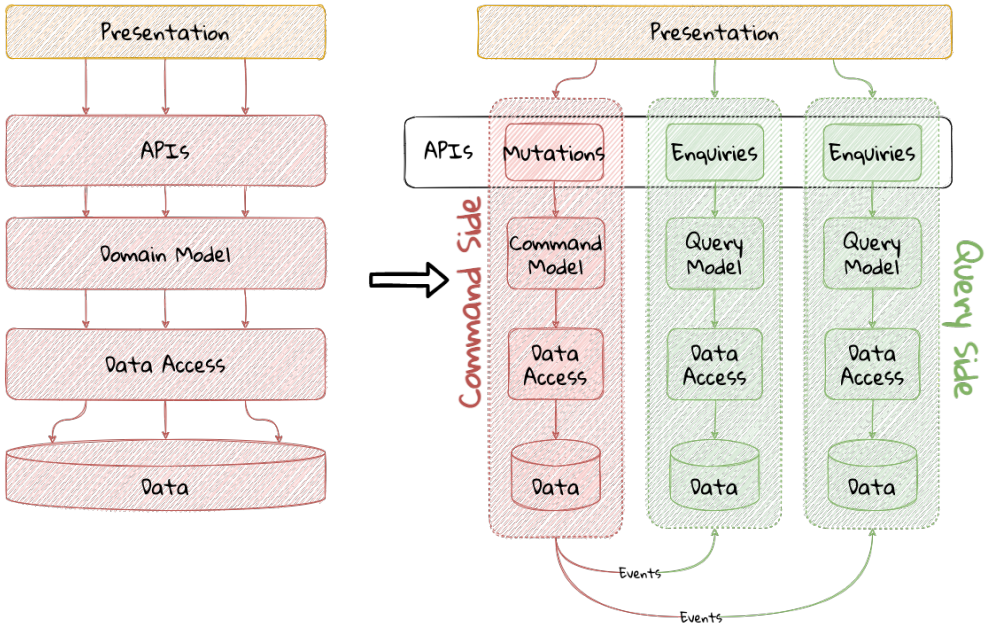


Figure 1- 37. Traditional versus CQRS Architecture

We depict multiple query models above because it is possible (but not necessary) to create more than one query model, depending on the kinds of query use cases that need to be supported.

For this to work predictably, the query model(s) need to be kept in sync with the write models (we will examine some of the techniques to do that in detail later.

Considerations

The traditional, single-model approach works well for simple, CRUD-style applications, but starts to become unwieldy for more complex scenarios. We discuss some of these scenarios below:

Volume imbalance between read and writes

In most systems, read operations often outnumber write operations by significant orders of magnitude. For example, consider the number of times a trader checks stock prices vs. the number of times they actually transact (buy or sell stock trades). It is also usually true that write operations are the ones that make businesses money. Having a single model for both reads and writes in a system with a majority of read operations can overwhelm a system to an extent where write performance can start getting affected.

Need for multiple read representations

When working with relatively complex systems, it is not uncommon to require more than one representation of the same data.For example, when looking at personal health data, one may want to look at a daily, weekly, monthly view.While these views can be computed on the fly from the *raw* data, each transformation (aggregation, summarization, etc.) adds to the cognitive load on the system.Several times, it is not possible to predict ahead of time, the nature of these requirements.By extension, it is not feasible to design a single canonical model that can provide answers to all these requirements.Creating domain models specifically designed to meet a focused set of requirements can be much easier.

Different security requirements

Managing authorization and access requirements to data/APIs when working a single model can start to become cumbersome. For example, higher levels of security may be desirable for debit operations in comparison to balance enquiries.Having distinct models can considerably ease the complexity in designing fine-grained authorization controls.

More uniform distribution of complexity

Having a model dedicated to serve only command-side use cases means that they can now be focused towards solving a single concern.For query-side use cases, we create models as needed that are distinct from the command-side model. This helps spread complexity more uniformly over a larger surface area — as opposed to increasing the complexity on the single model that is used to serve all use cases.It is worth noting that the essence of domain-driven design is mainly to work effectively with complex software systems and CQRS fits well with this line of thinking.

When working with a CQRS based architecture, choosing the persistence mechanism for the command side is a key decision. When working in conjunction with an event-driven architecture, one could choose to persist aggregates as a series of events (ordered in the sequence of their occurrence). This style of persistence is known as event sourcing. We will cover this in more detail in Chapter 5 in the section on event-sourced aggregates.

2.1.7. Serverless Architecture

Serverless architecture is an approach to software design that allows developers to build and run services without having to manage the underlying infrastructure. The advent of AWS Lambda service has popularized this style of architecture, although several other services (like S3 and DynamoDB for persistence, SNS for notifications, SQS for message queuing etc.) have existed long before Lambda was launched.While AWS Lambda provided a compute solution in the form of Functions-as-a-Service (FaaS), these other services are just as essential, if not more, in order to benefit from the serverless paradigm.

In conventional DDD, bounded contexts are formed by grouping related operations around an aggregate, which then informs how the solution is deployed as a unit — usually within the confines of a single process. With the serverless paradigm, each operation (task) is expected to be deployed as an independent unit of its own. This requires that we look at how we model aggregates and bounded contexts differently — now centered around individual tasks or functions as opposed to a group of related tasks.

Does that mean that the principles of DDD to arrive at a solution do not apply anymore?While serverless introduces an additional dimension of having to treat finely-grained deployable units as first-class citizens in the modeling process, the overall process of applying DDD's strategic and tactical design continue to apply. We will examine this in more detail in Chapter 11 when we refactor the solution we build throughout this book to employ a serverless approach.

2.1.8. Big ball of mud

Thus far, we have examined a catalog of named architecture styles along with their pitfalls and how applying DDD can help alleviate them.On the other extreme, we may encounter solutions that lack a perceivable architecture, infamously termed as the *big ball of mud*.

> A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well-defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.
>
> — Brian Foote and Joseph Yoder

Although Foote and Yoder advise avoiding this style of architecture at all costs, software systems that resemble the big ball of mud continue to be a day-to-day inevitability for a lot of us. The

strategic and tactical design elements of DDD provide a set of techniques to help deal with and recover from these near-hopeless situations in a pragmatic manner without potentially having to adopt a big bang approach. Indeed, the focus of this book is to apply these principles to prevent or at least delay further devolution towards the big ball of mud.

2.1.9. Which architecture style should you use?

As we have seen, there are a variety of architecture styles one can lean on to when crafting a software solution. A lot of these architecture styles share quite a few common tenets. It can become difficult to claim conformance to any single architecture style. DDD, with its emphasis on breaking down complex business problems into subdomains and bounded contexts, enables the use of more than one approach across bounded contexts. We would like to make a special mention of the vertical slice architecture because it places an emphasis on dividing functionality along specific business outcomes and thus more naturally to DDD's ideas of subdomains and bounded contexts. In reality, one may find the need to extend and even deviate from pedantic definitions of architecture styles in order to meet real-world needs. But when we do make such compromises, it is important to do so **intentionally** and make it unambiguously clear why we are making such a decision (preferably using some lightweight mechanism such as ADRs[14]). This is important because it may become hard to justify this to others and even ourselves when we look at it in the future.

In this section, we have examined popular architecture styles and how we can amplify their effectiveness when used in conjunction with DDD. Now let's look at how DDD can complement the use of existing programming paradigms.

2.2. Programming paradigms

The tactical elements of DDD introduce a specific vocabulary (aggregates, entities, value objects, repositories, services, factories, domain events, etc.) when arriving at a solution. At the end of the day, we need to translate these concepts into running software. Over the years, we have employed a variety of programming paradigms including procedural, object-oriented, functional, aspect-oriented, etc. Is it possible to apply DDD in conjunction with one or more of these paradigms? In this section, we will explore how some common programming paradigms and techniques help us express the tactical design elements in code.

2.2.1. Object-oriented programming

On the surface of it, DDD seems to simply replicate a set of OO terms and call them using different names. For example, the central concepts of tactical DDD such as `aggregates`, `entities` and `value objects` could simply be referred to as objects in OO terms. Others like `services` may not have a direct OO analog. So how does one apply DDD in an object-oriented world? Let's look at a simple example:

```
interface PasswordService {
    String generateStrongPassword();
    boolean isStrong(String password);
    boolean isWeak(String password);
}

class PasswordClient {
    private PasswordService service;

    void register(String userEnteredPassword) {
        if (service.isStrong(userEnteredPassword)) {
            //...
        }
    }
}
```

OO purists will be quick to point out that the `PasswordService` is procedural and that a `Password` class might be needed to encapsulate related behaviours. Similarly, DDD enthusiasts might point out that this is an anemic domain model implementation. An arguably better object-oriented version might look something like:

```
class Password {
    private final String password;

    private Password(String password) {
        this.password = password;
    }

    public boolean isStrong() { ... }
    public boolean isWeak() { ... }
    public static Password generateStrongPassword() { ... }
    public static Password passwordFrom(String password) { ... }

}

interface PasswordService {
    Password generateStrongPassword();
    Password createPasswordFrom(String userEntered);
}

class PasswordClient {
    private PasswordService service;

    void register(String userEnteredPassword) {
        Password password = service.createPasswordFrom(userEnteredPassword);
        if (password.isStrong()) {
            // ...
        }
    }
}
```

In this case, the `Password` class stops exposing its internals and exposes the idea of a strong or weak password in the form of behavior (the `isStrong` and `isWeak` methods). From an OO perspective, the second implementation is arguably superior. If so, shouldn't we be using the object-oriented version at all times? As it turns out, the answer is nuanced and depends on what the consumers desire and the ubiquitous language used in that context. If the concept of the `Password` is in common usage within the domain, it perhaps warrants introducing such a concept in the implementation as well. If not, the first solution might suffice even though it seems to violate OO principles of encapsulation.

Our default position is to apply good OO practices as a starting point. However, it is more important to mirror the language of the domain as opposed to applying OO in a dogmatic manner. So we will be willing to compromise on OO purity if it appears unnatural to do so in that context. As mentioned earlier, clearly communicating the rationale for such decisions can go a long way.

2.2.2. Functional programming

Functions are a fundamental building block to code organization that exist in all higher order programming languages. Functional programming is a programming paradigm where programs are constructed by applying and composing functions. This is in contrast to imperative

programming that uses statements to change a program's state. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming. Pure functional programming completely prevents side effects and forces immutability. Embracing a functional style when designing a domain model to be more declarative and express intent a lot more clearly while remaining terse. It also allows us to keep complexity in check by enabling us to compose more complex concepts by using simpler ones.The functional implementation allows us to use a language closer to the problem domain, while having the added benefit of also being terse. Consider a simple example where we need to find the item with the least inventory across all our warehouses using a functional style as shown here:

Functional example

```
class Functional {
    public static Optional<Item> scarcestItem(Warehouse... warehouses) {
        return Stream.of(warehouses)
                .flatMap(Warehouse::items)
                .collect(groupingBy(Item::name, summingInt(Item::quantity)))
                .entrySet().stream()
                .map(Item::new)
                .min(comparing(Item::quantity));
    }
}
```

The imperative style shown here does get the job done, but is arguably a lot more verbose and harder to follow, sometimes even for technical team members!

Imperative example

```
class Imperative {
    public static Optional<Item> scarcestItem(Warehouse... warehouses) {
        Collection<Item> allItems = new ArrayList<>();
        for (Warehouse warehouse : warehouses) {
            allItems.addAll(warehouse.getItems());
        }
        Map<String, Integer> itemNamesByQuantity = new HashMap<>();
        for (Item item : allItems) {
            final String name = item.name();
            final int quantity = item.quantity();
            if (itemNamesByQuantity.containsKey(name)) {
                itemNamesByQuantity.put(name, itemNamesByQuantity.get(name) +
quantity);
            } else {
                itemNamesByQuantity.put(name, quantity);
            }
        }
        final Map.Entry<String, Integer> min =
            Collections.min(itemNamesByQuantity.entrySet(), Map.Entry.
comparingByValue());
        return min != null ? Optional.of(new Item(min)) : Optional.empty();

    }
}
```

From a DDD perspective, this yields a few benefits:

- **Increase collaboration with domain experts** because the declarative style allows placing a bigger focus on the what, rather than the how. This makes it a lot less intimidating to technical and non-technical stakeholders alike to work with on an ongoing basis.
- **Better testability**: because the use of pure functions (those that are side effect free) makes it easier to create data-driven tests. This has also afforded us an additional benefit of less mocking/stubbing. These characteristics make tests that are a lot easier to maintain and reason about. This has the benefit of allowing even technical team members to visualize corner cases a lot earlier in the process.

2.2.3. Which paradigm should you choose?

DDD simply states that you should build your software around a domain model that represents the actual problem that the software is trying to solve. When encountered with complex real-life problems, often we will find it hard to conform to any single paradigm across the board. Looking to use a one-size-fits-all approach may work to one's detriment. Our experience indicates that we will need to make use of a variety of techniques in order to solve the problem at hand elegantly. Java is inherently an object-oriented language. But with the advent of Java 8, it has started to embrace a variety of functional constructs as well. This allows us to make use of a multitude of techniques to create elegant solutions. The most important thing is to agree on the ubiquitous language and allow it to guide the approach taken. It also largely depends on the talent and experience one has at their

disposal. Making use of a style that is foreign to a majority of the team will likely prove counter-productive. Although we haven't covered the procedural paradigm here in this text, there may be occasions where it might be the best solution given the current situation. As long as we are intentional about areas where we deviate from the accepted norm for a particular programming paradigm, we should be in a reasonably good place.

2.3. Summary

In this chapter, we covered a series of commonly used architecture patterns and how we can practice DDD when working with them. We also looked at common pitfalls and gotchas that one may need to be cognizant of when using these architectures. We also looked at popular programming paradigms and their influence on the tactical elements of DDD.

Additionally, you should have an appreciation of the various architecture styles that we need to employ when coming up with a solution. In addition, you should have an understanding of how DDD can play a role no matter which style of architecture you choose to adopt.

In the next section, we will look to apply all the learnings in this and previous chapters against a real-world business use case. We will apply both the strategic and tactical patterns of DDD to break a complex domain into subdomains, bounded contexts and iteratively build a solution using technologies that are based on the Java programming language.

2.4. Questions

1. What architecture style(s) are you using in your current ecosystem? Are you seeing any of the common pitfalls that we have covered in this chapter?
2. Do you see merit in employing a hybrid of the architecture approaches covered here?
3. What programming paradigms are you employing in your current ecosystem?
4. Are there areas where you have had to violate principles that are strictly aligned with any given paradigm or approach? Are these deviations well understood and documented?

[5] https://alistair.cockburn.us/hexagonal-architecture/

[6] https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[7] https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

[8] https://www.archunit.org/

[9] https://jimmybogard.com/vertical-slice-architecture/

[10] https://martinfowler.com/bliki/ServiceOrientedAmbiguity.html

[11] https://martinfowler.com/articles/microservices.html#HowBigIsAMicroservice

[12] https://www.infoq.com/news/2016/02/services-distributed-monolith/

[13] https://martinfowler.com/articles/201701-event-driven.html

[14] https://www.thoughtworks.com/de-de/radar/techniques/lightweight-architecture-decision-records

Part 2: Implementing DDD in the real world

In the first section of this book, we looked at the mechanics of domain driven design and how it fits in the context of commonly used architecture styles and programming paradigms. In this section, we will implement a real-world application employing a bunch of techniques and practices that enable us to apply the tenets of DDD's strategic and tactical design.

Chapter 3. Understanding the domain

> A spoon does not know the taste of soup, nor a learned fool the taste of wisdom.
>
> — Welsh Proverb

3.1. Introduction

In this chapter, we will introduce a fictitious organization named KP Bank that is looking to modernize its product offerings in the international trade business. In order to establish a business strategy that sets them up for sustained success in the medium to long term, we will employ a series of techniques and practices to help expedite their path from strategy to execution.

At the end of this chapter, you will gain an appreciation of how to employ techniques like the business value canvas and the lean canvas to establish a sound understanding of the business strategy. Furthermore, we will examine how plotting an impact map will allow us to correlate business deliverables to goals. Finally, the Wardley mapping exercise will allow to establish the importance of our business decisions in relation to our competitive landscape.

At the outset, let's gain a high level understanding of KB Bank's business domain before we start diving deeper.

3.2. The domain of international trade

In many countries, international trade represents a significant portion of the gross domestic product (GDP) — making an exchange of capital, goods, and services between untrusted parties spread across the globe a necessity. While economic organizations such as the World Trade Organization (WTO) were formed specifically to ease and facilitate this process, differences in factors such as economic policy, trade laws, currency, etc. ensure that carrying out trade internationally can be a complex process with several entities involved across countries. Letter of Credit exist to simplify this process. Let's take a look at how they work.

3.3. International trade at KP Bank

Kosmo Primo (KP) Bank has been in business for the last several years and has been focusing on providing a variety of banking solutions such as retail, corporate, securities and other products. They have been steadily expanding operations to other countries and continents. This has allowed them to expand their international trade business significantly in the last decade or so. While they have been among the leaders in this space, the recent onset of new digital-native competitors has started to eat into their business and impacting their top-line adversely. Customers are complaining that the process is too cumbersome, time-consuming and lately unreliable. In addition, due to a very inefficient, manual process that is currently in place, KP Bank has been finding it very hard to keep a check on costs. Just in the last three years, they have had to increase transaction processing costs by around fifty percent! Not surprisingly, this has coincided with plummeting customer satisfaction — which is evidenced by the fact that the number of customers serviced has remained

flat over the intervening time.

The CIO has recognized that there is a need to look at this problem afresh and come up with a strategy that sets KP Bank up for sustained success for the next several years and re-establish them as one of the leaders in the international trade business.

3.4. Understanding international trade strategy at KP Bank

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.

It is pertinent to note that these tools were conceived independently, but when practiced in conjunction with other DDD techniques can accentuate the effectiveness of the overall process and solution. The use of these should be considered to be complementary in your DDD journey.

Let's look at some of the most populars techniques we have employed to quickly gain understanding of the business problem and propose solutions.

3.4.1. 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, originally conceived by Swiss consultant Alexander Osterwalder as part of his PhD thesis, 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?

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. Here is an attempt we made to create the business model canvas for the international trade business at KP Bank:

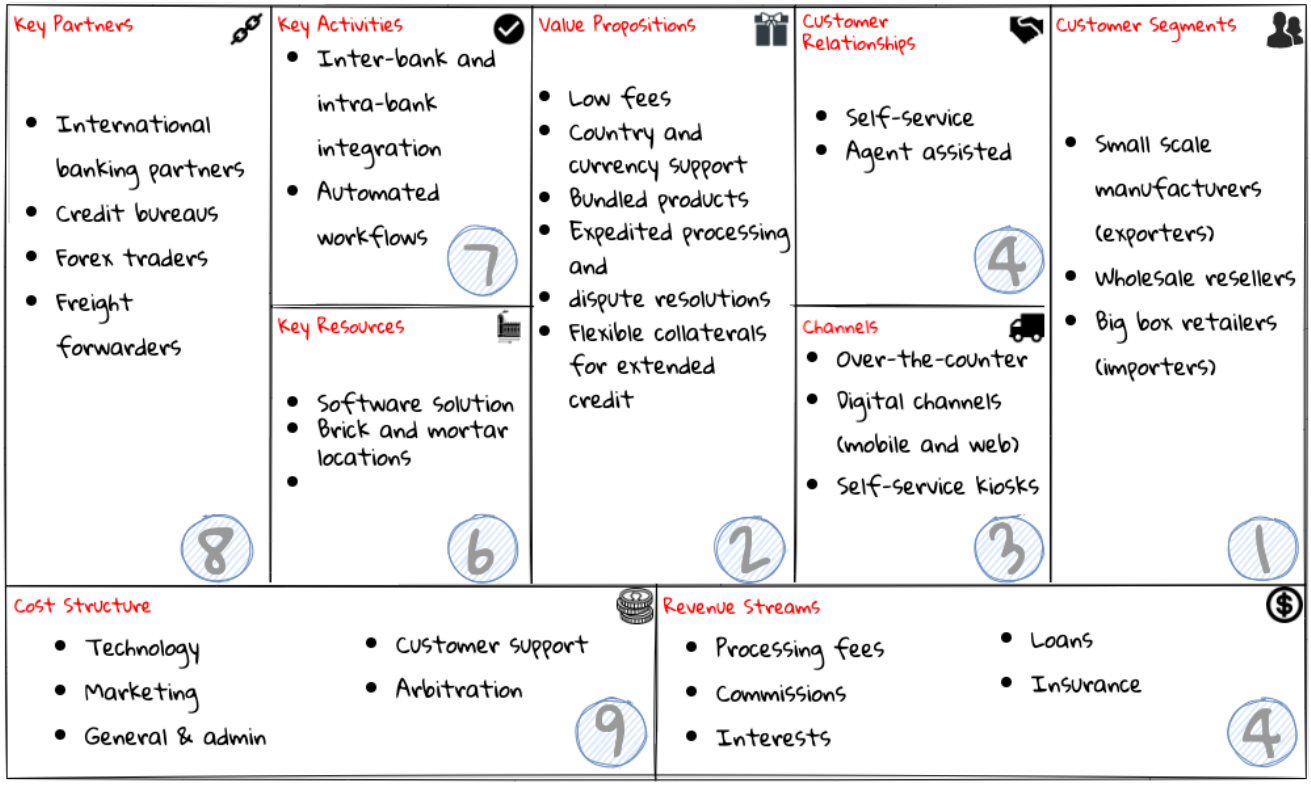


Figure 1- 38. Business model canvas for the international trade business at KP Bank

Using this canvas leads to insights about the customers we intend to serve at the bank, what value propositions are offered through what channels, and how we make money. When developing a business model canvas, it is recommended that we follow the numbered sequence depicted above in order to gain a better understanding of the:

1. Desirability of the business (who our customers are what they want).
2. Feasibility of the business (how we can operationalize and deliver it).
3. Economic viability of the business (how we can identify costs and capture profits).

Creating the business model canvas can prove challenging if you do not have an existing product already — which is usually true in case of startups or existing enterprises expanding into new business areas. In such cases, a variation in the form of the lean canvas is worth exploring.

3.4.2. Lean canvas

A variation of the business model canvas, called the *lean canvas* was conceived by Ash Maurya for lean startups. In contrast to the business model canvas, the main emphasis here is to first and foremost elaborate on the problem that needs to be solved and explore potential solutions. In his own words: “My approach to making the canvas actionable was capturing that which was most uncertain, or more accurately, that which was most risky.” This is pertinent for businesses operating under high uncertainty (which is usually true for startups). Similar to DDD, it encourages focusing on the problem as the starting point for building a business.

Structurally, it is similar to the business model canvas, with the following differences:

1. ***Problem*** instead of *Key Partners*: because it is common for businesses to fail due to misunderstanding the problem they are solving. The rationale for replacing the *Key Partners*

block is that when you are an unknown entity looking to establish an unproven product, pursuing key partnerships may be too premature.

2. ***Solutions*** instead of *Key Activities*: because it is important to try more than one solution iteratively and respond to feedback. *Key Activities* is removed because they are usually a by-product of the solution.
3. ***Key Metrics*** instead of *Key Resources*: because it is very important to know that we are progressing in the right direction. It is advisable to focus on a small number of **key metrics** to enable pivoting quickly if needed. *Key Resources* is removed because developing new products may not be as resource-intensive in the current climate. Furthermore, they may appear in the *Unfair Advantage* box, which we discuss next.
4. ***Unfair Advantage*** instead of *Customer Relationships*: because it clearly establishes our differentiators that are hard to replicate. This is closely aligned to the idea of the core subdomain we discussed in chapter 1 and gives us a clear picture of what we need to focus our energies on at the outset.

We have shown the result of a lean canvas workshop we ran for KP Bank here:

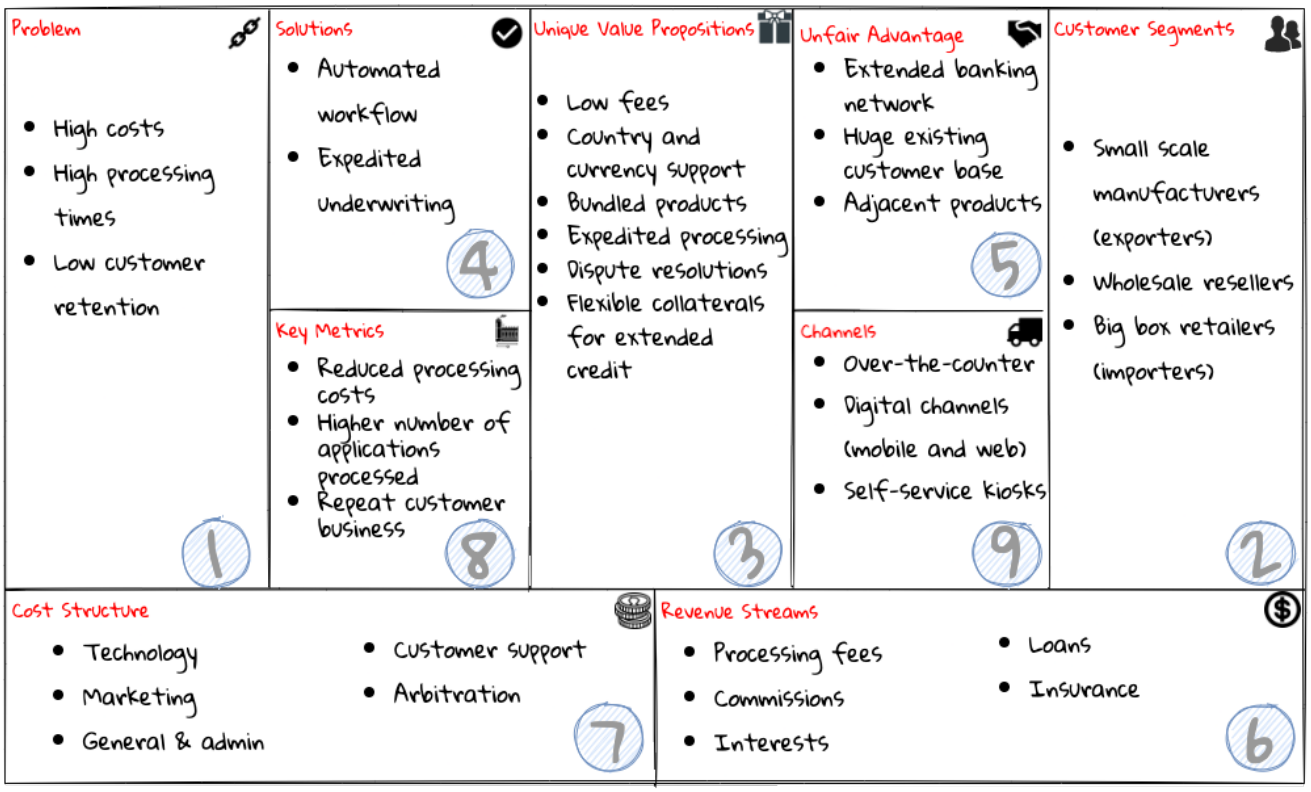


Figure 1- 39. Lean canvas for the international trade business at KP Bank

The exact sequence in which to fill out the lean canvas may vary. Ash Maurya himself suggests that there may not be a prescriptive run order to do this exercise on his blog[15]. Personally, we like starting from elaborating on the problem before moving on to other aspects of the canvas. Both the business model canvas and the lean canvas provide a high level view into the business model and the high priority problems and the potential solutions. Next, let's look at impact mapping, which is another lightweight planning technique to arrive at an outcome driven plan based on mind maps.

3.4.3. 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.

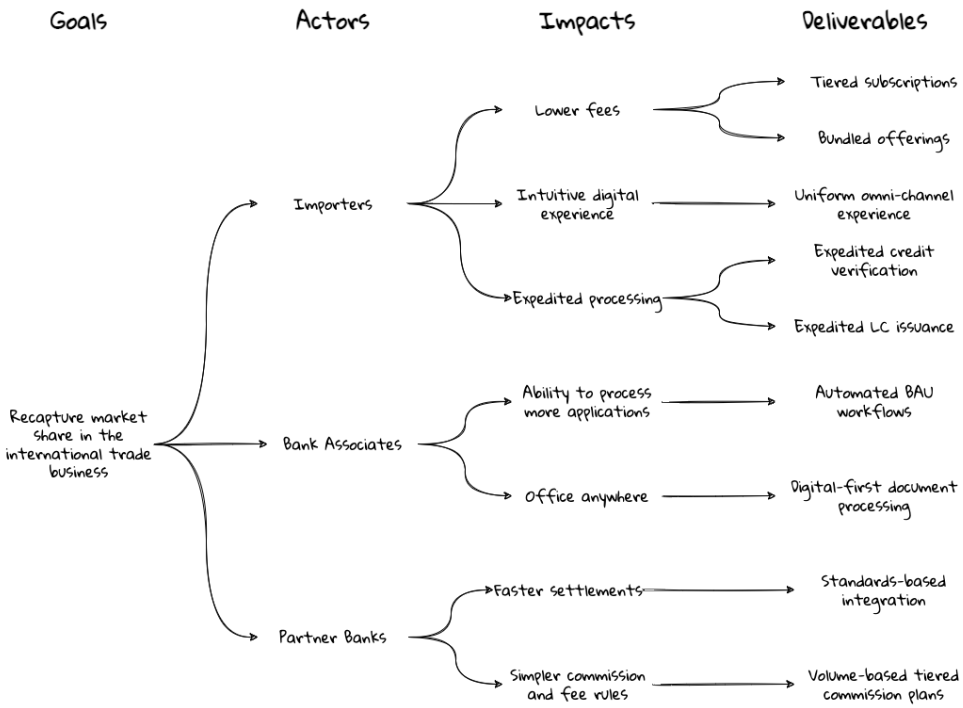


Figure 1- 40. A simple impact map for KP Bank's international trade business

Impact mapping provides an easy to understand visual representation of the relationship between the goals, the users and the impacts to the deliverables. Next, let us examine Wardley maps which enables us to dive deeper, understand our purpose and determine which portions of the business provide the most value.

3.4.4. Wardley maps

The business model canvas and lean canvas can help establish clarity of purpose at a high level. The Wardley map is another tool to help build a business strategy and establish purpose. 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. When building a Wardley map, it can be done

in six steps:

1. **Purpose**: What is your purpose? Why does the organization or project exist?
2. **Scope**: What is (and not) included within the scope of the map.
3. **Users**: Who uses or interacts with the thing you are mapping?
4. **User needs**: What do your users need from the thing you are mapping?
5. **Value chain**: What do we need to be doing to fulfill those needs captured above? These needs are arranged according to their dependencies — resulting in the creation of a value chain that maps user needs to a series of activities in the order of their visibility to the user (going from most visible to the least).
6. **Map**: Finally, plot the map using the evolutionary characteristics to decide where to place each component along the horizontal axis.

We conducted a Wardley mapping exercise at KP Bank for their international trade business as shown here:

On this canvas, we have chosen to elaborate the needs of only one class of users (Importers and Exporters) for brevity. In a real-world scenario, we would have to repeat steps 4, 5 and 6 for all types of users.

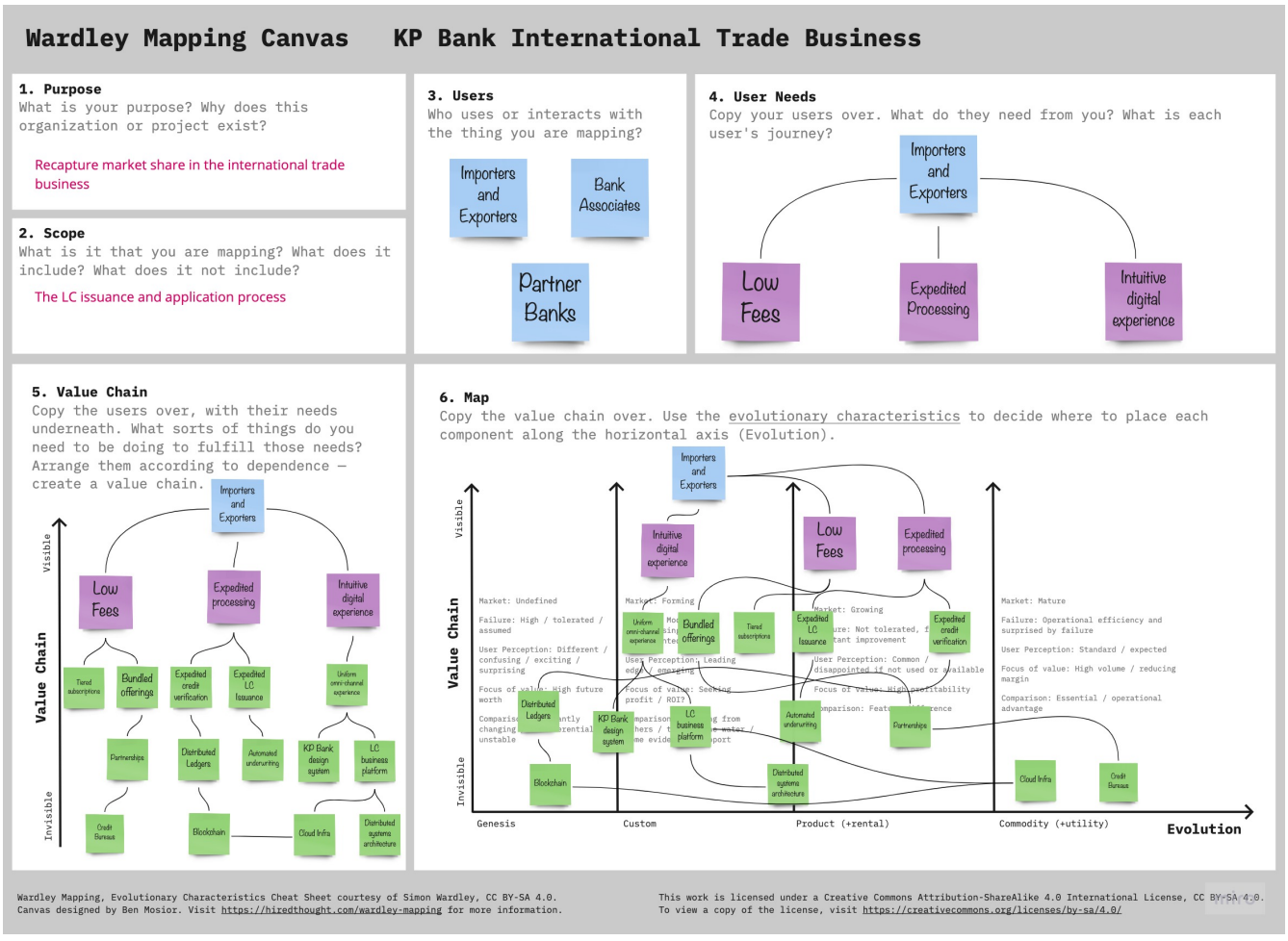


Figure 1- 41. Wardley map for the international trade business at KP Bank

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.

We have examined a number of lightweight and collaborative techniques to quickly gain an understanding of the problem space and the impacts we can have on our users and to our business. Each of these techniques is fairly lightweight and can be completed within a matter of a few hours. Each enables us to zoom in on the most impactful areas of the business and maximize ROI. In our experience, it is worth experimenting with more than one of these exercises (potentially all of them) as each can highlight a different facet of the business/user needs.

3.5. International trade products and services

International trade is fraught with risk, which then presents a degree of uncertainty over the timing of payments between the seller (exporter) and the buyer (importer), especially due to the lack of trust between the parties involved. For exporters, until payment is received, all sales are gifts. Consequently, exporters prefer receiving payment as soon as the order is placed or at least before the goods are shipped. For importers, until the goods are received, all payments made towards a purchase are donations. Consequently, importers prefer receiving goods as soon as possible and delaying payment until the goods are resold to generate enough money to pay the seller.

This situation presents an opportunity for trusted intermediaries (such as KP Bank) to play a significant role in brokering international trade transactions in a secure manner. When it comes to KP bank, it offers a number of products to facilitate international trade payments as listed here:

1. Letter of Credit (LC)
2. Documentary Collections (DC)
3. Open Account
4. Cash-in-advance
5. Consignments

The diagram below shows the risk profile of each of these payment methods from both an exporter’s and importer’s perspective:

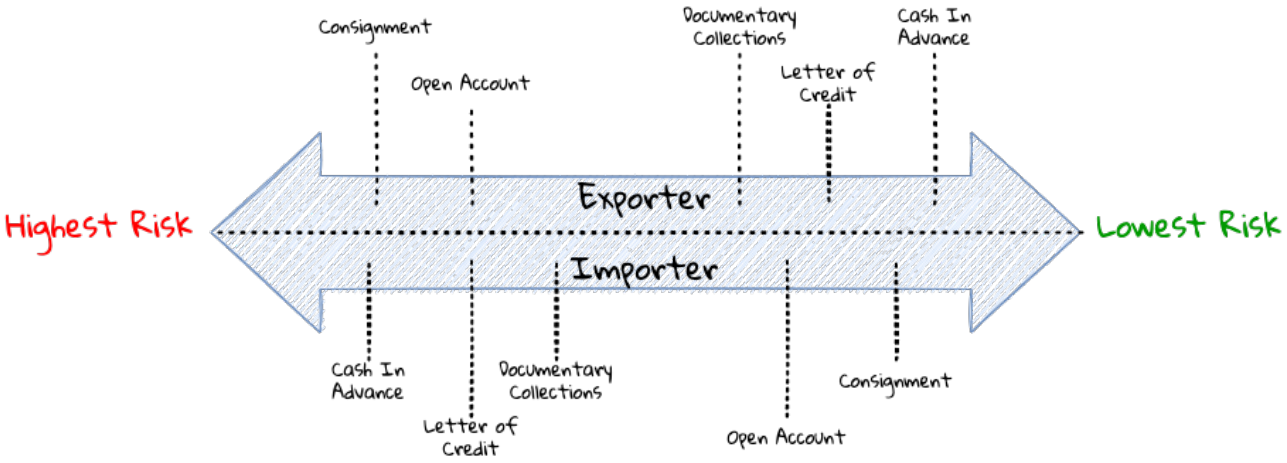


Figure 1- 42. Risk profile of international trade payment methods

As is evident here, the Documentary Collections and Letter of Credit products offer a good balance in providing solutions that are relatively secure from the perspective of both parties. The involvement of trusted intermediaries like KP Bank are required to play in the fulfillment process makes these payment methods less risky for both parties. From the bank's perspective, streamlining the process for these products on a priority also provides a greater business opportunity compared to the other products. Between the two, the Letter of Credit product satisfies most criteria outlined against user needs in the recently concluded business strategy sessions we elaborated above. Hence, the stakeholders at KP Bank have decided to move forward with investing heavily against the Letter of Credit product at the outset.

In the next chapter, and indeed the rest of the book, we will elaborate on how we can improve the Letter of Credit application, issuance and related processes making use of tenets that align closely with the tenets of DDD.

3.6. Summary

In this chapter, we have explored a variety of techniques that help establish whether a problem is the right one to be solving. Specifically, we looked at business value canvas and the lean canvas to clarify the business strategy for both startups and established enterprises alike. We then looked at impact maps which enable you to unambiguously correlate business goals to user impacts and the deliverables needed to create that impact. Finally, we looked at Wardley maps to further drill down areas that are important to focus energies on through establishing build versus buy decisions and the importance of business strategy in relation to the competitors and the relative risk involved when treading on uncharted waters.

In the next chapter, we will look at techniques and practices to drill down further and gain an understanding of the Letter of Credit business so that we can start crafting domain model(s) to enable us to arrive at an appropriate solution.

[15] https://blog.leanstack.com/what-is-the-right-fill-order-for-a-lean-canvas/

Chapter 4. Domain analysis and modeling

> He who asks a question remains a fool for five minutes. He who does not ask remains a fool forever.
>
> — Chinese Proverb

4.1. Introduction

As we saw in the previous chapter, misinterpreted requirements cause a significant portion of software projects to fail. Arriving at a shared understanding and creating a useful domain model necessitates high degrees of collaboration with domain experts. In this chapter, we will introduce the sample application we will use throughout the book and explore modeling techniques such as domain storytelling and eventstorming to enhance our collective understanding of the problem in a reliable and structured manner.

The following topics will be covered in this chapter:

- Introducing the example application (Letter of Credit)
- Enhancing shared understanding
- Domain storytelling
- EventStorming

This chapter will help developers and architects learn how to apply these techniques in real-life situations to produce elegant software solutions that mirror the problem domain that needs to be solved. Similarly, non-technical domain experts will understand how to communicate their ideas and collaborate effectively with technical team members to accelerate the process of arriving at a shared understanding.

4.2. Technical requirements

There are no specific technical requirements for this chapter. However, given that it may become necessary to collaborate remotely as opposed to being in the same room with access to a whiteboard, it will be useful to have access to the following:

1. Digital whiteboard (like https://www.mural.co/ or http://miro.com/)
2. Online domain storytelling modeler (like https://www.wps.de/modeler/)

4.2.1. Understanding Letter of Credit (LC)

Documentary Letter of Credit (LC) is a financial instrument issued by the banks as a contract between the importer (or buyer) and the exporter (or seller). This contract specifies terms and conditions of the transaction under which importer promises to pay the exporter in exchange for the goods or services provided by the exporter. Letter of Credit transaction typically involves multiple parties. A simplified summary of the parties involved is described below:

1. **Importer**: The buyer of the goods or services.
2. **Exporter**: The seller of the goods or services.
3. **Freight Forwarder**: The agency that handles shipment of goods on behalf of the exporter. This is only applicable in cases there is an exchange of physical goods.
4. **Issuing Bank**: The bank that the importer requests to issue the LC application. Usually the importer has a pre-existing relationship with this bank.
5. **Advising Bank**: The bank that informs the exporter about the issuance of the LC. This is usually a bank that is native to the exporter's country.
6. **Negotiating Bank**: The bank that the exporter submits documents for the shipment of goods, or the services provided. Usually the exporter has a pre-existing relationship with this bank.
7. **Reimbursement Bank**: The bank that reimburses the funds to the negotiating bank, at the request of the issuing bank.

It is important to note that the same bank can play more than one role for a given transaction. In the most complex cases, there can be four distinct banks involved for a transaction (sometimes even more, but we will skip those cases for brevity).

4.3. The LC issuance application

As discovered in the previous chapter, Kosmo Primo Bank needs us to focus on streamlining the process used for LC application and issuance functions. In this chapter, and indeed the rest of this book, we will strive to understand, evolve, design and build a software solution to make the process more efficient by replacing the largely manual and error-prone workflows with more simplified processes based on larger amounts of automation.

We understand that unless one is an expert dealing with international trade, it is unlikely that one would have an intimate understanding of concepts like Letters of Credit (LCs). In the upcoming section, we will look at demystifying LCs and how to work with them.

4.4. Enhancing shared understanding

When working with a problem where domain concepts are unclear, there is a need to arrive at a common understanding among key team members (both those that have bright ideas — the business/product people, and those that translate those ideas into working software — the software developers). For this process to be effective, we tend to look for approaches that are:

- Quick, informal and effective
- Collaborative - Easy to learn and adopt for both non-technical and technical team members
- Pictorial - because a picture can be worth a thousand words
- Usable for both coarse grained and fine-grained scenarios

There are several means to arrive at this shared understanding. Some commonly used approaches are listed below:

- UML
- BPMN
- Use Cases
- User Story Mapping
- CRC Models
- Data Flow Diagrams

The above-mentioned modeling techniques have tried to formalize knowledge and express them in form of a structure diagram or text to help in delivering the business requirements as a software product. However, this attempt has not narrowed but has widened the gap between the business and the software systems. While these methods tend to work well for technical audiences, they are usually not as appealing to non-technical users.

In order to restore the balance and promote the use of techniques that can work for both parties, we will use **domain storytelling** and **eventstorming** as our means to capture business knowledge from domain experts for consumption of developers, business analysts etc.

4.5. Domain storytelling

> You're never going to kill storytelling because it's built into the human plan. We come with it.
>
> — Margaret Atwood

4.5.1. Introducing Domain Storytelling

Scientific research has now proven that learning methods that employ audiovisual aids assist both the teacher and the learners in retaining and internalizing concepts very effectively. In addition, teaching what one has learnt to someone else helps reinforce ideas and also stimulates the formation of new ones. Domain storytelling is a collaborative modeling technique that combines a pictorial language, real-world examples, and a workshop format to serve as a very simple, quick and effective technique for sharing knowledge among team members. Domain Storytelling is a technique invented and popularized by Stefan Hofer and Henning Schwentner based on some related work done at the University of Hamburg called *cooperation pictures*.

A pictorial notation of the technique is illustrated in the diagram below:

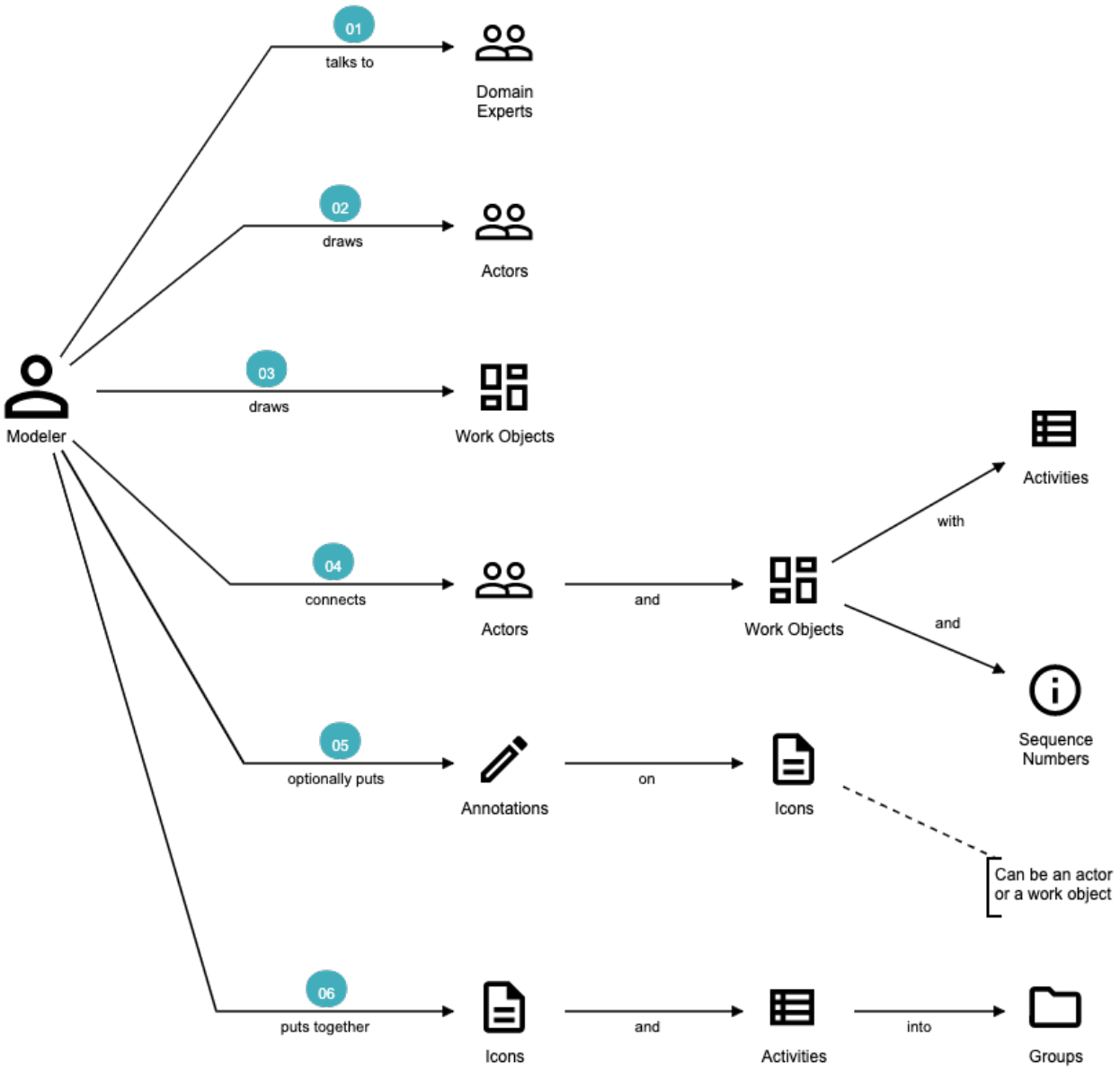


Figure 1- 43. Domain storytelling summarized

A domain story is conveyed using the following attributes:

Actors - Stories are communicated from the perspective of an actor (noun), for example, the issuing bank, who plays an active role in the context of that particular story. It is a good practice to use the ubiquitous language for the particular domain.

Work Objects - Actors act on some object, for example, applying for an LC. Again, this would be a term (noun) commonly used in the domain.

Activities - Actions (verb) performed by the actor on a work object. Represented by a labelled arrow connecting the actor and the work object.

Annotations - Used to capture additional information as part of the story, usually represented in few sentences.

Sequence Numbers - Usually, stories are told one sentence after the other. Sequence numbers helps capture the sequence of the activities in a story.

Groups - An outline to represent a collection of related concepts ranging from repeated/optional activities to subdomains/organizational boundaries.

4.5.2. Using DST for the LC application

XYZ Bank has a process that allows processing of LCs. However, this process is very archaic, paper-based and manually intensive. Very few at the bank fully understand the process end-to-end and natural attrition has meant that the process is overly complex without good reason. So they are looking to digitize and simplify this process. DST itself is just a graphical notation which can be done in isolation. However, it is typical to not do this on your own and employ a workshop style with domain experts and software experts working collaboratively.

In this section, we will employ a DST workshop to capture the current business flow. The following is an excerpt of such a conversation between **Katie**, *the domain expert* and **Patrick**, *the software developer*.

Patrick : *"Can you give me a high level overview of a typical LC Flow?"*
Katie : *"Sure, it all begins with the importer and the exporter entering into a contract for purchase of goods or services."*
Patrick : *"What form does this contract take? Is it a formal documentClause? Or is this just a conversation?"*
Katie : *"This is just a conversation."*
Patrick : *"Oh okay. What does the conversation cover?"*
Katie : *Several things — nature and quantity of goods, pricing details, payment terms, shipment costs and timelines, insurance, warranty, etc. These details may be captured in a purchase order — which is a simple documentClause elaborating the above.*

At this time, Patrick draws this part of the interaction between the importer and the exporter. This graphic is depicted in the following diagram:

Figure 1- 44. Interaction between importer and exporter

Patrick : *"Seems straight forward, so where does the bank come into the picture?"*
Katie : *"This is international trade and both the importer and the exporter need to mitigate the financial risk involved in such business transactions. So they involve a bank as a trusted mediator."*
Patrick : *"What kind of bank is this?"*
Katie : "*Usually, there are multiple banks involved. But it all starts with an **issuing bank**.*"
Patrick : *"What is an issuing bank?"*

Katie : *"Any bank that is authorized to mediate international trade deals. This has to be a bank in the importer's country."*
Patrick : *"Does the importer need to have an existing relationship with this bank?"*
Katie : *"Not necessarily. There may be other banks with whom the importer may have a relationship with — which in turn liaises with the issuing bank on the importer's behalf. But to keep it simple, let's assume that the importer has an existing relationship with the issuing bank — which is our bank in this case."*
Patrick : *"Does the importer provide details of the purchase order to the issuing bank to get started?"*
Katie : *"Yes. The importer provides the details of the transaction by making an **LC application**."*

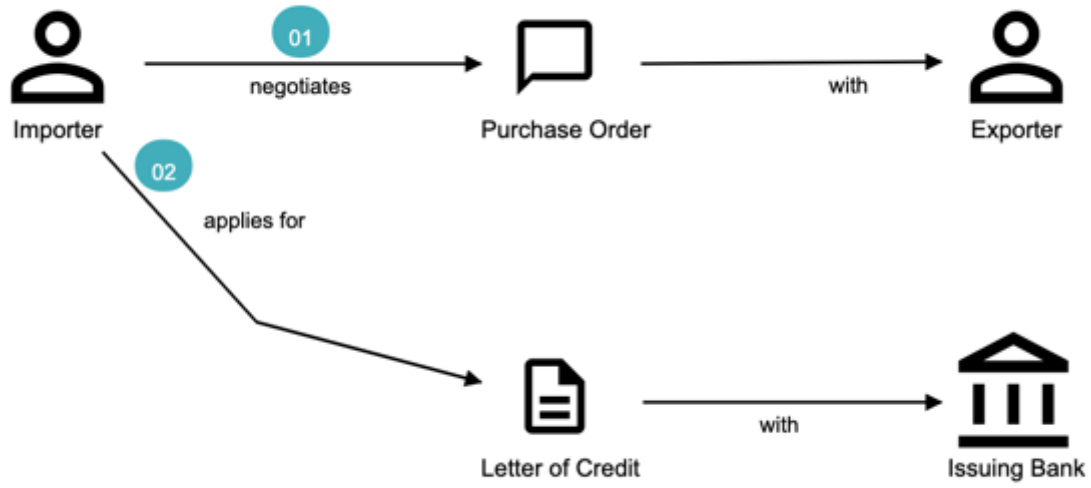


Figure 1- 45. Introducing the LC and the issuing bank

Patrick : *"What does the issuing bank do when they receive this LC application?"*
Katie : *"Mainly two things — whet the financial standing of the importer and the legality of the goods being imported."*
Patrick : "Okay. What happens if everything checks out?"
Katie : *"The issuing bank approves the LC and notifies the importer."*

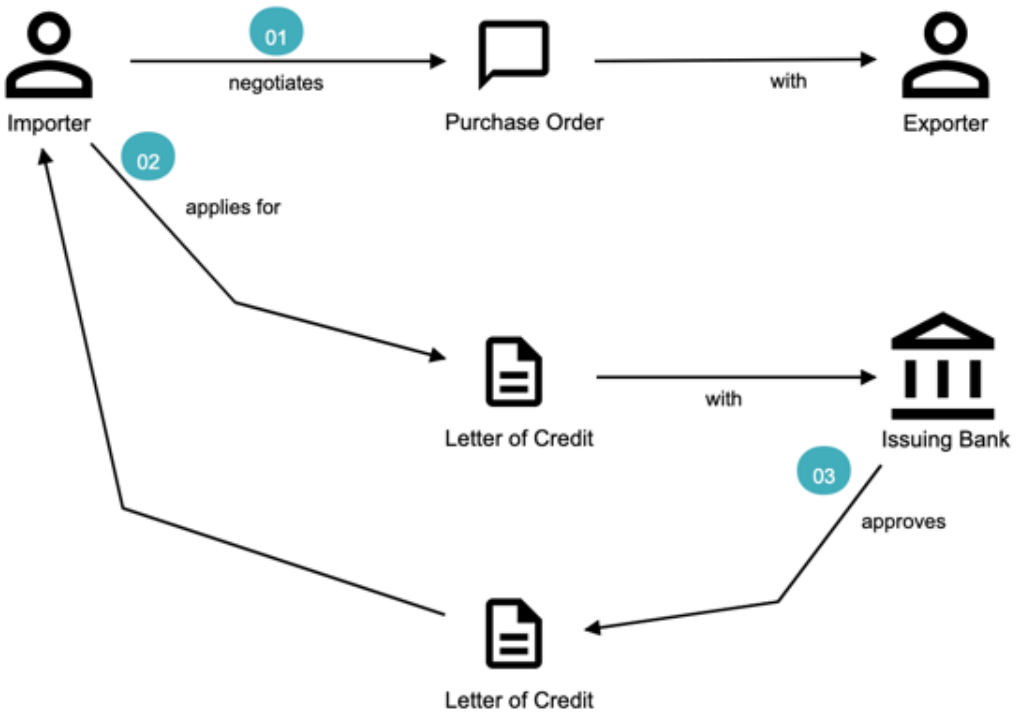


Figure 1- 46. Notifying LC approval to the importer

Patrick : *"What happens next? Does the issuing bank contact the exporter now?"*
Katie : *"Not yet. It is not that simple. The issuing bank can only deal with a counterpart bank in the exporter's country. This bank is called the* ***advising bank****."*

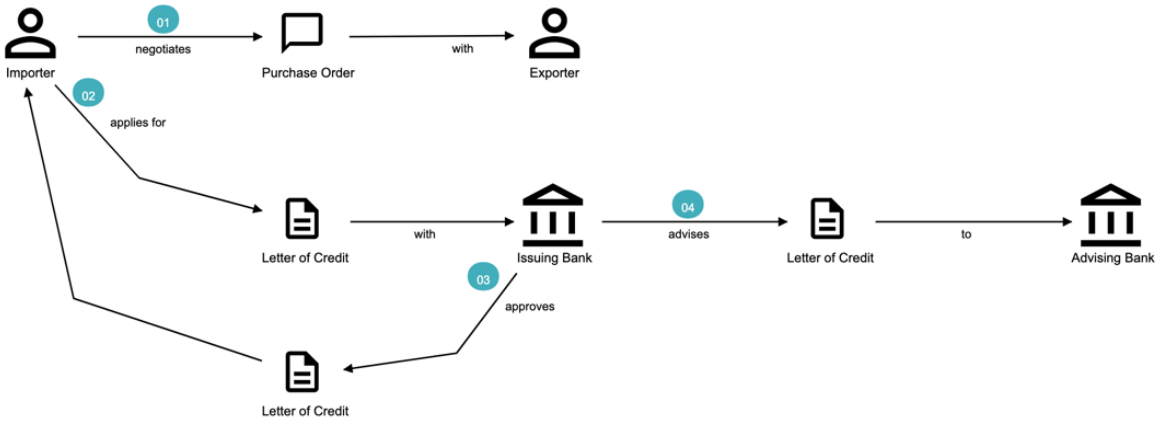


Figure 1- 47. Introducing the advising bank

Patrick : *"What does the advising bank do?"*
Katie : *"The advising bank notifies the exporter about the LC."*
Patrick : *"Doesn't the importer need to know that the LC has been advised?"*
Katie : *"Yes. The issuing bank notifies the importer that the LC has been advised to the exporter."*

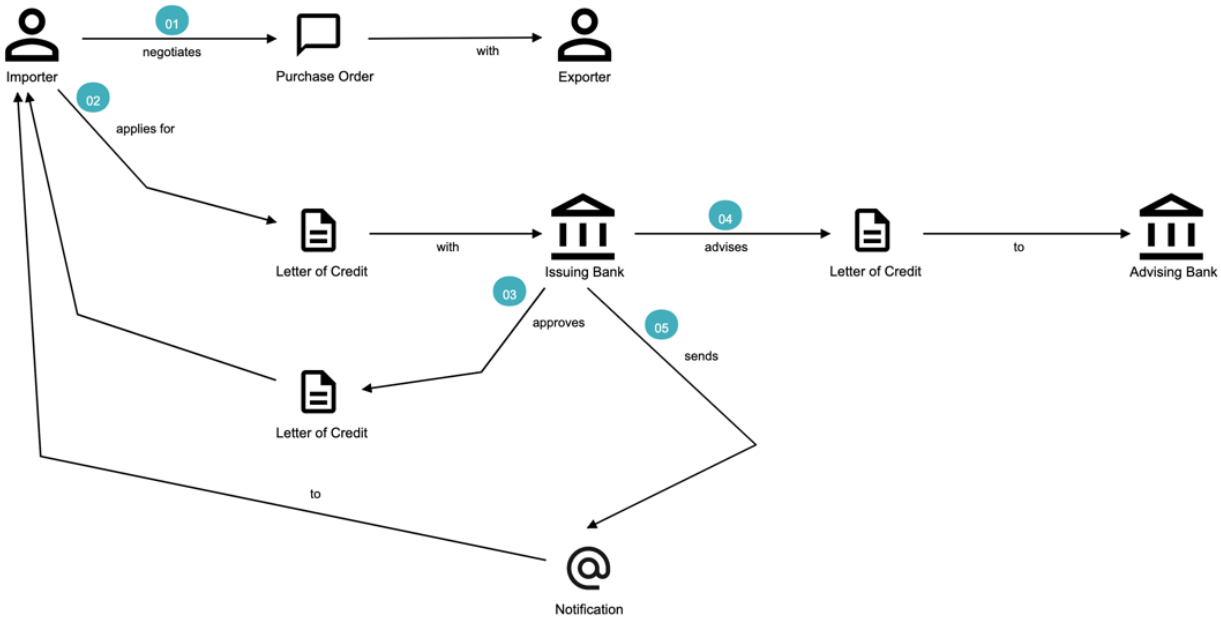


Figure 1- 48. Advice notification to the importer

Patrick : *"How does the exporter know how to proceed?"*
Katie : *"Through the advising bank — they notify the exporter that the LC was issued."*

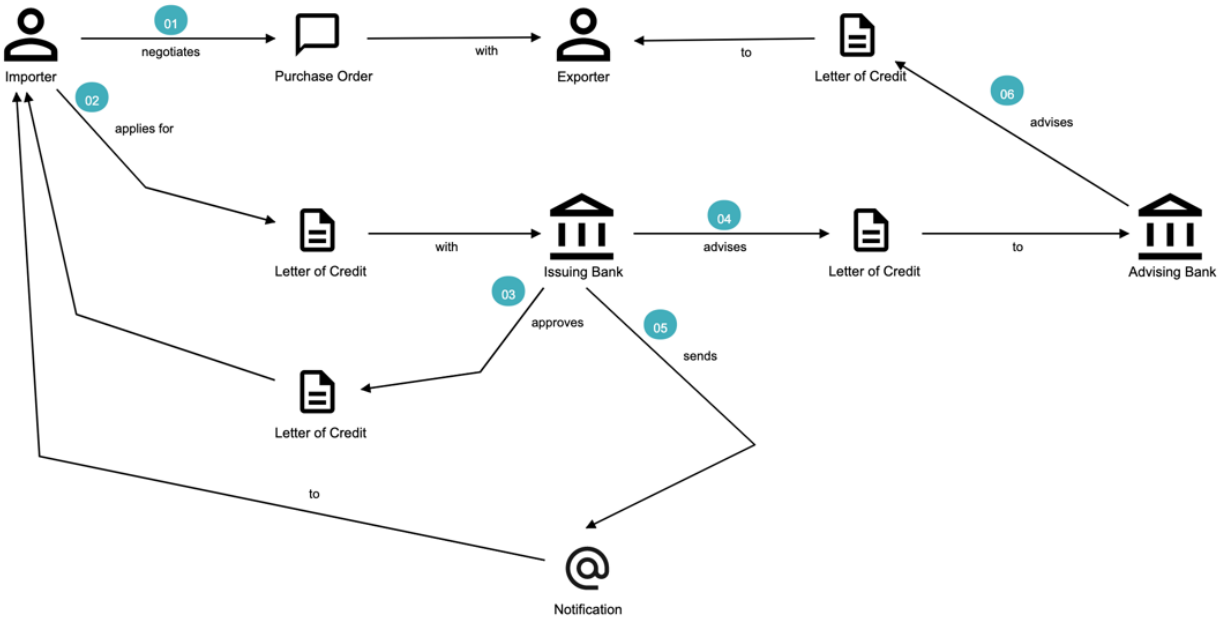


Figure 1- 49. Dispatching the advice to the exporter

Patrick : *"Does the exporter initiate shipping at this time and how do they get paid?"*
Katie : *"Through the advising bank — they notify the exporter that the LC was issued and this triggers the next steps in the process — this process of settling the payment is called* ***settlement****. But let's focus on issuance right now. We will discuss settlement at a later time."*

We have now looked at an excerpt of a typical DST workshop. The DST workshop has served to provide a reasonably good understanding of the high level business flow. Note that we have not referenced any technical artifacts during the process.

To be able to refine this flow and convert it into a form that can be used to design the software solution, we will need to further enhance this view. In the upcoming section, we will use EventStorming as a structured approach to achieve that.

4.6. EventStorming

> The amount of energy necessary to refute bullshit is an order of magnitude bigger than to produce it.
>
> — Alberto Brandolini

4.6.1. Introducing EventStorming

In the previous section, we gained a high level understanding of the LC Issuance process. To be able to build a real-world application, it will help to use a method that delves into the next level of detail. EventStorming, originally conceived by Alberto Brandolini, is one such method for the collaborative exploration of complex domains.

In this method, one simply starts by listing out all the events that are significant to the business domain in roughly chronological order on a wall or whiteboard using a bunch of colored sticky notes. Each of the note types (denoted by a color) serve a specific purpose as outlined below:

- **Domain Event**: An event that is significant to the business process — expressed in past tense.
- **Command**: An action or an activity that may result in one or more domain events occurring. This is either user initiated or system initiated, in response to a domain event.
- **User**: A person who performs a business action/activity.
- **Policy**: A set of business invariants (rules) that need to be adhered to, for an action/activity to be successfully performed.
- **Query/Read Model**: A piece of information required to perform an action/activity.
- **External System**: A system significant to the business process, but out of scope in the current context.
- **Hotspot**: Point of contention within the system that is likely confusing and/or puzzling beyond a small subsection of the team.
- **Aggregate**: An object graph whose state changes consistently and atomically. This is consistent with the definition of aggregates we saw in Chapter 2.

The depiction of the stickies for our EventStorming workshop is shown here:

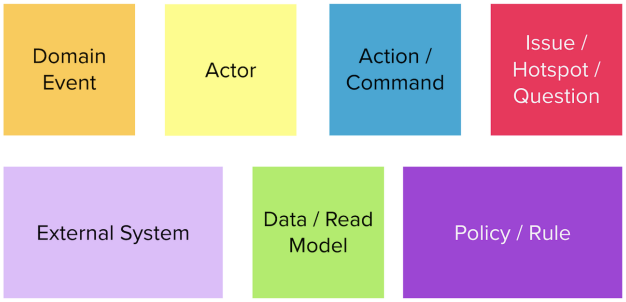


Figure 1- 50. EventStorming legend

Why domain events? When trying to understand a business process, it is convenient to express significant facts or things that happen in that context. It can also be informal and easy for audiences that are uninitiated with this practice. This provides an easy to digest visual representation of the domain complexity.

4.6.2. Using eventStorming for the LC issuance application

Now that we have a high level understanding of the current business process, thanks to the domain storytelling workshop, let's look at how we can delve deeper using eventstorming. The following is an excerpt of the stages from an eventstorming workshop for the same application.

1. Outline the event chronology

During this exercise, we recall significant **domain events** (using orange stickies) that happen in the system and paste them on the whiteboard, as depicted below. We ensure that the event stickies are pasted roughly in the chronological order of occurrence. As the timeline is enforced, the business flow will begin to emerge.

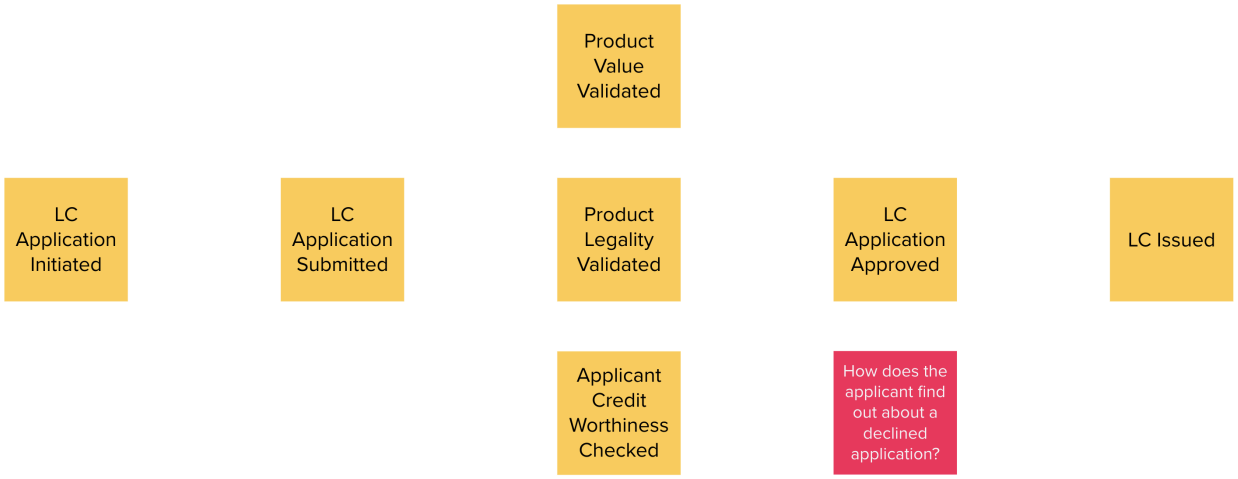


Figure 1- 51. Event chronology

This acts as an aid to understand the big picture. This also enables people in the room to identify hotspots in the existing business process. In the above illustration, we realized that, the process to handle "declined LC applications" is sub-optimal, i.e. applicants do not receive any information when their application is declined.

To address this, we added a new domain event which explicitly indicates that an application is

declined, as depicted below:

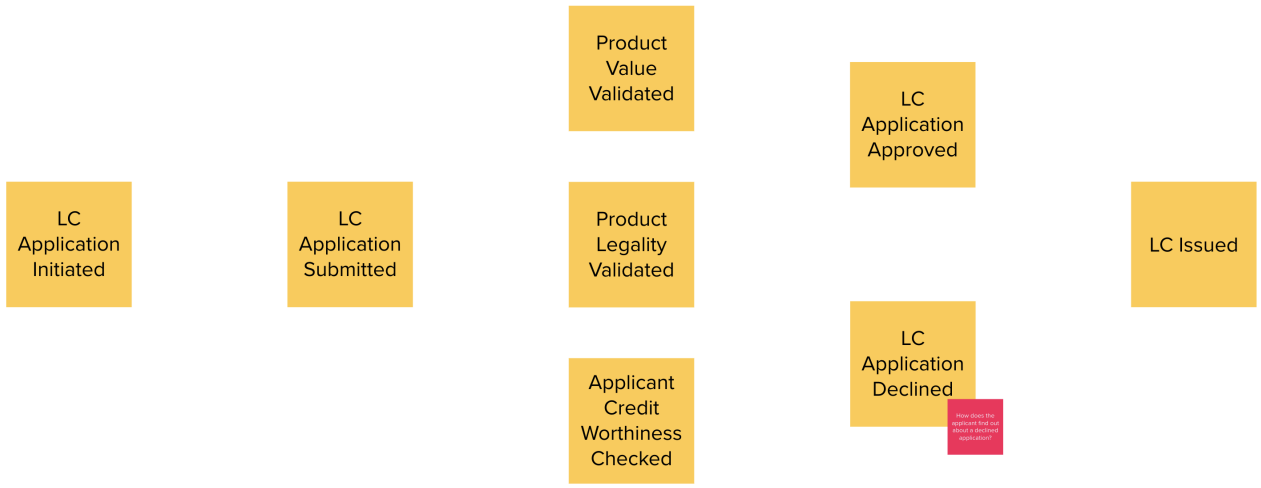


Figure 1- 52. New event to handle declined applications

2. Identify triggering activities and external systems

Having arrived at a high level understanding of event chronology, the next step is to embellish the visual with **activities/actions** that cause these events to occur (using blue stickies) and interactions with **external systems** (using pink stickies).

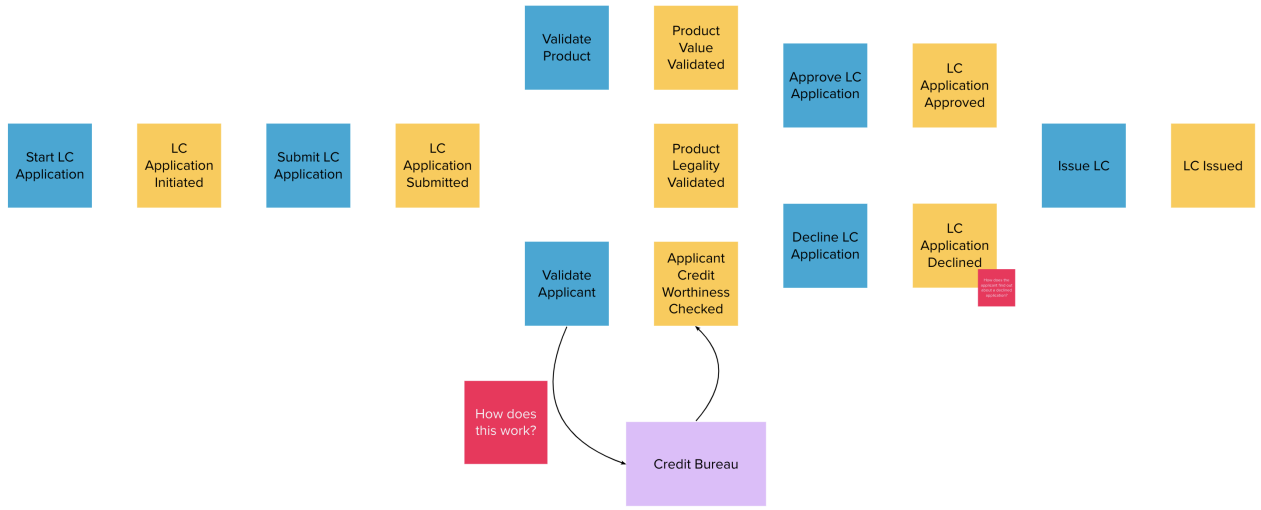


Figure 1- 53. Activities and external systems

3. Capture users, context and policies

The next step is to capture **users** who perform these activities along with their functional **context** (using yellow stickies) and policies (using purple stickies).

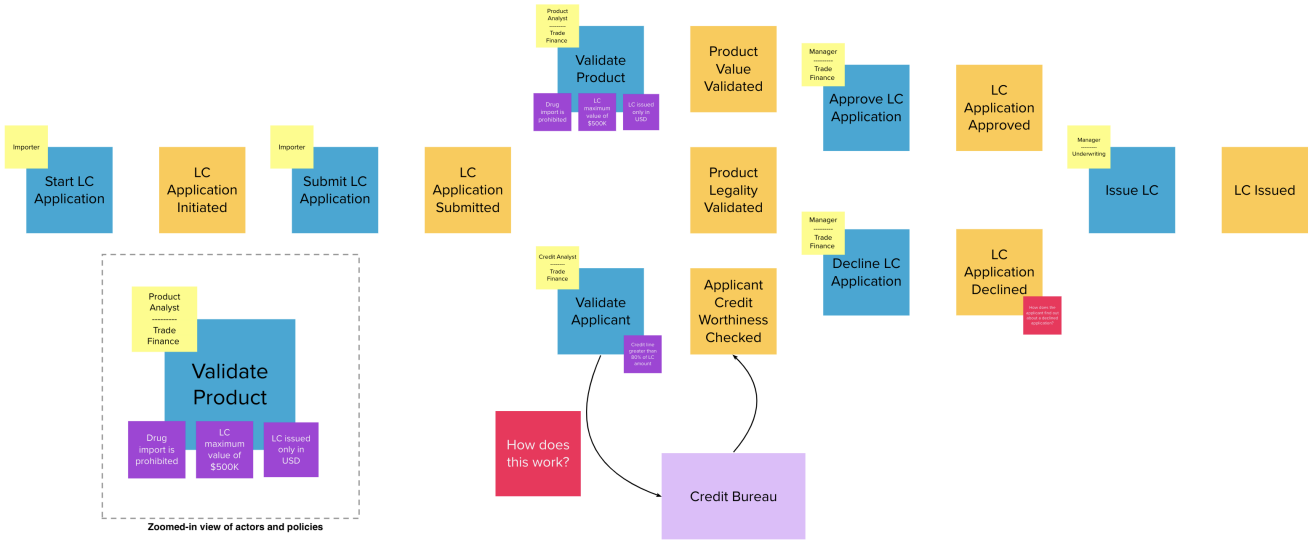


Figure 1- 54. Users and policies

4. Outline query models

Every activity requires a certain set of data to be able to be performed. Users will need to view out-of-band data that they need to act upon and also see the result of their actions. These sets of data are represented as **query models** (using green stickies).

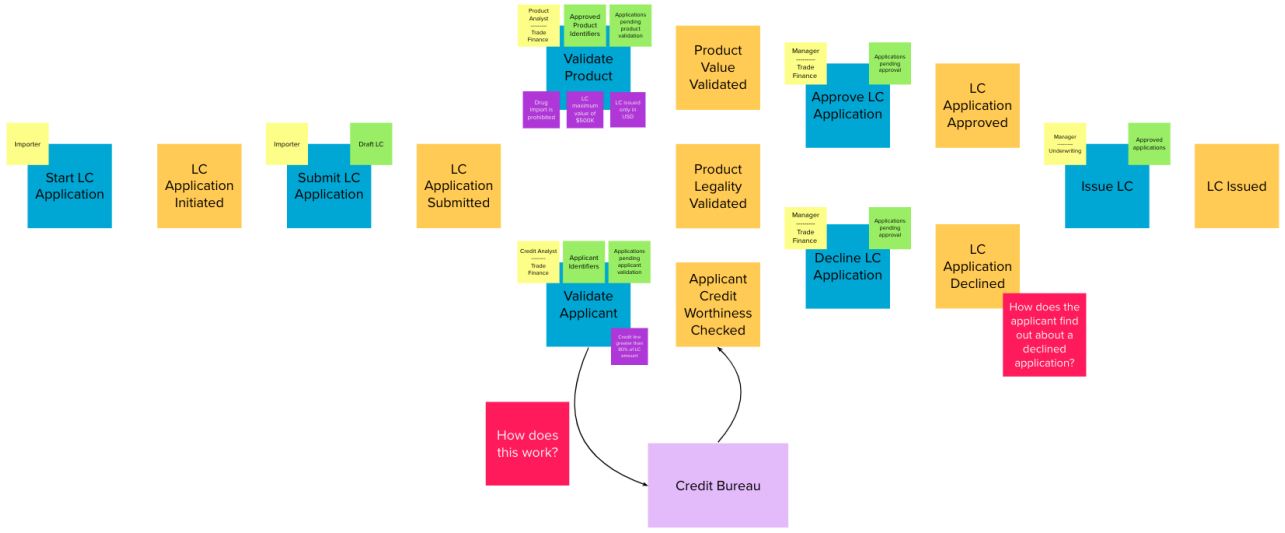


Figure 1- 55. Big picture eventstorming workshop board

For both the domain storytelling and eventstorming workshops, it works best when we have approximately 6-8 people participating with a right mix of domain and technology experts.

This concludes the eventstorming workshop to gain a reasonably detailed understanding of the LC application and issuance process. Does this mean that we have concluded the domain requirements gathering process? Not at all — while we have made significant strides in understanding the domain, there is still a long way to go. The process of elaborating domain requirements is perpetual. Where are we in this continuum? The picture below is an attempt to clarify:

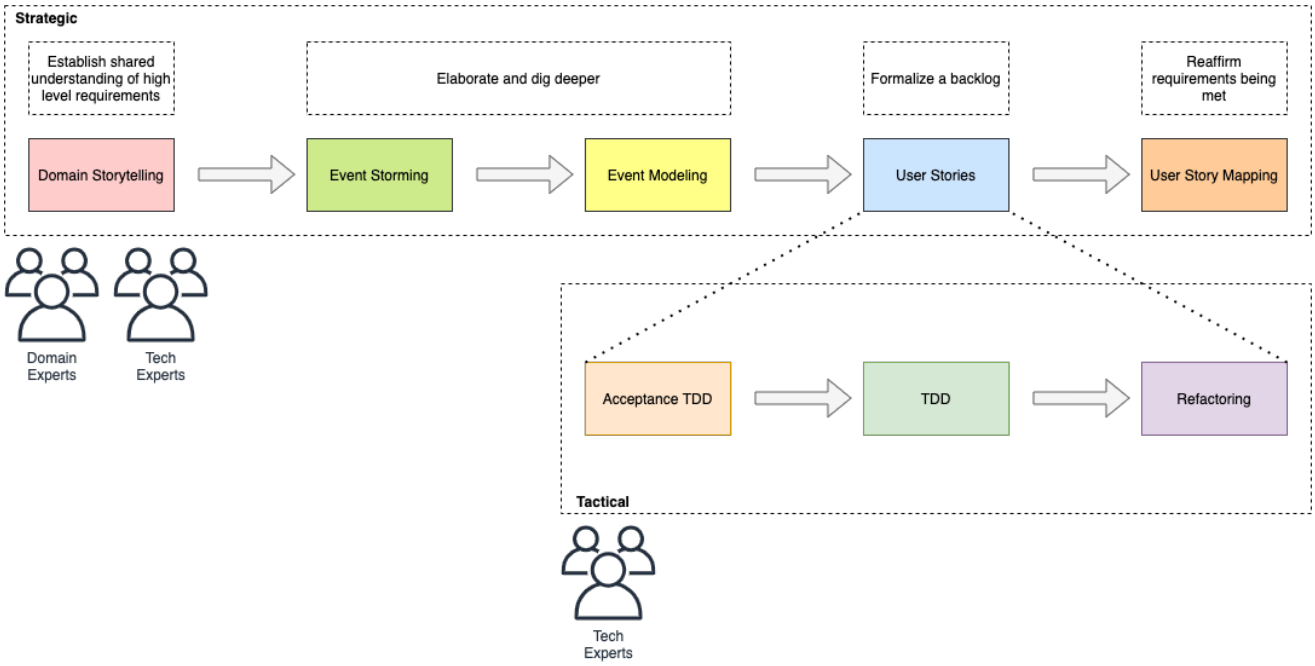


Figure 1- 56. Domain requirements elaboration continuum

In subsequent chapters we will examine the other techniques in more detail.

4.7. Summary

In this chapter we examined two ways to enhance our collective understanding of the problem domain using two lightweight modeling techniques — domain storytelling and eventstorming.

Domain storytelling uses a simple pictorial notation to share business knowledge among domain experts and technical team members. Eventstorming, on the other hand, uses a chronological ordering of domain events that occur as part of the business process to gain that same shared understanding.

Domain storytelling can be used as an introductory technique to establish high level understanding of the problem space, while eventstorming can be used to inform detailed design decisions of the solution space.

With this knowledge, we should be able to dive deeper into the technical aspects of solution implementation. In the next chapter, we will start implementation of the business logic, model our aggregate along with commands and domain events.

4.8. Questions

1. When should you use domain storytelling?
2. Pick an application in your current context. Can you use domain storytelling to capture actors, work objects and activities for the scenario you picked?
3. When should you use eventstorming?
4. Pick an application in your current context. Can you use eventstorming to capture domain events, actors, actions, hotspots, query models, external systems, etc. for the scenario you

picked?

4.9. Further reading

Title	Author	Location
Domain Storytelling	Stefan Hofer and Henning Schwentner	https://leanpub.com/domainstorytelling
An Introduction to Domain Storytelling	Virtual Domain-Driven Design	https://www.youtube.com/watch?v=d9k9Szkdprk
Domain Storytelling Resources	Stefan Hofer	https://github.com/hofstef/awesome-domain-storytelling
Introducing EventStorming	Alberto Brandolini	https://leanpub.com/introducing_eventstorming
Introducing Event Storming	Alberto Brandolini	https://ziobrando.blogspot.com/2013/11/introducing-event-storming.html
Event storming for fun and profit	Dan Terhorst-North	https://speakerdeck.com/tastapod/event-storming-for-fun-and-profit
EventStorming	Allen Holub	https://holub.com/event-storming/

4.10. Answers

1. Refer to section Section 4.5.1
2. Share and validate the domain storytelling artifact you created with your teammates.
3. Refer to section Section 4.6.1
4. Share and validate the eventstorming artifact you created with your teammates.

Chapter 5. Implementing Domain Logic

> To communicate effectively, the code must be based on the same language used to write the requirements—the same language that the developers speak with each other and with domain experts.
>
> — Eric Evans

In the Command Query Responsibility Segregation (CQRS) section, we describe how DDD and CQRS complement each other and how the command side (write requests) is the home of business logic. In this chapter, we will implement the command side API for the LC application using Spring Boot and Axon Framework, JSR-303 Bean Validations and persistence options by contrasting between state-stored vs event-sourced aggregates. The list of topics to be covered is as follows:

- Identifying aggregates
- Handling commands and emitting events
- Test-driving the application
- Persisting aggregates
- Performing validations

By the end of this chapter, you would have learned how to implement the core of your system (the domain logic) in a robust, well encapsulated manner. You will also learn how to decouple your domain model from persistence concerns. Finally, you will be able to appreciate how to perform DDD's tactical design using services, repositories, aggregates, entities and value objects.

5.1. Technical requirements

To follow the examples in this chapter, you will need access to:

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- Maven 3.x
- Spring Boot 2.4.x
- JUnit 5.7.x (Included with spring boot)
- Axon Framework 4.4.7 (DDD and CQRS Framework)
- Project Lombok (To reduce verbosity)
- Moneta 1.4.x (Money and currency reference implementation - JSR 354)

5.2. Continuing our design journey

In the previous chapter, we discussed eventstorming as a lightweight method to clarify business flows. As a reminder, this is the output produced from our eventstorming session:

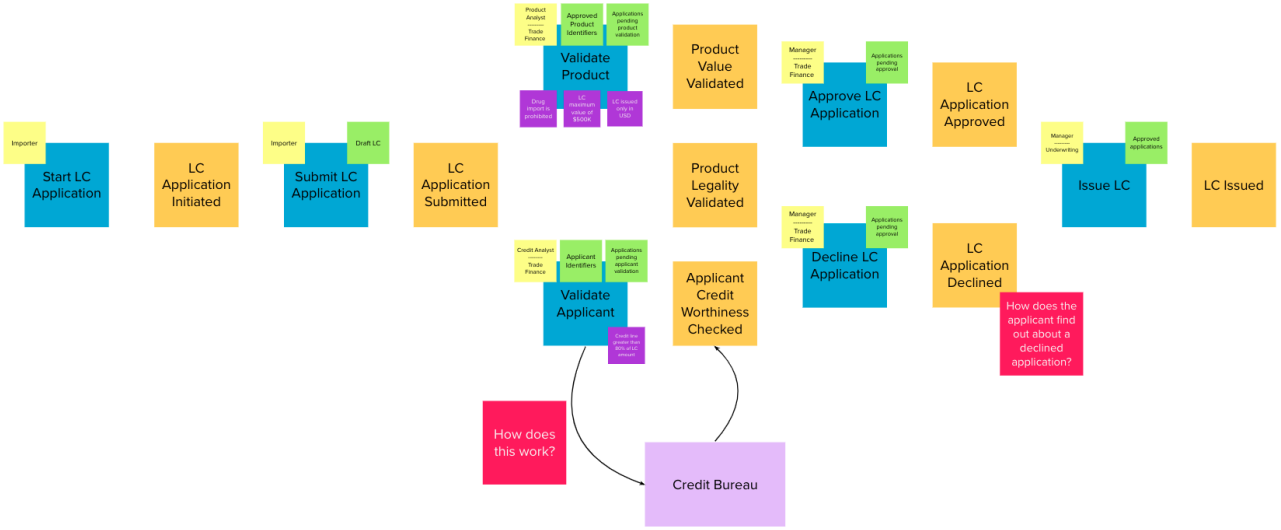


Figure 1- 57. Recap of eventstorming session

As mentioned previously, the **blue** stickies in this diagram represent **commands**. We will be using the Command Query Responsibility Segregation (CQRS) pattern as a high level architecture approach to implement the domain logic for our LC issuance application. Let's examine the mechanics of using CQRS and how it can result in an elegant solution. For a recap of what CQRS is and when it is appropriate to apply this pattern, please refer to the "When to use CQRS" section in Chapter 3.

CQRS is by no means a silver bullet. Although it is general-purpose enough to be used in a variety of scenarios, it is a paradigm shift as applied to mainstream software problems. Like any other architecture decision, you should apply due diligence when choosing to adopt CQRS to your situation.

Let's look at how this works in practice by implementing a representative sliver of the **command** side of the Letter of Credit application using the Spring and Axon frameworks.

5.3. Implementing the command side

In this section, we will focus on implementing the command side of the application. This is where we expect all the business logic of the application to be implemented. Logically, it looks as shown here:

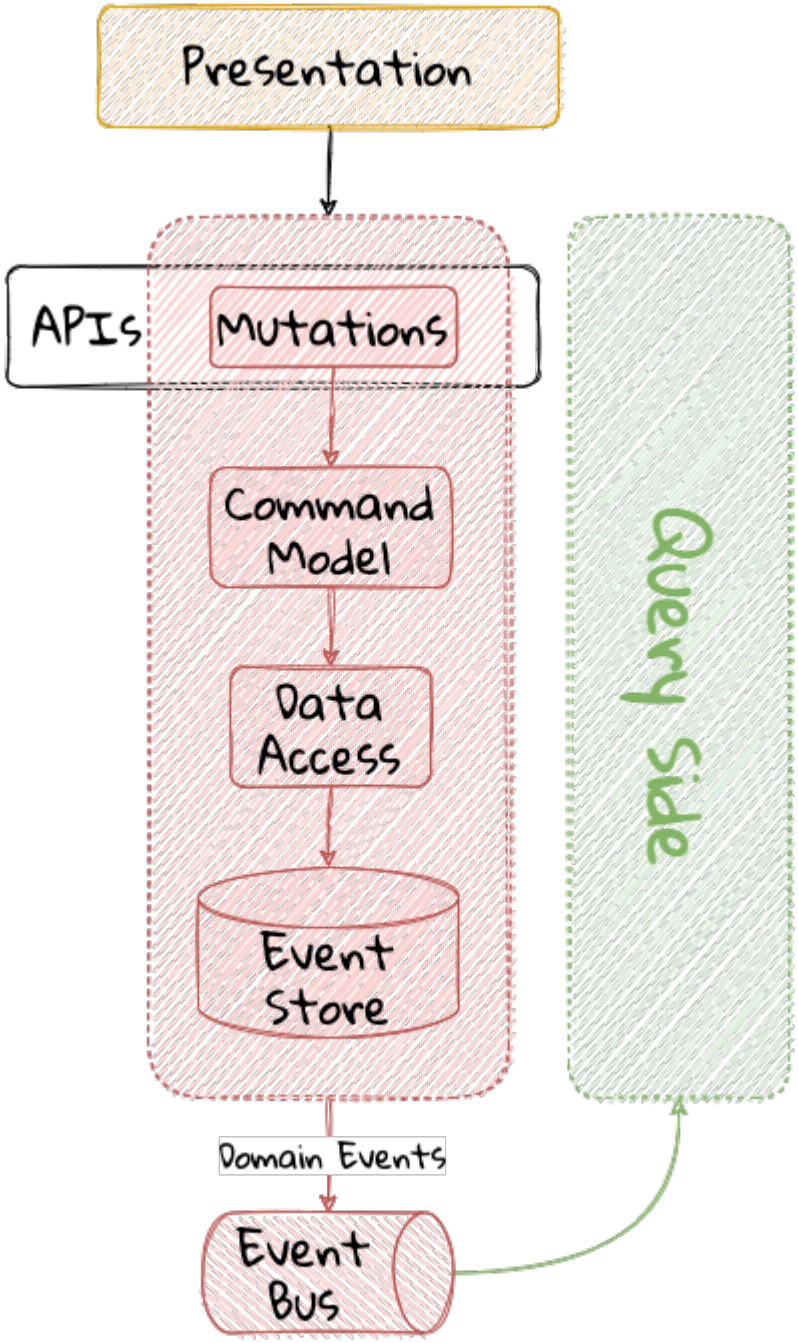


Figure 1- 58. CQRS application — command side

The high level sequence on the command side is described here:

1. A request to mutate state (command) is received.
2. In an event-sourced system, the command model is constructed by replaying existing events that have occurred for that instance. In a state-stored system, we would simply restore state by reading state from the persistence store.
3. If business invariants (validations) are satisfied, one or more domain events are published.
4. In an event-sourced system, the domain event is persisted on the command side. In a state-stored system, we would update the state of the instance in the persistence store.
5. The external world is notified by publishing these domain events onto an event bus. The event bus is an infrastructure component onto which events are published.

Let's look at how we can implement this in the context of our Letter of Credit (LC) issuance application.

5.3.1. Tooling choices

Implementing CQRS does not require the use of any framework. Greg Young, who is considered the father of the CQRS pattern, advises against rolling our own CQRS framework in this essay[16], which is worth taking a look at. Using a *good* framework can help enhance developer effectiveness and accelerate the delivery of business functionality, while abstracting the low-level plumbing and non-functional requirements without limiting flexibility. In this book we will make use of the Axon Framework[17] to implement application functionality as we have real-world experience of having used it in large scale enterprise development with success. There are other frameworks that work comparably, like Lagom Framework[18] and Eventuate[19] that are worth exploring as well.

5.3.2. Bootstrapping the application

To get started, let's create a simple spring boot application. There are several ways to do this. You can always use the Spring starter application at https://start.spring.io to create this application. Here, we will make use of the `spring` CLI to bootstrap the application.

To install the `spring` CLI for your platform, please refer to the detailed instructions at https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started.installing.

To bootstrap the application, use the following command:

```
spring init \
      --dependencies 'web,data-jpa,lombok,validation,h2,actuator' \
      --name lc-issuance-api \
      --artifactId lc-issuance-api \
      --groupId com.example.api \
      --packaging jar \
      --description 'LC Issuance API' \
      --package-name com.example.api \
      --force
```

The entire command is split into multiple lines for better readability. Unix based operating systems requires us to use backslash [\ ] character to split the command into multiple lines. If you are using Windows OS, then please make sure to replace the backslash character with back-tick [`] character before running the command.

This should create a file named `lc-issuance-api.zip` in the current directory. Unzip this file to a location of your choice and add a dependency on the Axon framework in the `dependencies` section of the `pom.xml` file:

```
    <dependency>
        <groupId>org.axonframework</groupId>
        <artifactId>axon-spring-boot-starter</artifactId>
        <version>${axon-framework.version}</version> ①
    </dependency>
```

① You may need to change the version. We are at version `4.5.3` at the time of writing this book.

Also, add the following dependency on the `axon-test` library to enable unit testing of aggregates:

```
    <dependency>
        <groupId>org.axonframework</groupId>
        <artifactId>axon-test</artifactId>
        <scope>test</scope>
        <version>${axon-framework.version}</version>
    </dependency>
```

With the above set up, you should be able to run the application and start implementing the LC issuance functionality.

5.3.3. Identifying commands

From the eventstorming session in the previous chapter, we have the following commands to start with:

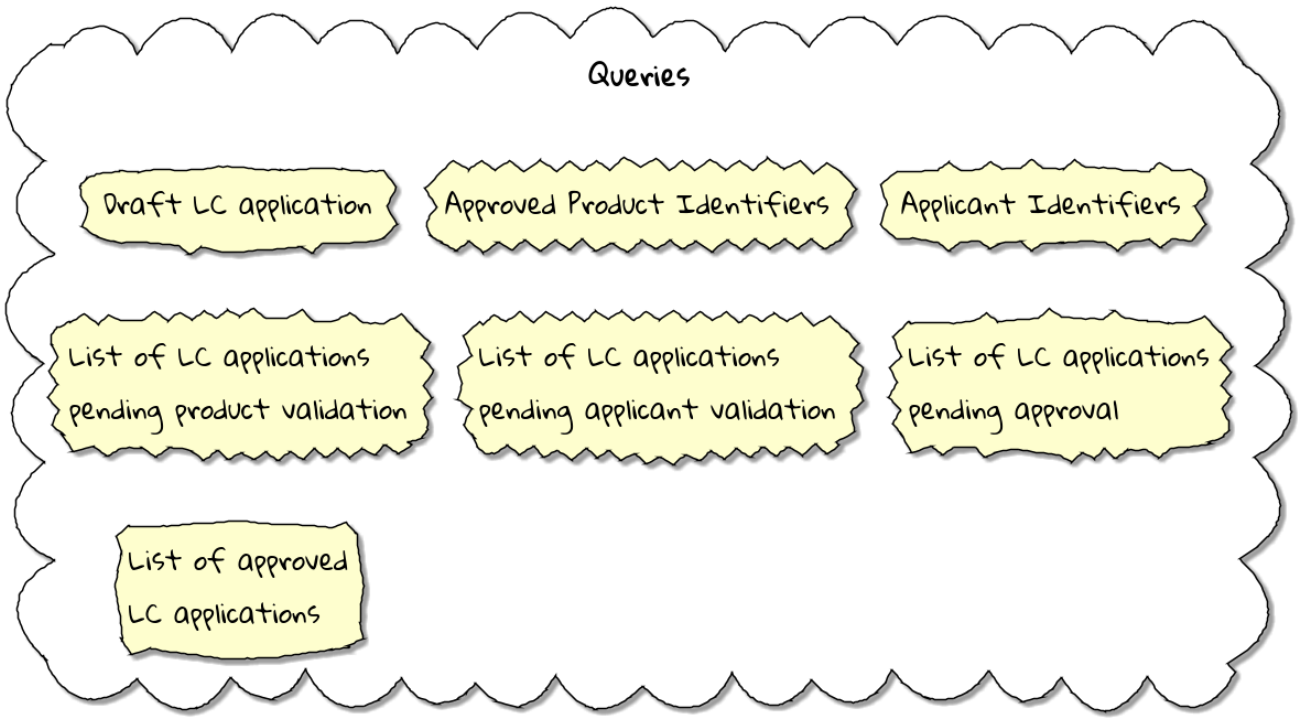


Figure 1- 59. Identified commands

Commands are always directed to an aggregate (the root entity) for processing (handling). This means that we need to resolve each of these commands to be handled by an aggregate. While the sender of the command does not care which component within the system handles it, we need to

decide which aggregate will handle each command. It is also important to note that any given command can only be handled by a single aggregate within the system. Let's look at how to group these commands and assign them to aggregates. To be able to do that, we need to identify the aggregates in the system first.

5.3.4. Identifying aggregates

Looking at the output of the eventstorming session of our LC (Letter of Credit) application, one potential grouping of commands can be as follows:

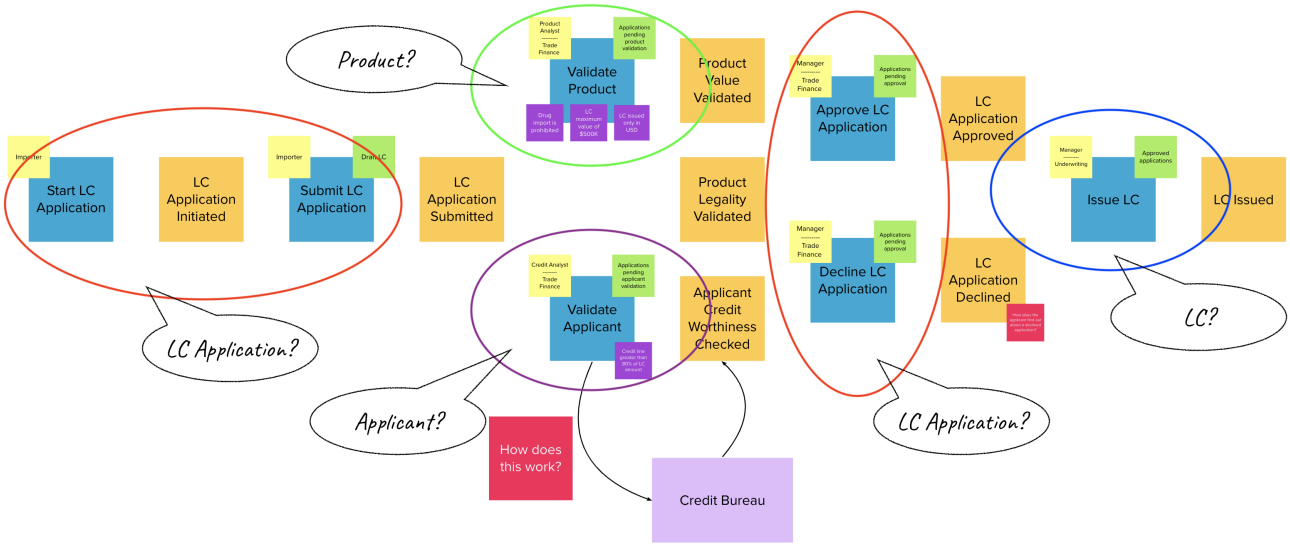


Figure 1- 60. First cut attempt at aggregate design

Before we arrive at aggregates, the grouping above allows us to identify entities. Some or all of these entities may be aggregates (For a more detailed explanation on the difference between aggregates and entities, please refer to Chapter 2).

At first glance, it appears that we have four potential aggregates to handle these commands:

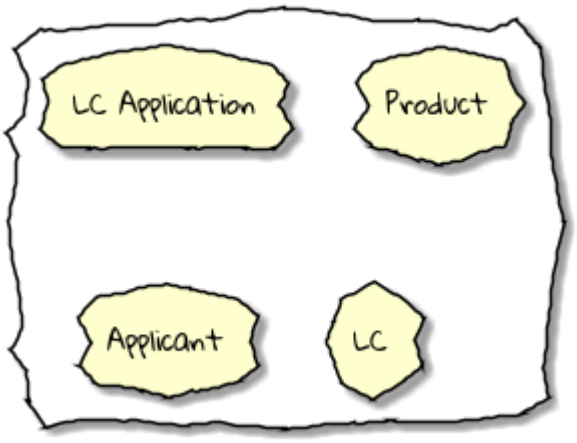
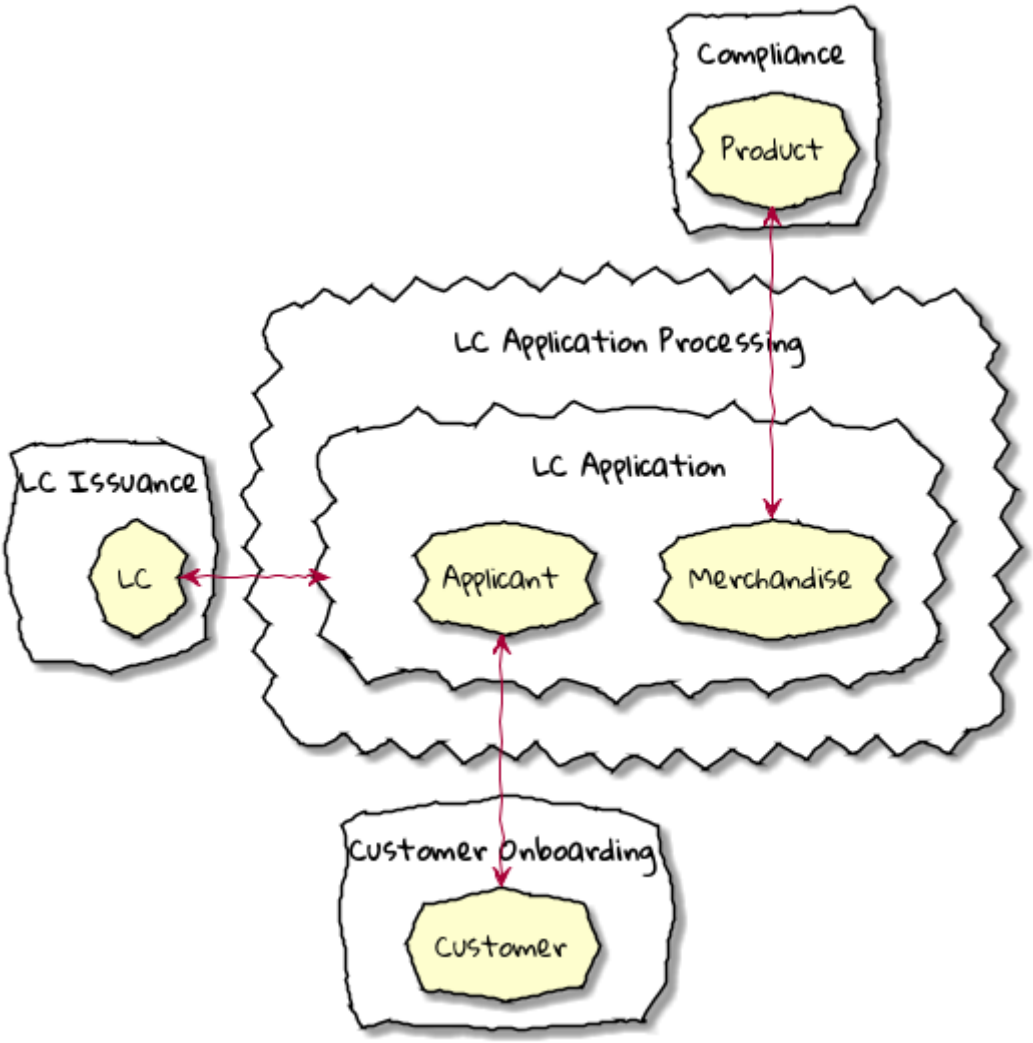


Figure 1- 61. Potential aggregates at first glance

However, it is a bit more nuanced than that. Before we conclude this conversation on aggregates, let's also examine our current organizational structures because they can and will play a very influential role in how we choose aggregates. When implementing the solution, these organizational structures decompose into bounded contexts, so let's also examine how that works.

5.3.5. Discovering bounded contexts

Our current organization is segregated to handle the business functions outlined here:



As a starting point, we can use these business functions as natural boundaries to act as bounded contexts for our solution. We may evolve this design in the future as we gain more understanding of the problem and the solution, but for now, this will suffice. Aggregates live within the confines of a single bounded context, so we need to correlate the two concepts. Let's look at how this works.

5.3.6. Correlating aggregates to bounded contexts

If we examine the lifecycle of the letter of credit (LC) as a whole, we notice the structure outlined here. Each of these bounded contexts work with the same entities, but call them by different names making use of their own ubiquitous language. The context map for the system looks like this:

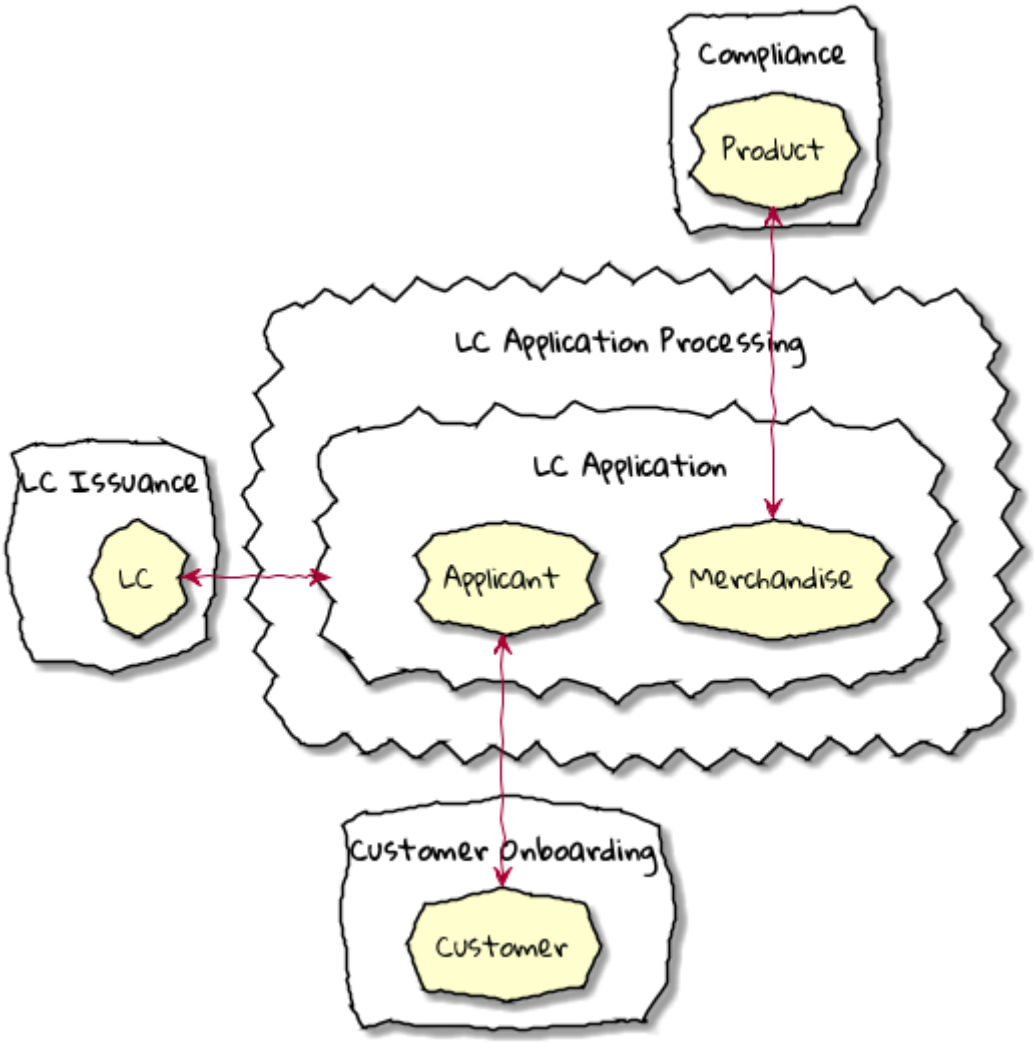


Figure 1- 62. Relationship between bounded contexts

Notice how the `Customer` in the Customer Onboarding bounded context is called an `Applicant` in the LC Application Processing context. Similarly, Compliance works with a `Product` entity, whereas LC Application Processing calls it `Merchandise`. Within the LC Application Processing bounded context, we always work within the purview of an `LC Application`, not directly with either the `Applicant` or the `Merchandise`. This leads us to the conclusion (at least for now), that the `Applicant` and `Merchandise` entities are not aggregates within the LC Application Processing context. The `LC Application` entity acts as the **aggregate** for this bounded context. Furthermore, it is at the root of the aggregate hierarchy, hence it is termed as the **aggregate root**.

The terms aggregate and aggregate root are sometimes used interchangeably to mean the same thing. While both aggregates and aggregate roots handle commands, only one aggregate can exist as the root in a given context, and it encapsulates access to its child aggregates.

It is important to note that entities may be required to be treated as aggregates in a different bounded context and this kind of treatment is entirely context dependent.

When we look at the output of our eventstorming session, the `LC Application` transitions to become an `LC` much later in the lifecycle in the Issuance context. Our focus right now is to optimize and automate the LC application flow of the overall issuance process. Now that we have settled on

working with the `LC Application` aggregate (root), let's start writing our first command to see how this manifests itself in code.

5.3.7. Test-driving the system

While we have a reasonably good conceptual understanding of the system, we are still in the process of refining this understanding. Test-driving the system allows us to exercise our understanding by acting as the first client of the solution that we are producing.

The practice of test-driving the system is very well illustrated in the best-selling book — *Growing Object-Oriented Software, Guided by Tests* by authors Nat Price and Steve Freeman. This is worth looking at, to gain a deeper understanding of this practice.

So let's start with the first test. To the external world, an event-driven system typically works in a manner depicted below:

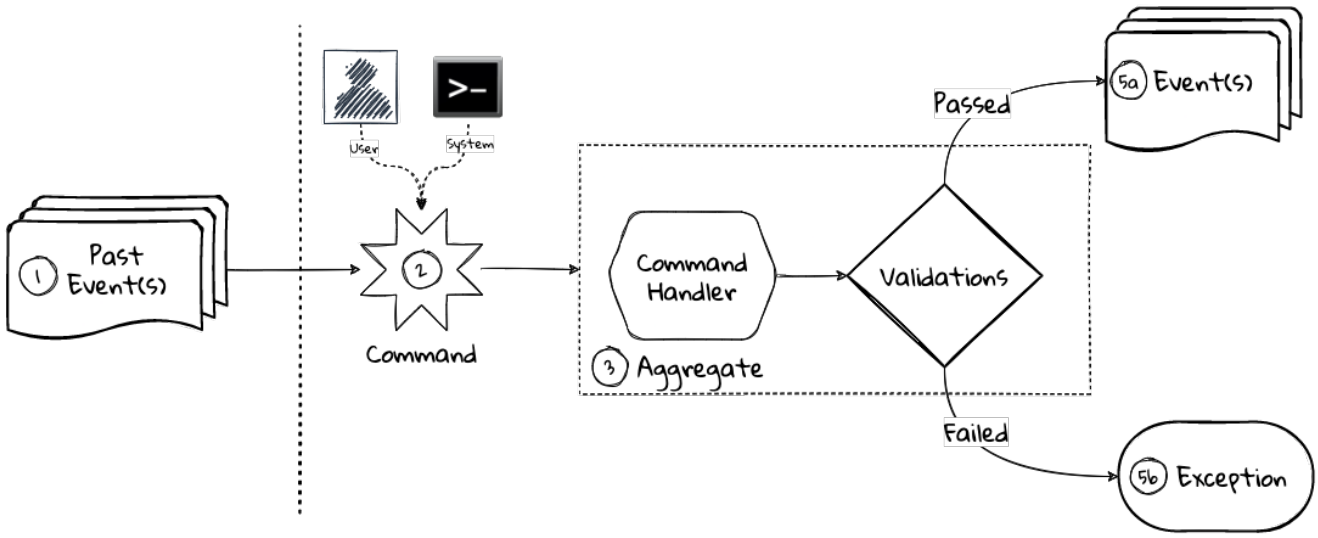


Figure 1- 63. An event-driven system

1. An optional set of domain events may have occurred in the past.
2. A command is received by the system (initiated manually by a user or automatically by a part of the system), which acts as a stimulus.
3. The command is handled by an aggregate which then proceeds to validate the received command to enforce invariants (structural and domain validations).
4. The system then reacts in one of two ways:
    1. Emit one or more events
    2. Throw an exception

The Axon framework allows us to express tests in the following form.

The code snippets shown in this chapter are excerpts to highlight significant concepts and techniques. For the full working example, please refer to the accompanying source code for this chapter (included in the ch05 directory).

```
public class LCApplicationAggregateTests {

    private FixtureConfiguration<LCApplication> fixture;                          ①

    @BeforeEach
    void setUp() {
        fixture = new AggregateTestFixture<>(LCApplication.class);                ②
    }

    @Test
    void shouldPublishLCApplicationCreated() {
        fixture.given()                                                           ③

                .when(new CreateLCApplicationCommand())                           ④

                .expectEventsMatching(exactSequenceOf(                            ⑤
                        messageWithPayload(any(LCApplicationCreatedEvent.class)), ⑥
                        andNoMore()                                               ⑦
                ));
    }
}
```

① `FixtureConfiguration` is an Axon framework utility to aid testing of aggregate behaviour using a BDD style given-when-then syntax.

② `AggregateTestFixture` is a concrete implementation of `FixtureConfiguration` where you need to register your aggregate class — `LCApplication` in our case as the candidate to handle commands directed to our solution.

③ Since this is the start of the business process, there are no events that have occurred thus far. This is signified by the fact that we do not pass any arguments to the `given` method. In other examples we will discuss later, there will likely be events that have already occurred prior to receiving this command.

④ This is where we instantiate a new instance of the command object. Command objects are usually similar to data transfer objects, carrying a set of information. This command will be routed to our aggregate for handling. We will take a look at how this works in detail shortly.

⑤ Here we are declaring that we expect events matching an exact sequence.

⑥ Here we are expecting an event of type `LCApplicationCreated` to be emitted as a result of successfully handling the command.

⑦ We are finally saying that we do not expect any more events — which means that we expect exactly one event to be emitted.

5.3.8. Implementing the command

The `CreateLCApplicationCommand` in the previous simplistic example does not carry any state. Realistically, the command will likely look something like what is depicted as follows:

```
import lombok.Data;

@Data
public class CreateLCApplicationCommand {  ①

    private LCApplicationId id;            ②
    private ClientId clientId;
    private Party applicant;               ③
    private Party beneficiary;
    private AdvisingBank advisingBank;     ③
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;

}
```

① The command class. When naming commands, we typically use an imperative style i.e. they usually begin with a verb denoting the action required. Note that this is a data transfer object. In other words, it is simply a bag of data attributes. Also note how it is devoid of any logic (at least at the moment).

② The identifier for the LC Application. We are assuming client generated identifiers in this case. The topic of using server-generated versus client-generated identifiers is out of scope for the subject of this book. You may use either depending on what is advantageous in your context. Also note that we are using a strong type for the identifier `LCApplicationId` as opposed to a primitive such as a numeric or a string value. It is also common in some cases to use UUIDs as the identifier. However, we prefer using strong types to be able to differentiate between identifier types. Notice how we are using a type `ClientId` to represent the creator of the application.

③ The `Party` and `AdvisingBank` types are complex types to represent those concepts in our solution. Care should be taken to consistently use names that are relevant in the problem (business) domain as opposed to using names that only make sense in the solution (technology) domain. Note the attempt to make use of the *ubiquitous language* of the domain experts in both cases. This is a practice that we should always be conscious of when naming things in the system.

It is worth noting that the `merchandiseDescription` is left as a primitive `String` type. This may feel contradictory to the commentary we present above. We will address this in the upcoming section on Structural validations.

Now let's look at what the event we will emit as a result of successfully processing the command will look like.

5.3.9. Implementing the event

In an event-driven system, mutating system state by successfully processing a command usually results in a domain event being emitted to signal the state mutation to the rest of the system. A simplified representation of a real-world `LCApplicationCreatedEvent` is shown here:

```
import lombok.Data;

@Data
public class LCApplicationCreatedEvent {    ①

    private LCApplicationId id;
    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;

}
```

① The event type. When naming events, we typically use names in the past tense to denote things that have already occurred and are to be accepted unconditionally as empirical facts that cannot be changed.

You will likely notice that the structure of the event is currently identical to that of the command. While this is true in this case, it may not always be that way. The amount of information that we choose to disclose in an event is context-dependent. It is important to consult with domain experts when publishing information as part of events. One may choose to withhold certain information in the event payload. For example, consider a `ChangePasswordCommand` which contains the newly changed password. It might be prudent to not include the changed password in the resulting `PasswordChangedEvent`.

We have looked at the command and the resulting event in the previous test. Let's look at how this is implemented under the hood by looking at the aggregate implementation.

5.3.10. Designing the aggregate

The aggregate is the place where commands are handled and events are emitted. The good thing about the test that we have written is that it is expressed in a manner that hides the implementation details. But let's look at the implementation to be able to appreciate how we can get our tests to pass and meet the business requirement.

```
public class LCApplication {

    @AggregateIdentifier                                                    ①
    private LCApplicationId id;

    @SuppressWarnings("unused")
    private LCApplication() {
        // Required by the framework
    }

    @CommandHandler                                                         ②
    public LCApplication(CreateLCApplicationCommand command) {              ③
        // TODO: perform validations here
        AggregateLifecycle.apply(new LCApplicationCreatedEvent(command.getId()));   ④
    }

    @EventSourcingHandler                                                   ⑤
    private void on(LCApplicationCreatedEvent event) {
        this.id = event.getId();
    }
}
```

① The aggregate identifier for the `LCApplication` aggregate. For an aggregate, the identifier uniquely identifies one instance from another. For this reason, all aggregates are required to declare an identifier and mark it so using the `@AggregateIdentifier` annotation provided by the framework.

② The method that is handling the command needs to be annotated with the `@CommandHandler` annotation. In this case, the command handler happens to be the constructor of the class given that this the first command that can be received by this aggregate. We will see examples of subsequent commands being handled by other methods later in the chapter.

③ The `@CommandHandler` annotation marks a method as being a command handler. The exact command that this method can handle needs to be passed as a parameter to the method. Do note that there can only be one command handler in the **entire** system for any given command.

④ Here, we are emitting the `LCApplicationCreatedEvent` using the `AggregateLifecycle` utility provided by the framework. In this very simple case, we are emitting an event unconditionally on receipt of the command. In a real-world scenario, it is conceivable that a set of validations will be performed before deciding to either emit one or more events or failing the command with an exception. We will look at more realistic examples later in the chapter.

⑤ The need for the `@EventSourcingHandler` and its role are likely very unclear at this time. We will explain the need for this in detail in an upcoming section of this chapter.

This was a whirlwind introduction to a simple event-driven system. We still need to understand the role of the `@EventSourcingHandler`. To understand that, we will need to appreciate how aggregate persistence works and the implications it has on our overall design.

5.4. Persisting aggregates

When working with any system of even moderate complexity, we are required to make interactions durable. That is, interactions need to outlast system restarts, crashes, etc. So the need for persistence is a given. While we should always endeavour to abstract persistence concerns from the rest of the system, our persistence technology choices can have a significant impact on the way we architect our overall solution. We have a couple of choices in terms of how we choose to persist aggregate state that are worth mentioning:

1. State stored
2. Event sourced

Let's examine each of these techniques in more detail below:

5.4.1. State stored aggregates

Saving current values of entities is by far the most popular way to persist state — thanks to the immense popularity of relational databases and object-relational mapping (ORM) tools like Hibernate. And there is good reason for this ubiquity. Until recently, a majority of enterprise systems used relational databases almost as a default to create business solutions, with ORMs arguably providing a very convenient mechanism to interact with relational databases and their object representations. For example, for our `LCApplication`, it is conceivable that we could use a relational database with a structure that would look something like below:

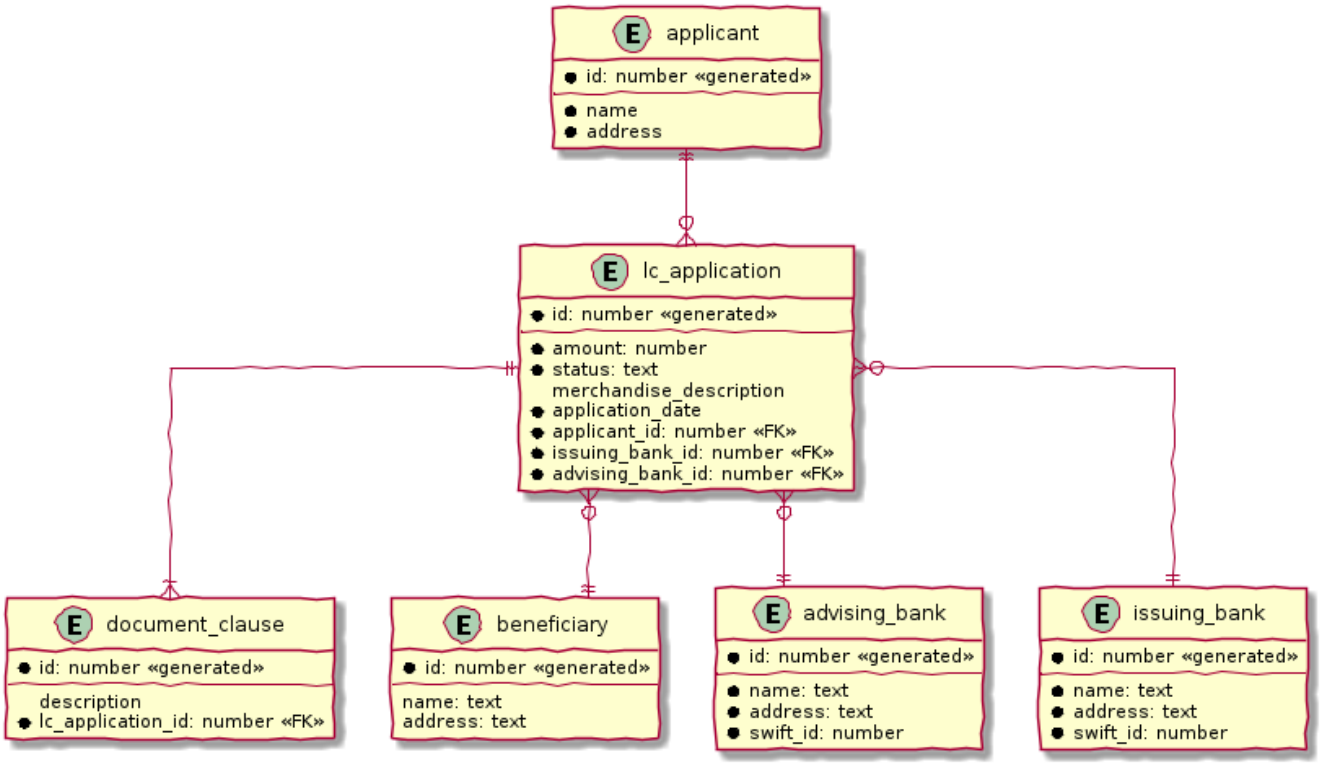


Figure 1- 64. Typical entity relationship model

Irrespective of whether we choose to use a relational database or a more modern NoSql store — for instance, a document store, key-value store, column family store, etc., the style we use to persist information remains more or less the same — which is to store the current values of the attributes

of the said aggregate/entity. When the values of attributes change, we simply overwrite old values with newer ones i.e. we store the current state of aggregates and entities — hence the name *state stored*. This technique has served us very well over the years, but there is at least one more mechanism that we can use to persist information. We will look at this in more detail below.

5.4.2. Event sourced aggregates

Developers have also been relying on logs for a variety of diagnostic purposes for a very long time. Similarly, relational databases have been employing commit logs to store information durably almost since their inception. However, developers' use of logs as a first class persistence solution for structured information in mainstream systems remains extremely rare.

A log is an extremely simple, append-only sequence of immutable records ordered by time. The diagram here illustrates the structure of a log where records are written sequentially. In essence, a log is an append-only data structure as depicted here:.

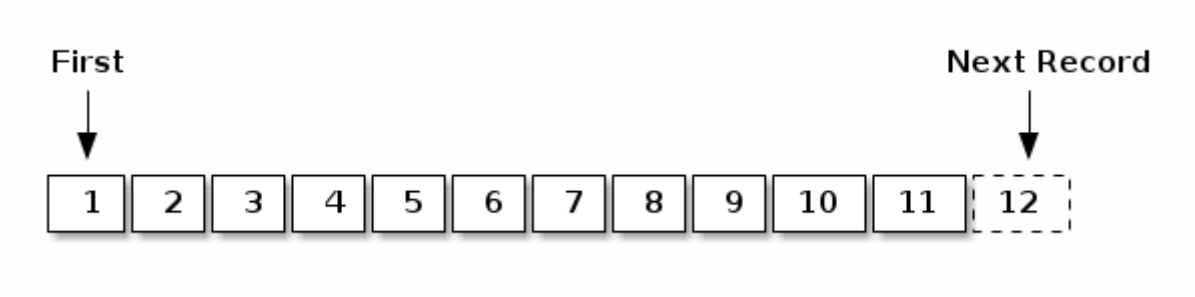


Figure 1- 65. The log data structure

Writing to a log as compared to a more complex data structure like a table is a relatively simple and fast operation and can handle extremely high volumes of data while providing predictable performance. Indeed, a modern event streaming platform like Kafka makes use of this pattern to scale to support extremely high volumes. We do feel that this can be applied to act as a persistence store when processing commands in mainstream systems because this has benefits beyond the technical advantages listed above.Consider the example of an online order flow below:

User Action	Traditional Store	Event Store
Add milk to cart	Order 123: Milk in cart	E1: Cart#123 created E2: Milk added to cart
Add white bread to cart	Order 123: Milk, White bread in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart
Remove White bread from cart	Order 123: Milk in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart

User Action	Traditional Store	Event Store
Add Wheat bread to cart	Order 123: Milk, Wheat bread in cart	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart E5: Wheat bread added to cart
Confirm cart checkout	Order 123: Ordered Milk, Wheat bread	E1: Cart#123 created E2: Milk added to cart E3: White bread added to cart E4: White bread removed from cart E5: Wheat bread added to cart E6: Order 123 confirmed

As you can see, in the event store, we continue to have full visibility of all user actions performed.This allows us to reason about these behaviors more holistically.In the traditional store, we lost the information that the user replaced white with wheat bread.While this does not impact the order itself, we lose the opportunity to gather insights from this user behavior.We recognize that this information can be captured in other ways using specialized analytical solutions, however, the event log mechanism provides a natural way to do this without requiring any additional effort. It also acts as an audit log providing full history of all events that have occurred thus far.This fits well with the essence of domain-driven design where we are constantly exploring ways in which to reduce complexity.

However, there are implications to persisting data in the form of a simple event log.Before processing any command, we will need to hydrate past events in exact order of occurrence and reconstruct aggregate state to allow us to perform validations.For example, when confirming checkout, just having the ordered set of elapsed events will not suffice.We still need to compute the exact items that are in the cart before allowing the order to be placed.This *event replay* to restore aggregate state (at least those attributes that are required to validate said command) is necessary before processing that command.For example, we need to know which items are in the cart currently before processing the `RemoveItemFromCartCommand`.This is illustrated in the following table:

Elapsed Events	Aggregate State	Command	Event(s) Emitted
—	—	Add item: milk	E1: Cart#*123* created E2: Milk added
E1: Cart#123 created E2: Milk added	**Cart Items**: Milk	Add item: white bread	E2: White bread added
E1: Cart#123 created E2: Milk added E3: White bread added	**Cart Items**: Milk, White Bread	Remove item: white bread	E3: White bread removed
E1: Cart#123 created E2: Milk added E3: White bread added E4: White bread removed	**Cart Items**: Milk	Add item: wheat bread	E4: Wheat bread added

Elapsed Events	Aggregate State	Command	Event(s) Emitted
E1: Cart#123 created E2: Milk added E3: White bread added E4: White bread removed E5: Wheat bread added	**Cart Items**: Milk Wheat bread	Confirm checkout for Cart#123	E5: Order created!

The corresponding source code for the whole scenario is illustrated in the following code snippet:

```
public class Cart {

    private boolean isNew;
    private CartItems items;
    //..

    private Cart() {                                                    ①
        // Required by the framework
    }

    @CommandHandler
    public void addItem(AddItemToCartCommand command) {
        // Business validations here
        if (this.isNew) {
            apply(new CartCreatedEvent(command.getId()));               ②
        }
        apply(new ItemAddedEvent(id, command.getItem()));               ②
    }

    @CommandHandler
    public void removeItem(RemoveItemFromCartCommand command) {
        // Business validations here
        apply(new ItemRemovedEvent(id, commmand.getItem()));
    }

    @CommandHandler
    public void checkout(ConfirmCheckoutCommand command) {
        // Business validations here
        apply(new OrderCreatedEvent(this.items));
    }

    @EventSourcingHandler                                               ③
    private void on(CartCreatedEvent event) {
        this.id = event.getCartId();
        this.items = new CartItems();
        this.isNew = true;
    }

    @EventSourcingHandler                                               ③
    private void on(ItemAddedEvent event) {
```

```
        this.items.add(event.getItem());
        this.isNew = false;
    }

    @EventSourcingHandler                                    ③
    private void on(ItemRemovedEvent event) {
        this.items.remove(event.getItem());
    }

    @EventSourcingHandler                                    ③
    private void on(CheckoutConfirmedEvent event) {
        // ..
    }
}
```

① Before processing any command, the aggregate loading process commences by first invoking the no-args constructor.For this reason, we need the no-args constructor to be **empty** i.e. it should **not** have any code that restores state.State restoration **must** happen only in those methods that trigger an event replay.In the case of the Axon framework, this translates to methods embellished with the `@EventSourcingHandler` annotation.

② It is important to note that it is possible (but not necessary) to emit **more than one event** after processing a command.This is illustrated in the first instance of the `AddItemCommand` in the previous code where we emit `CartCreatedEvent` and `ItemAddedEvent`.Command handlers do not mutate state of the aggregate.They only make use of existing aggregate state to enforce invariants (validations) and emit events if those invariants hold true.

③ The loading process continues through the invocation of event sourcing handler methods in exactly the order of occurrence for that aggregate instance.Event sourcing handlers are only needed to hydrate aggregate state on the basis of past events.This means that they usually are devoid of any business (conditional) logic.It goes without saying that these methods do not emit any events.Event emission is restricted to happen within command handlers when invariants are successfully enforced.

When working with event sourced aggregates, it is very important to be disciplined about the kind of code that one can write:

Type of Method	State Restoration	Business Logic	Event Emission
`@CommandHandler`	No	Yes	Yes
`@EventSourcingHandler`	Yes	No	No

If there are a large number of historic events to restore state, the aggregate loading process can become a time-consuming operation — directly proportional to the number of elapsed events for that aggregate.There are techniques (like snapshotting) we can employ to overcome this.We will cover this in more detail in Chapter 11 – Non-Functional Requirements.

5.4.3. Persistence technology choices

If you are using a state store to persist your aggregates, using your usual evaluation process for choosing your persistence technology should suffice. However, if you are looking at event-sourced

aggregates, the decision can be a bit more nuanced. In our experience, even a simple relational database can do the trick. Indeed, we had made use of a relational database to act as an event store for a high volume transactional application with billions of events. This setup worked just fine for us. It is worth noting that we were only using the event store to insert new events and loading events for a given aggregate in sequential order. However, there are a multitude of specialized technologies that have been purpose built to act as an event store that support several other value-added features such as time travel, full event replay, event payload introspection, etc. If you have such requirements, it might be worth considering other options such as NoSql databases (document stores like MongoDB or column family stores like Cassandra) or purpose-built commercial offerings such as EventStoreDB[20] and Axon Server[21] to evaluate feasibility in your context.

5.4.4. Which persistence mechanism should we choose?

Now that we have a reasonably good understanding of the two types of aggregate persistence mechanisms (state-stored and event-sourced), it begs the question of which one we should choose. We list a few benefits of using event sourcing below:

- We get to use the events as a **natural audit log** in high compliance scenarios.
- It provides the ability to perform **more insightful analytics** on the basis of the fine-grained events data.
- It arguably produces more flexible designs when we work with an system based on **immutable events** — because the complexity of the persistence model is capped. Also, there is no need to deal with complex ORM impedance mismatch problems.
- The domain model is much more **loosely coupled** with the persistence model — enabling it to evolve mostly independently from the persistence model.
- Enables going back in time to be able to create **adhoc views and reports** without having to deal with upfront complexity.

On the flip side, these are some challenges that you might have to consider when implementing an event sourced solution:

- Event sourcing requires a **paradigm shift**. Which means that development and business teams will have to spend time and effort understanding how it works.
- The persistence model does not store state directly. This means that **adhoc querying** directly on the persistence model can be a lot more **challenging**. This can be alleviated by materializing new views, however there is added complexity in doing that.
- Event sourcing usually tends to work very well when implemented in conjunction with **CQRS** which arguably may add more complexity to the application. It also requires applications to pay closer attention to strong vs **eventual consistency** concerns.

Our experiences indicate that event sourced systems bring a lot of benefits in modern event-driven systems. However, you will need to be cognizant of the considerations presented above in the context of your own ecosystems when making persistence choices.

5.5. Enforcing policies

When processing commands, we need to enforce policies or rules. Policies come in two broad categories:

- Structural rules — those that enforce that the syntax of the dispatched command is valid.
- Domain rules — those that enforce that business rules are adhered to.

It may also be prudent to perform these validations in different layers of the system.And it is also common for some or all of these policy enforcements to be repeated in more than one layer of the system.However, the important thing to note is that before a command is successfully handled, all these policy enforcements are uniformly applied.Let's look at some examples of these in the upcoming section.

5.5.1. Structural validations

Currently, to create an LC application, one is required to dispatch a `CreateLCApplicationCommand`. While the command dictates a structure, none of it is enforced at the moment. Let's correct that.

To be able to enable validations declaratively, we will make use of the JSR-303 bean validation libraries. We can add that easily using the `spring-boot-starter-validation` dependency to our `pom.xml` file as shown here:

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
```

Now we can add validations to the command object using the JSR-303 annotations as depicted below:

```
import lombok.Data;
import javax.validation.*;
import javax.validation.constraints.*;

@Data
public class CreateLCApplicationCommand {

    @NotNull
    private LCApplicationId id;

    @NotNull
    private ClientId clientId;

    @NotNull
    @Valid
    private Party applicant;

    @NotNull
    @Valid
    private Party beneficiary;

    @NotNull
    @Valid
    private AdvisingBank advisingBank;

    @Future
    private LocalDate issueDate;

    @Positive
    private MonetaryAmount amount;

    @NotBlank
    private String merchandiseDescription;
}
```

Most structural validations can be accomplished using the built-in validator annotations. It is also possible to create custom validators for individual fields or to validate the entire object (for example, to validate inter-dependent attributes). For more details on how to do this, please refer to the bean validation specification at https://beanvalidation.org/2.0/ and the reference implementation at http://hibernate.org/validator/.

5.5.2. Business rule enforcements

Structural validations can be accomplished using information that is already available in the command.However, there is another class of validations that requires information that is not present in the incoming command itself.This kind of information can be present in one of two places: within the aggregate that we are operating on or outside of the aggregate itself, but made available within the bounded context.

Let's look at an example of a validation that requires state present within the aggregate. Consider the example of submitting an LC.While we can make several edits to the LC when it is in draft state, no changes can be made after it is submitted.This means that we can only submit an LC once.This act of submitting the LC is achieved by issuing the `SubmitLCApplicationCommand` as shown in the artifact from the eventstorming session:

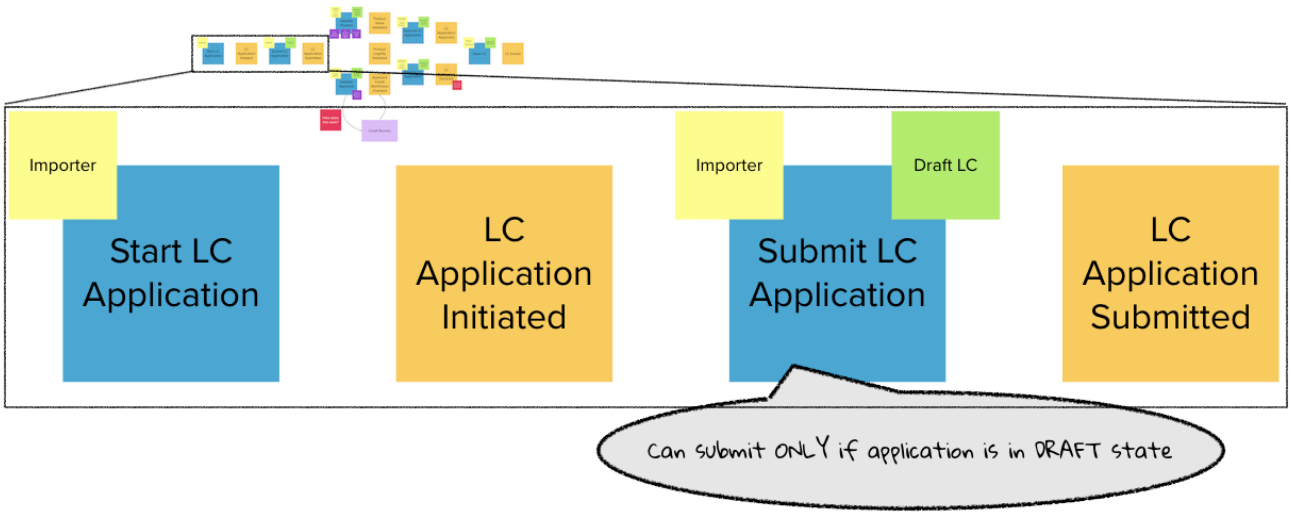


Figure 1- 66. Validations during the submit LC process

Let's begin with a test to express our intent:

```
class LCApplicationAggregateTests {
    //..
    @Test
    void shouldAllowSubmitOnlyInDraftState() {
        final LCApplicationId applicationId = LCApplicationId.randomId();

        fixture.given(new LCApplicationCreatedEvent(applicationId))            ①
                .when(new SubmitLCApplicationCommand(applicationId))           ②
                .expectEvents(new LCApplicationSubmittedEvent(applicationId)); ③
    }
}
```

① Given that the `LCApplicationCreatedEvent` has already occurred — in other words, the LC application is already created.

② When we try to submit the application by issuing the `SubmitLCApplicationCommand` for the same application.

③ We expect the `LCApplicationSubmittedEvent` to be emitted.

The corresponding implementation will look something like:

```
class LCApplication {
    // ..
    @CommandHandler
    public void submit(SubmitLCApplicationCommand command) {
        apply(new LCApplicationSubmittedEvent(id));
    }
}
```

The implementation above allows us to submit an LC application unconditionally — more than once. However, we want to restrict users to be able to submit only once. To be able to do that, we need to remember that the LC application has already been submitted. We can do that in the `@EventSourcingHandler` of the corresponding events as shown below:

```
class LCApplication {
    // ..
    @EventSourcingHandler
    private void on(LCApplicationSubmittedEvent event) {
        this.state = State.SUBMITTED; ①
    }
}
```

① When the `LCApplicationSubmittedEvent` is replayed, we set the state of the `LCApplication` to `SUBMITTED`.

While we have remembered that the application has changed to be in `SUBMITTED` state, we are still not preventing more than one submit attempt. We can fix that by writing a test as shown below:

```
class LCApplicationAggregateTests {
    @Test
    void shouldNotAllowSubmitOnAnAlreadySubmittedLC() {
        final LCApplicationId applicationId = LCApplicationId.randomId();

        fixture.given(
                new LCApplicationCreatedEvent(applicationId),          ①
                new LCApplicationSubmittedEvent(applicationId))        ①

                .when(new SubmitLCApplicationCommand(applicationId))   ②

                .expectException(AlreadySubmittedException.class)      ③
                .expectNoEvents();                                     ④
    }
}
```

① The `LCApplicationCreatedEvent` and `LCApplicationSubmittedEvent` have already happened — which means that the `LCApplication` has been submitted once.

② We now dispatch another `SubmitLCApplicationCommand` to the system.

③ We expect an `AlreadySubmittedException` to be thrown.

④ We also expect no events to be emitted.

The implementation of the command handler to make this work is shown below:

```
class LCApplication {
    // ..
    @CommandHandler
    public void submit(SubmitLCApplicationCommand command) {
        if (this.state != State.DRAFT) {                                    ①
            throw new AlreadySubmittedException("LC is already submitted!");
        }
        apply(new LCApplicationSubmittedEvent(id));
    }
}
```

① Note how we are using the state attribute from the `LCApplication` aggregate to perform the validation. If the application is not in `DRAFT` state, we fail with the `AlreadySubmittedException` domain exception.

Let's also look at an example where information needed to perform the validation is not part of either the command or the aggregate. Let's consider the scenario where country regulations prohibit transacting with a set of so called *sanctioned* countries. Changes to this list of countries may be affected by external factors. Hence it does not make sense to pass this list of sanctioned countries as part of the command payload. Neither does it make sense to maintain it as part of every single aggregate's state — given that it can change (albeit very infrequently). In such a case, we may want to consider making use of a command handler that is outside the confines of the aggregate class. Thus far, we have only seen examples of `@CommandHandler` methods within the aggregate. But the `@CommandHandler` annotation can appear on any other class external to the aggregate. However, in such a case, we need to load the aggregate ourselves. The Axon framework provides a `org.axonframework.modelling.command.Repository` interface to allow us to do that. It is important to note that this `Repository` is distinct from spring framework's interface that is part of the spring data libraries. An example of how this works is shown below:

```
import org.axonframework.modelling.command.Repository;

class MyCustomCommandHandler {

    private final Repository<LCApplication> repository;                ①

    MyCustomCommandHandler(Repository<LCApplication> repository) {
        this.repository = repository;                                  ①
    }

    @CommandHandler
    public void handle(SomeCommand command) {
        Aggregate<LCApplication> application
            = repository.load(command.getAggregateId());               ②
        // Command handling code
    }

    @CommandHandler
    public void handle(AnotherCommand command) {
        Aggregate<LCApplication> application
            = repository.load(command.getAggregateId());
        // Command handling code
    }
}
```

① We are injecting the Axon `Repository` to allow us to load aggregates. This ws not required previously because the `@CommandHandler` annotation appeared on aggregate methods directly.

② We are using the `Repository` to load aggregates and work with them. The `Repository` interface supports other convenience methods to work with aggregates. Please refer to the Axon framework documentation for more usage examples.

Coming back to the sanctioned countries example, let's look at how we need to set up the test slightly differently:

```
public class CreateLCApplicationCommandHandlerTests {
    private FixtureConfiguration<LCApplication> fixture;

    @BeforeEach
    void setUp() {
        final Set<Country> sanctioned = Set.of(SOKOVIA);
        fixture = new AggregateTestFixture<>(LCApplication.class);              ①

        final Repository<LCApplication> repository = fixture.getRepository();  ②

        CreateLCApplicationCommandHandler handler =
                new CreateLCApplicationCommandHandler(repository, sanctioned);  ③
        fixture.registerAnnotatedCommandHandler(handler);                       ④
    }
}
```

① We are creating a new aggregate fixture as usual

② We are using the fixture to obtain an instance of the Axon `Repository`

③ We instantiate the custom command handler passing in the `Repository` instance. Also note how we inject the collection of sanctioned countries into the handler using simple dependency injection. In real life, this set of sanctioned countries will likely be obtained from external configuration.

④ We finally need to register the command handler with the fixture, so that it can route commands to this handler as well.

The tests for this look fairly straightforward:

```
class CreateLCApplicationCommandHandlerTests {
    // ..

    @BeforeEach
    void setUp() {
    final Set<Country> sanctioned = Set.of(SOKOVIA);                          ①
        fixture = new AggregateTestFixture<>(LCApplication.class);

        final Repository<LCApplication> repository = fixture.getRepository();

        CreateLCApplicationCommandHandler handler =
                new CreateLCApplicationCommandHandler(repository, sanctioned);  ②
        fixture.registerAnnotatedCommandHandler(handler);
    }

    @Test
    void shouldFailIfBeneficiaryCountryIsSanctioned() {
        fixture.given()
                .when(new CreateLCApplicationCommand(randomId(), SOKOVIA))      ③
                .expectNoEvents()
                .expectException(CannotTradeWithSanctionedCountryException.class);
    }

    @Test
    void shouldCreateIfCountryIsNotSanctioned() {
        final LCApplicationId applicationId = randomId();
        fixture.given()
                .when(new CreateLCApplicationCommand(applicationId, WAKANDA))   ④
                .expectEvents(new LCApplicationCreatedEvent(applicationId));
    }
}
```

① For the purposes of the test, we mark the country `SOKOVIA` as a *sanctioned* country. In a more realistic scenario, this will likely come from some form external configuration (e.g. a lookup table or form of external configuration). However, this is appropriate for our unit test.

② We then inject this set of *sanctioned countries* into the command handler.

③ When the `LCApplication` is created for the sanctioned country, we expect no events to be emitted and furthermore, the `CannotTradeWithSanctionedCountryException` exception to be thrown.

④ Finally, when the beneficiary belongs to a non-sanctioned country, we emit the `LCApplicationCreatedEvent` to be emitted.

The implementation of the command handler is shown below:

```
import org.springframework.stereotype.Service;

@Service                                                    ①
public class CreateLCApplicationCommandHandler {
    private final Repository<LCApplication> repository;
    private final Set<Country> sanctionedCountries;

    public CreateLCApplicationCommandHandler(Repository<LCApplication> repository,
                                             Set<Country> sanctionedCountries) {
        this.repository = repository;
        this.sanctionedCountries = sanctionedCountries;
    }

    @CommandHandler
    public void handle(CreateLCApplicationCommand command) {
        // Validations can be performed here as well          ②
        repository.newInstance(()
            -> new LCApplication(command, sanctionedCountries)); ③
    }
}
```

① We mark the class as a `@Service` to mark it as a component devoid of encapsulated state and enable auto-discovery when using annotation-based configuration or classpath scanning. As such, it can be used to perform any "plumbing" activities.

② Do note that the validation for the beneficiary's country being sanctioned could have been performed on line 18 as well. Some would argue that this would be ideal because we could avoid a potentially unnecessary invocation of the Axon `Repository` method if we did that. However, we prefer encapsulating business validations within the confines of the aggregate as much as possible — so that we don't suffer from the problem of creating an anemic domain model[22].

Finally, the aggregate implementation along with the validation is shown here:

```
class LCApplication {
// ...
    public LCApplication(CreateLCApplicationCommand command, Set<Country> sanctioned)
{
        if (sanctioned.contains(command.getBeneficiaryCountry())) { ①
            throw new CannotTradeWithSanctionedCountryException();
        }
        apply(new LCApplicationCreatedEvent(command.getId()));
    }
}
```

① The validation itself is fairly straightforward. We throw a `CannotTradeWithSanctionedCountryException` when the validation fails.

With the above examples, we looked at different ways to implement the policy enforcements encapsulated within the boundaries the aggregate.

5.6. Summary

In this chapter, we used the outputs of the eventstorming session and used it as a primary aid to create a domain model for our bounded context. We looked at how to implement this using the command query responsibility segregation (CQRS) architecture pattern. We looked at persistence options and the implications of using event sourced vs state stored aggregates. Finally, we rounded off by looking at a variety of ways in which to perform business validations. We looked at all this through a set of code examples using Spring boot and the Axon framework.

With this knowledge, we should be able to implement robust, well encapsulated, event-driven domain models. In the next chapter, we will look at implementing a user interface for these domain capabilities and examine a few options such as CRUD-based vs task-based UIs.

5.7. Questions

1. Can you examine the eventstorming session artifact from the last chapter, and identify the possible aggregates that would be required?
2. In your problem domain, can you determine the right approach for persisting aggregates? What are the reasons for choosing one approach over the other?
3. Based on your current understanding, would you apply CQRS architecture pattern in your solution? And how would you justify the choice to your team ?

5.8. Further reading

Title	Author	Location
CQRS	Martin Fowler	https://martinfowler.com/bliki/CQRS.html
Bootiful CQRS and Event Sourcing with Axon Framework	SpringDeveloper and Allard Buijze	https://www.youtube.com/watch?v=7e5euKxHhTE
The Log: What every software engineer should know about real-time data's unifying abstraction	Jay Kreps	https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
Event Sourcing	Martin Fowler	https://martinfowler.com/eaaDev/EventSourcing.html
Using a DDD Approach for Validating Business Rules	Fabian Lopez	https://www.infoq.com/articles/ddd-business-rules/
Anemic Domain Model	Martin Fowler	https://www.martinfowler.com/bliki/AnemicDomainModel.html

5.9. Answers

1. Refer to section Section 5.3.4
2. Refer to section Section 5.4, note down the pros and cons of state stored and event sourced approach, and discuss the reasons for your choice with your teammates.
3. Refer to section Section 2.1.6.1 to list down the advantages of the approach versus the traditional approach. Share the reasoning with your teammates.

[16] https://ordina-jworks.github.io/domain-driven%20design/2016/02/02/A-Decade-Of-DDD-CQRS-And-Event-Sourcing.html
[17] http://axonframework.org/
[18] https://www.lagomframework.com/
[19] https://eventuate.io/
[20] https://www.eventstore.com/
[21] https://axoniq.io/product-overview/axon-server
[22] https://www.martinfowler.com/bliki/AnemicDomainModel.html

Chapter 6. Implementing the User Interface — Task-based

> To accomplish a difficult task, one must first make it easy.
>
> — Marty Rubin

The essence of Domain Driven Design(DDD) is a lot about capturing the business process and user intent a lot more closely. In the previous chapter, we designed a set of APIs without paying a lot of attention to how those APIs would get consumed by its eventual users. In this chapter, we will design the GUI for the LC application using the JavaFX[23] framework. As part of that, we will examine how this approach of designing APIs in isolation can cause an impedance mismatch between the producers and the consumers. We will examine the consequences of this *impedance mismatch* and how task-based UIs can help cope with this mismatch a lot better.

In this chapter, we will implement the UI for LC Application and wire up the integration to the backend APIs. The list of topics to be covered is as follows:

- API styles
- Implementing the UI

At the end of the chapter, you will learn how to employ DDD principles to help you build robust user experiences that are simple and intuitive. You will also learn why it may be prudent to design your backend interfaces (APIs) from the perspective of the consumer.

6.1. Technical requirements

To follow the examples in this chapter, you will need access to:

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- JavaFX SDK 16 and SceneBuilder
- Spring Boot 2.4.x
- mvvmFX 1.8 (https://sialcasa.github.io/mvvmFX/)
- JUnit 5.7.x (Included with spring boot)
- TestFX (for UI testing)
- OpenJFX Monocle (for headless UI testing)
- Project Lombok (To reduce verbosity)
- Maven 3.x

Before we dive deep into building the GUI solution, let's do a quick recap of where we left the APIs.

6.2. API Styles

If you recall from chapter 5, we created the following commands:

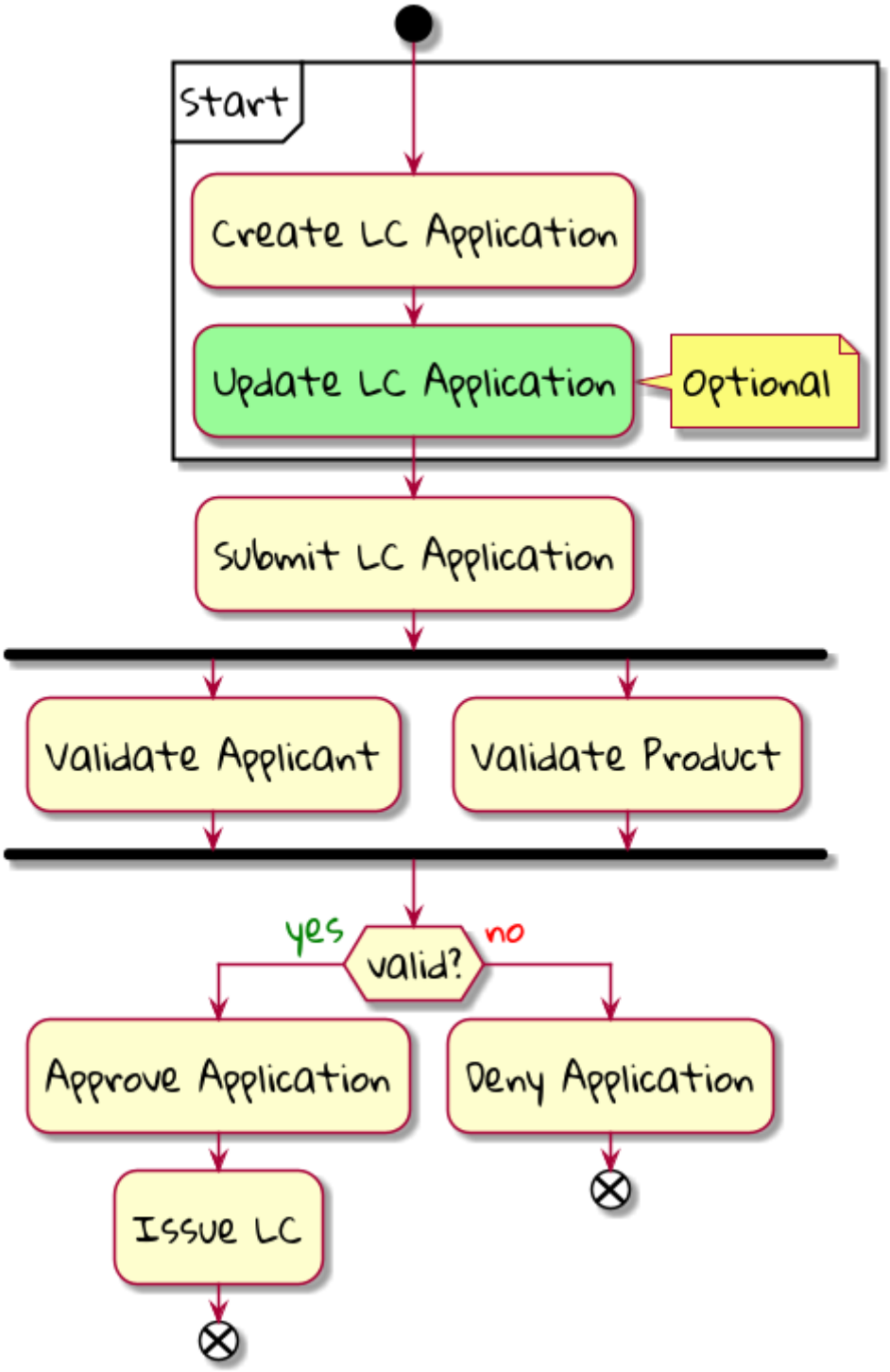


Figure 1- 67. Commands from the event storming session

If you observe carefully, there seem to be commands at two levels of granularity. The "Create LC Application" and "Update LC application" are coarse grained, whereas the others are a lot more focused in terms of their intent. One possible decomposition of the coarse grained commands can be as depicted here:

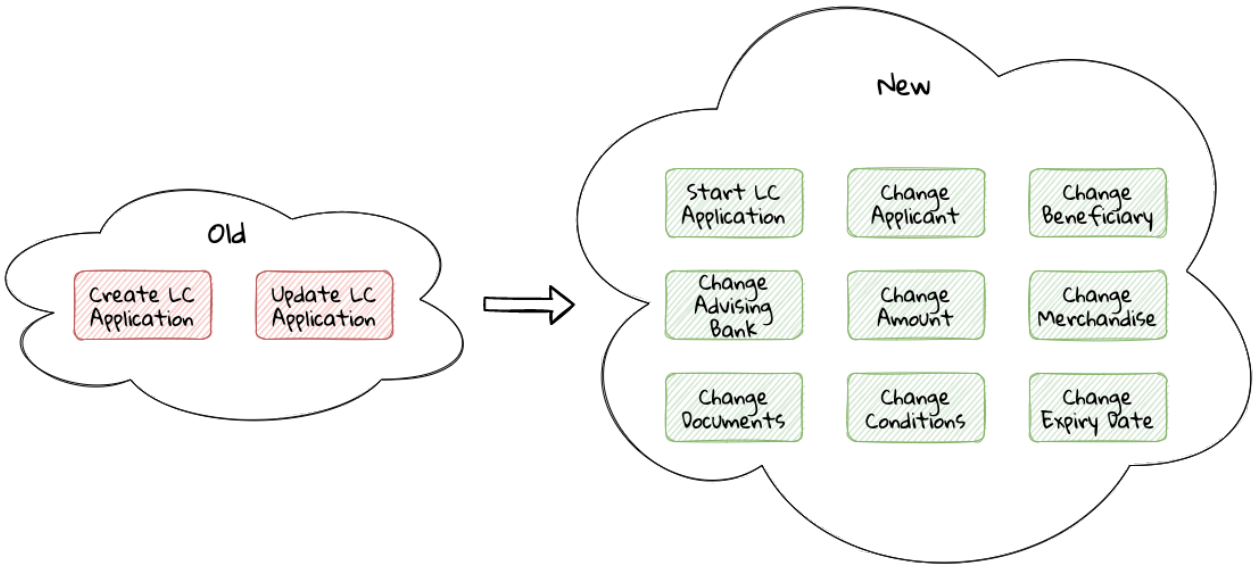


Figure 1- 68. Decomposed commands

In addition to just being more fine-grained than the commands in the previous iteration, the revised commands seem to better capture the user's intent. This may feel like a minor change in semantics, but can have a huge impact on the way our solution is used by its ultimate end-users. The question then is whether we should *always* prefer fine-grained APIs over coarse grained ones. The answer can be a lot more nuanced. When designing APIs and experiences, we see two main styles being employed:

- CRUD-based
- Task-based

Let's look at each of these in a bit more detail:

6.2.1. CRUD-based APIs

CRUD is an acronym used to refer to the four basic operations that can be performed on database applications: Create, Read, Update, and Delete. Many programming languages and protocols have their own equivalent of CRUD, often with slight variations in naming and intent. For example, SQL — a popular language for interacting with databases — calls the four functions Insert, Select, Update, and Delete. Similarly, the HTTP protocol has `POST`, `GET`, `PUT` and `DELETE` as verbs to represent these CRUD operations. This approach has got extended to our design of APIs as well. This has resulted in the proliferation of both CRUD-based APIs and user experiences. Take a look at the `CreateLCApplicationCommand` from Chapter 5:

```
import lombok.Data;

@Data
public class CreateLCApplicationCommand {

    private LCApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

Along similar lines, it would not be uncommon to create a corresponding `UpdateLCApplicationCommand` as depicted here:

```
import lombok.Data;

@Data
public class UpdateLCApplicationCommand {

    @TargetAggregateIdentifier
    private LCApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

While this is very common and also very easy to grasp, it is not without problems. Here are some questions that taking this approach raises:

1. Are we allowed to change everything listed in the `update` command?
2. Assuming that everything can change, do they all change at the same time?
3. How do we know what exactly changed? Should we be doing a diff?
4. What if all the attributes mentioned above are not included in the `update` command?
5. What if we need to add attributes in future?
6. Is the business intent of what the user wanted to accomplish captured?

In a simple system, the answer to these questions may not matter that much. However, as system complexity increases, will this approach remain resilient to change? We feel that it merits taking a look at another approach called task-based APIs to be able to answer these questions.

6.2.2. Task-based APIs

In a typical organization, individuals perform tasks relevant to their specialization. The bigger the organization, the higher the degree of specialization. This approach of segregating tasks according to one's specialization makes sense, because it mitigates the possibility of stepping on each others' shoes, especially when getting complex pieces of work done. For example, in the LC application process, there is a need to establish the value/legality of the product while also determining the credit worthiness of the applicant. It makes sense that each of these tasks are usually performed by individuals in unrelated departments. It also follows that these tasks can be performed independently from the other.

In terms of a business process, if we have a single `CreateLCApplicationCommand` that precedes these operations, individuals in both departments firstly have to wait for the entire application to be filled out before either can commence their work. Secondly, if either piece of information is updated through a single `UpdateLCApplicationCommand`, it is unclear what changed. This can result in a spurious notification being sent to at least one department because of this lack of clarity in the process.

Since most work happens in the form of specific tasks, it can work to our advantage if our processes and by extension, our APIs mirror these behaviors.

Keeping this in mind, let's re-examine our revised APIs for the LC application process:

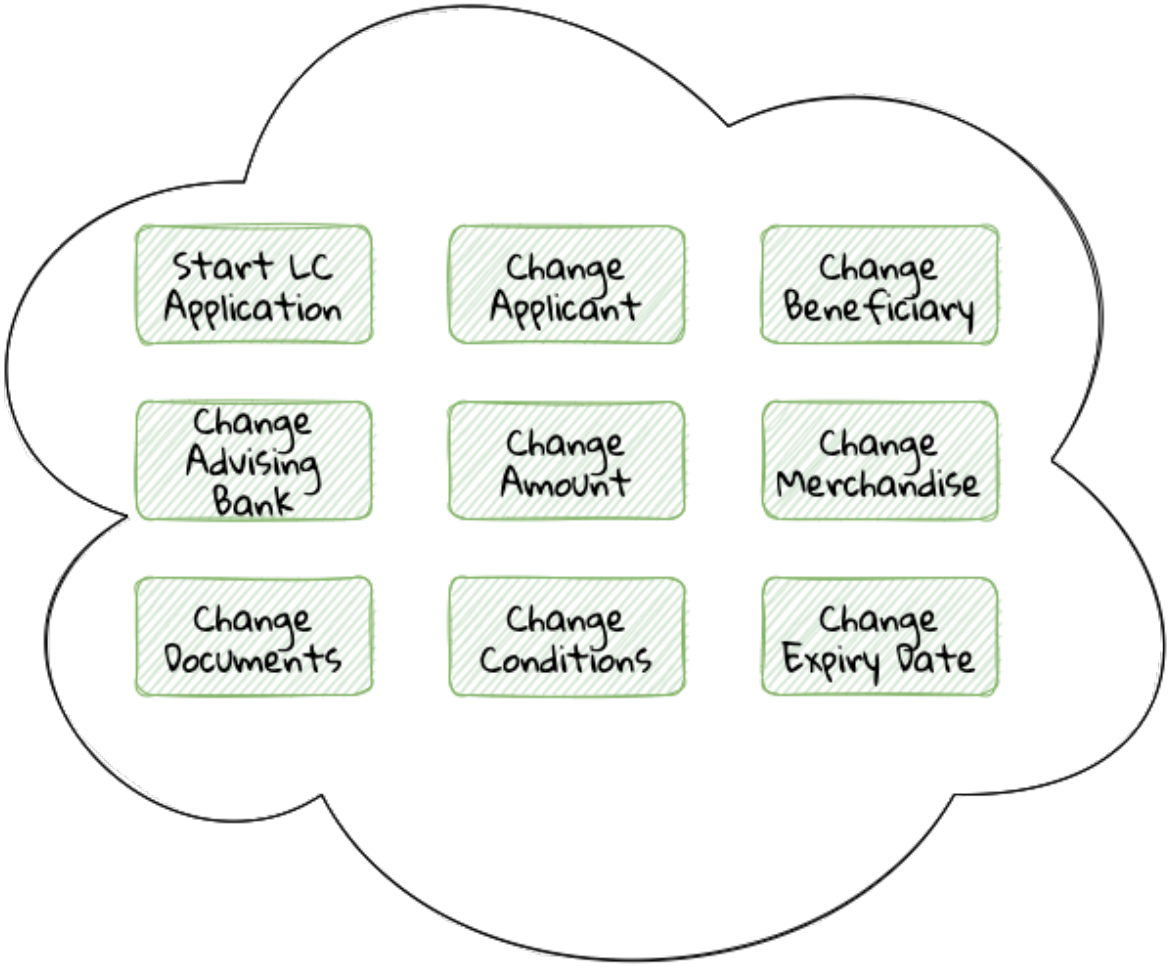


Figure 1- 69. Revised commands

While it may have appeared previously that we have simply converted our coarse-grained APIs to become more fine-grained, this in reality is a better representation of the tasks that the user intended to perform. So, in essence, task-based APIs are the decomposition of work in a manner that aligns more closely to the users' intents. With our new APIs, product validation can commence as soon as `ChangeMerchandise` happens. Also, it is unambiguously clear what the user did and what needs to happen in reaction to the user's action. It then begs the question on whether we should employ task-based APIs all the time? Let's look at the implications in more detail.

6.2.3. Task-based or CRUD-based?

CRUD-based APIs seem to operate at the level of the aggregate. In our example, we have the LC aggregate. In the simplest case, this essentially translates to four operations aligned with each of the CRUD verbs. However, as we are seeing, even in our simplified version, the LC is becoming a fairly complex concept. Having to work with just four operations at the level of the LC is cognitively complex. With more requirements, this complexity will only continue to increase. For example, consider a situation where the business expresses a need to capture a lot more information about the `merchandise`, where today, this is simply captured in the form of free-form text. A more elaborate version of merchandise information is shown here:

```
public class Merchandise {
    private MerchandiseId id;
    private Set<Item> items;
    private Packaging packaging;
    private boolean hazardous;
}

class Item {
    private ProductId productId;
    private int quantity;
    // ...
}

class Packaging {
    // ...
}
```

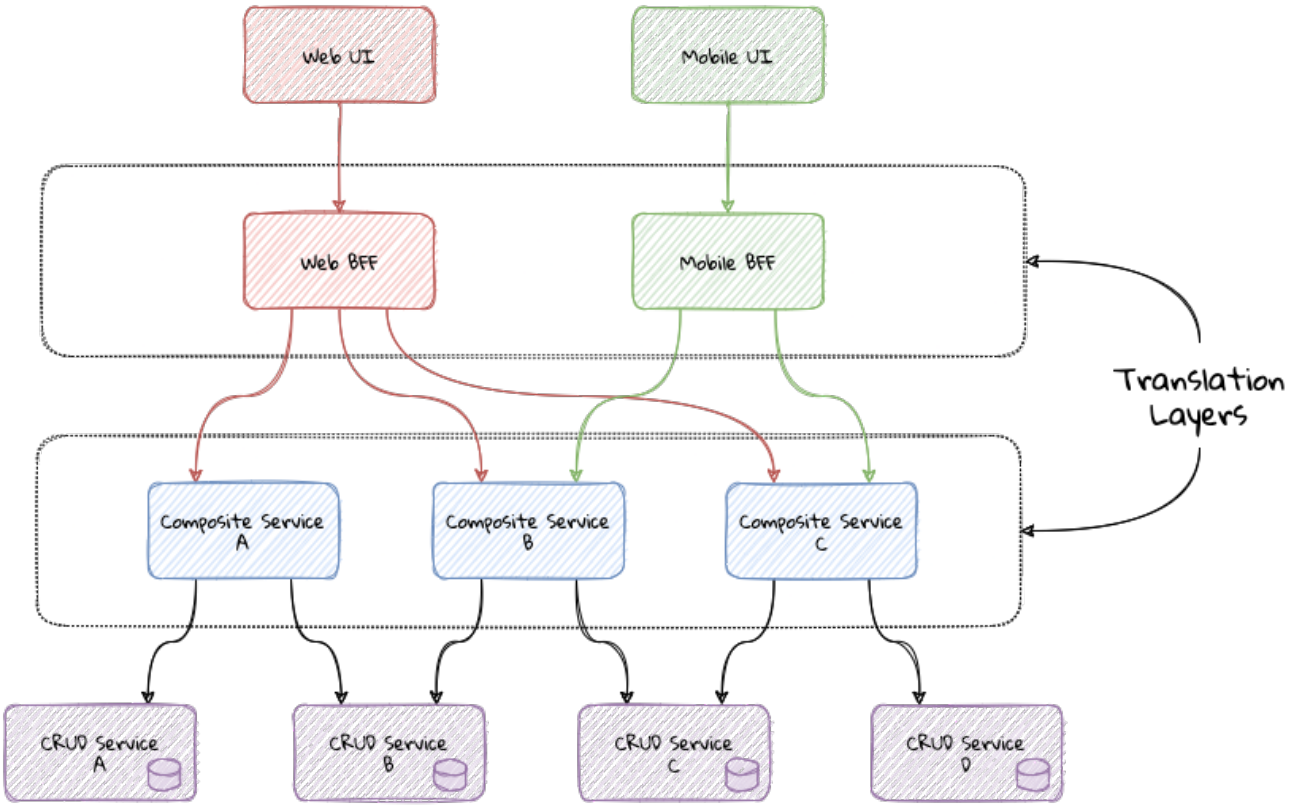
In our current design, the implications of this change are far reaching for both the provider and the consumer(s). Let's look at some of the consequences in more detail:

Characteristic	CRUD-based	Task-based	Commentary
Usability	👎	👍	Task-based interfaces tend to provide more fine-grained controls that capture user intent a lot more explicitly, making them naturally more usable — especially in cases where the domain is complex.
Reusability	👎	👍	Task-based interfaces enable more complex features to be composed using simpler ones providing more flexibility to the consumers.
Scalability	👎	👍	Task-based interfaces have an advantage because they can provide the ability to independently scale specific features. However, if the fine-grained task-based interfaces are used almost all the time in unison, it may be required to re-examine whether the users' intents are accurately captured.
Security	👎	👍	For task-based interfaces, security is enhanced from the producer's perspective by enabling application of the *principle of least privilege*[24].
Complexity	👎	👍	Complexity of the system as a whole is proportional to the number of features that need to be implemented. Assuming accidental complexity is avoided in both cases, task-based interfaces allow spreading complexity more or less uniformly across multiple simpler interfaces.
Latency	👍	👎	Arguably, coarse-grained CRUD interfaces can enable consumers to achieve a lot more in less interactions, thereby providing low latency.

Characteristic	CRUD-based	Task-based	Commentary
Management Overhead	👍	👎	For the provider, fine-grained interfaces require a lot more work managing a larger number of interfaces.

As we can see, the decision between CRUD-based and task-based interfaces is nuanced. We are not suggesting that you should choose one over the other. Which style you use will depend on your specific requirements and context. In our experience, task-based interfaces treat user intents as first class citizens and perpetrate the spirit of DDD's ubiquitous language very elegantly. Our preference is to design interfaces as task-based where possible, because they result in more intuitive interfaces that better express the problem domain.

As systems evolve, and the support richer user experiences and multiple channels, CRUD-based seem to require additional translation layers to cater to user experience needs. The visual here depicts a typical layered architecture of a solution that supports multiple user experience channels:



This set up is usually composed of:

1. Domain tier comprised of CRUD-based services that simply map closely to database entities.
2. Composite tier comprised of business capabilities that span more than one core service.
3. Backend-for-frontend (BFF[25]) tier comprised of channel-specific APIs.

Do note that the composite and BFF tiers exist primarily as a means to map backend capabilities to user intent. In an ideal world, where backend APIs reflect user intent closely, the need for translations should be minimal (if at all). Our experience suggests that such a setup causes business logic to get pushed closer to the user channels as opposed to being encapsulated within the confines

of well-factored business services. In addition, these tiers cause inconsistent experiences across channels for the same functionality, given that modern teams are structured along tier boundaries.

We are not opposed to the use of layered architectures. We recognize that a layered architecture can bring modularity, separation of concerns and other related benefits. However, we are opposed to creating additional tiers merely as a means to compensate for poorly factored core domain APIs.

A well factored API tier can have a profound effect on how great user experiences are built. However, this is a chapter on implementing the user interface. Let's revert to creating the user interface for the LC application.

6.3. Bootstrapping the UI

We will be building the UI for the LC issuance application we created in Chapter 5: Implementing Domain Logic. For detailed instructions, refer to the section on *Bootstrapping the application*. In addition, we will need to add the following dependencies to the `dependencies` section of the Maven `pom.xml` file in the root directory of the project:

```
<dependencies>
    <!--...-->
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-graphics</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx</artifactId>
        <version>${mvvmfx.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx-spring-boot</artifactId>
        <version>${mvvmfx.version}</version>
    </dependency>
    <!--...-->
</dependencies>
```

To run UI tests, you will need to add the following dependencies:

```
<dependencies>
    <!--...-->
    <dependency>
        <groupId>org.testfx</groupId>
        <artifactId>testfx-junit5</artifactId>
        <scope>test</scope>
        <version>${testfx-junit5.version}</version>
    </dependency>
    <dependency>
        <groupId>org.testfx</groupId>
        <artifactId>openjfx-monocle</artifactId>
        <version>${openjfx-monocle.version}</version>
    </dependency>
    <dependency>
        <groupId>de.saxsys</groupId>
        <artifactId>mvvmfx-testing-utils</artifactId>
        <version>${mvvmfx.version}</version>
        <scope>test</scope>
    </dependency>
    <!--...-->
</dependencies>
```

To be able to run the application from the command line, you will need to add the `javafx-maven-plugin` to the `plugins` section of your `pom.xml`, per the following:

```
<plugin>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-maven-plugin</artifactId>
    <version>${javafx-maven-plugin.version}</version>
    <configuration>
        <mainClass>com.premonition.lc.ch06.App</mainClass>
    </configuration>
</plugin>
```

To run the application from the command line, use:

```
mvn javafx:run
```

If you are using a JDK greater that version 1.8, the JavaFX libraries may not be bundled with the JDK itself. When running the application from your IDE, you will likely need to add the following:

```
--module-path=<path-to-javafx-sdk>/lib/ \
   --add-modules=javafx.controls,javafx.graphics,javafx.fxml,javafx.media
```

We are making use of the mvvmFX framework to assemble the UI. To make this work with spring boot, the application launcher looks as depicted here:

```
@SpringBootApplication
public class App extends MvvmfxSpringApplication { ①

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void startMvvmfx(Stage stage) {
        stage.setTitle("LC Issuance");

        final Parent parent = FluentViewLoader
                .fxmlView(MainView.class)
                .load().getView();

        final Scene scene = new Scene(parent);
        stage.setScene(scene);
        stage.show();
    }
}
```

① Note that we are required to extend from the mvvmFX framework class `MvvmfxSpringApplication`.

Please refer to the ch06 directory of the accompanying source code repository for the complete example.

6.4. Implementing the UI

When working with user interfaces, it is fairly customary to use one of these presentation patterns:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

The MVC pattern has been around for the longest time. The idea of separating concerns among collaborating model, view and controller objects is a sound one. However, beyond the definition of these objects, actual implementations seem to vary wildly — with the controller becoming overly complex in a lot of cases. In contrast, MVP and MVVM, while being derivatives of MVC, seem to bring out better separation of concerns between the collaborating objects. MVVM, in particular

when coupled with data binding constructs, make for code that is much more readable, maintainable and testable. In this book, we make use of MVVM because it enables test-driven development which is a strong personal preference for us. Let's look at a quick MVVM primer as implemented in the `mvvmfx` framework.

6.4.1. Model View View-Model (MVVM) primer

Modern UI frameworks started adopting a declarative style to express the view. MVVM was designed to remove all GUI code (code-behind) from the view by making use of binding expressions. This allowed for a cleaner separation of stylistic vs. programming concerns. A high level visual of how this pattern is implemented is shown here:

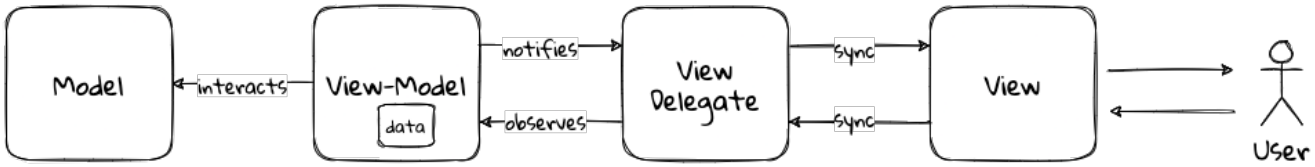


Figure 1- 70. MVVM design pattern

The pattern comprises the following components:

- **Model**: responsible to house the business logic and managing the state of the application.
- **View**: responsible for presenting data to the user and notifying the view-model about user interactions through the view delegate.
- **View Delegate**: responsible for keeping the view and the view model in sync as changes are made by the user or on the view model. It is also responsible for transmitting actions performed on the view to the view model.
- **View-Model**: responsible for handling user interactions on behalf of the view. The view-model interacts with the view using the observer pattern (typically one-way or two-way data binding to make it more convenient). The view-model interacts with the model for updates and read operations.

6.4.2. Creating a new LC

Let's consider the example of creating a new LC. To start creation of a new LC, all we need is for the applicant to provide a friendly client reference. This is an easy to remember string of free text. A simple rendition of this UI is shown here:

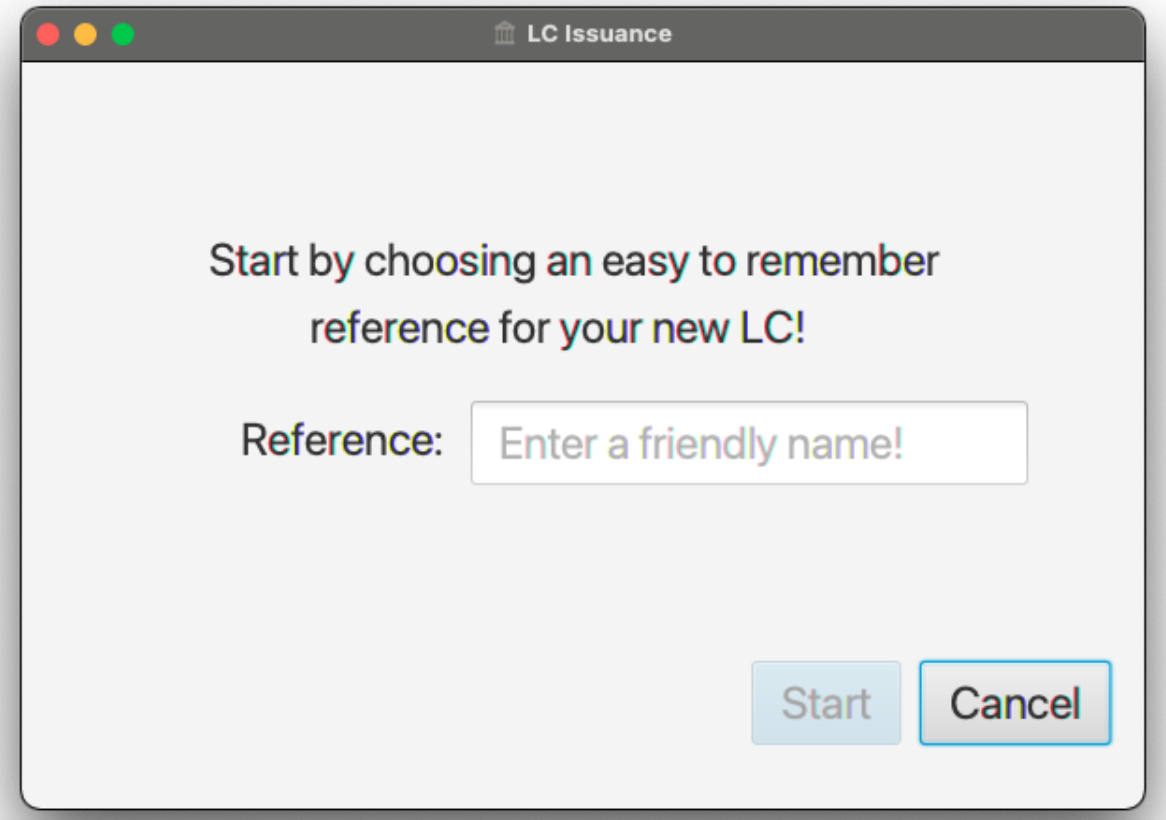


Figure 1- 71. Start LC creation screen

Let's examine the implementation and purpose of each component in more detail.

Declarative view

When working with JavaFX, the view can be rendered using a declarative style in FXML format. Important excerpts from the `StartLCView.fxml` file to start creating a new LC are shown here:

```
<?import javafx.scene.layout.Pane?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.TextField?>

<Pane id="start-lc"  xmlns="http://javafx.com/javafx/16"
                      xmlns:fx="http://javafx.com/fxml/1"
      fx:controller="com.premonition.lc.ch06.ui.views.StartLCView">      ①
    ...

    <TextField id="client-reference"
               fx:id="clientReference"/>                                  ②

    <Button id="start-button"
            fx:id="startButton"
            text="Start"
            onAction="#start"/>                                         ③
    ...
</Pane>
```

① The `StartLCView` class acts as the view delegate for the FXML view and is assigned using the `fx:controller` attribute of the root element (`javafx.scene.layout.Pane` in this case).

② In order to reference `client-reference` input field in the view delegate, we use the `fx:id` annotation — `clientReference` in this case.

③ Similarly, the `start-button` is referenced using `fx:id="startButton"` in the view delegate. Furthermore, the `start` method in the view delegate is assigned to handle the default action (the button press event for `javafx.scene.control.Button`).

View delegate

Next, let's look at the structure of the view delegate `com.premonition.lc.issuance.ui.views.StartLCView`:

```
import javafx.fxml.FXML;
//...
public class StartLCView {                      ①

    @FXML
    private TextField clientReference;          ②
    @FXML
    private Button startButton;                 ③

    public void start(ActionEvent event) {      ④
        // Handle button press logic here
    }

    // Other parts omitted for brevity...
}
```

① The view delegate class for the `StartLCView.fxml` view.

② The Java binding for the `clientReference` textbox in the view. The name of the member needs to match exactly with the value of the `fx:id` attribute in the view. Further, it needs to be annotated with the `@javafx.fxml.FXML` annotation. The use of the `@FXML` annotation is optional if the member in the view delegate is `public` and matches the name in the view.

③ Similarly, the `startButton` is bound to the corresponding button widget in the view.

④ The method for the action handler when the `startButton` is pressed.

View-Model

The view-model class `StartLCViewModel` for the `StartLCView` is shown here:

```
import javafx.beans.property.StringProperty;
import de.saxsys.mvvmfx.ViewModel;

public class StartLCViewModel implements ViewModel {          ①

    private final StringProperty clientReference;              ②

    public StartLCViewModel() {
        this.clientReference = new SimpleStringProperty();     ③
    }

    public StringProperty clientReferenceProperty() {          ④
        return clientReference;
    }

    public String getClientReference() {
        return clientReference.get();
    }

    public void setClientReference(String clientReference) {
        this.clientReference.set(clientReference);
    }

    // Other getters and setters omitted for brevity
}
```

① The view-model class for the `StartLCView`. Note that we are required to implement the `de.saxsys.mvvmfx.ViewModel` interface provided by the mvvmFX framework.

② We are initializing the `clientReference` property using the `SimpleStringProperty` provided by JavaFX. There are several other property classes to define more complex types. Please refer to the JavaFX documentation for more details.

③ The value of the `clientReference` in the view-model. We will look at how to associate this with value of the `clientReference` textbox in the view shortly. Note that we are using the `StringProperty` provided by `JavaFX`, which provides access to the underlying `String` value of the client reference.

④ JavaFX beans are required to create a special accessor for the property itself in addition to the standard getter and setter for the underlying value.

Binding the view to the view-model

Next, let's look at how to associate the view to the view-model:

```
import de.saxsys.mvvmfx.Initialize;
import de.saxsys.mvvmfx.FxmlView;
import de.saxsys.mvvmfx.InjectViewModel;
//...
public class StartLCView implements FxmlView<StartLCViewModel> {     ①

    @FXML
    private TextField clientReference;
    @FXML
    private Button startButton;

    @InjectViewModel
    private StartLCViewModel viewModel;                              ②

    @Initialize
    private void initialize() {                                      ③
        clientReference.textProperty()
            .bindBidirectional(viewModel.clientReferenceProperty()); ④
        startButton.disableProperty()
            .bind(viewModel.startDisabledProperty());                ⑤
    }

    // Other parts omitted for brevity...
}
```

① The mvvmFX framework requires that the view delegate implement the `FXMLView<? extends ViewModelType>`. In this case, the view-model type is `StartLCViewModel`. The mvvmFX framework supports other view types as well. Please refer to the framework documentation for more details.

② The framework provides a `@de.saxsys.mvvmfx.InjectViewModel` annotation to allow dependency injecting the view-model into the view delegate.

③ The framework will invoke all methods annotated with the `@de.saxsys.mvvmfx.Initialize` annotation during the initialization process. The annotation can be omitted if the method is named `initialize` and is declared `public`. Please refer to the framework documentation for more details.

④ We have now bound the text property of the `clientReference` textbox in the view delegate to the corresponding property in the view-model. Note that this is a **bidirectional** binding, which means that the value in the view and the view model are kept in sync if it changes on either side.

⑤ This is another variation of binding in action, where we are making use of a unidirectional binding. Here, we are binding the disabled property of the `start` button to the corresponding

property on the view-model. We will look at why we need to do this shortly.

Enforcing business validations in the UI

We have a business validation that the client reference for an LC needs to be at least 4 characters in length. This will be enforced on the back-end. However, to provide a richer user experience, we will also enforce this validation on the UI.

This may feel contrary to the notion of centralizing business validations on the back-end. While this may be a noble attempt at implementing the DRY (Don't Repeat Yourself) principle, in reality, it poses a lot of practical problems. Distributed systems expert — Udi Dahan has a very interesting take on why this may not be such a virtuous thing to pursue[26]. Ted Neward also talks about this in his blog titled *The Fallacies of Enterprise Computing*[27].

The advantage of using MVVM is that this logic is easily testable in a simple unit test of the view-model. Let's see this in action test-drive this now:

```
class StartLCViewModelTests {

    private StartLCViewModel viewModel;

    @BeforeEach
    void before() {
        int clientReferenceMinLength = 4;
        viewModel = new StartLCViewModel(clientReferenceMinLength);
    }

    @Test
    void shouldNotEnableStartByDefault() {
        assertThat(viewModel.getStartDisabled()).isTrue();
    }

    @Test
    void shouldNotEnableStartIfClientReferenceLesserThanMinimumLength() {
        viewModel.setClientReference("123");
        assertThat(viewModel.getStartDisabled()).isTrue();
    }

    @Test
    void shouldEnableStartIfClientReferenceEqualToMinimumLength() {
        viewModel.setClientReference("1234");
        assertThat(viewModel.getStartDisabled()).isFalse();
    }

    @Test
    void shouldEnableStartIfClientReferenceGreaterThanMinimumLength() {
        viewModel.setClientReference("12345");
        assertThat(viewModel.getStartDisabled()).isFalse();
    }
}
```

Now, let's look at the implementation for this functionality in the view-model:

```
public class StartLCViewModel implements ViewModel {

    //...
    private final StringProperty clientReference;
    private final BooleanProperty startDisabled;                    ①

    public StartLCViewModel(int clientReferenceMinLength) {         ②
        this.clientReference = new SimpleStringProperty();
        this.startDisabled = new SimpleBooleanProperty();
        this.startDisabled
            .bind(this.clientReference.length()
                    .lessThan(clientReferenceMinLength));           ③
    }

    //...
}

public class StartLCView implements FxmlView<StartLCViewModel> {

    //...
    @Initialize
    public void initialize() {
        startButton.disableProperty()
            .bind(viewModel.startDisabledProperty());               ④
        clientReference.textProperty()
            .bindBidirectional(viewModel.clientReferenceProperty());
    }
    //...
}
```

① We declare a `startDisabled` property in the view-model to manage when the start button should be disabled.

② The minimum length for a valid client reference is injected into the view-model. It is conceivable that this value will be provided as part of external configuration, or possibly from the back-end.

③ We create a binding expression to match the business requirement.

④ We bind the view-model property to the disabled property of the start button in the view delegate.

Let's also look at how to write an end-to-end, headless UI test as shown here:

```
@UITest
public class StartLCViewTests {                                    ①

    @Autowired
    private ApplicationContext context;

    @Init
    public void init() {
        MvvmFX.setCustomDependencyInjector(context::getBean);      ②
    }

    @Start
    public void start(Stage stage) {                               ③
        final Parent parent = FluentViewLoader
                .fxmlView(StartLCView.class)
                .load().getView();
        stage.setScene(new Scene(parent));
        stage.show();
    }

    @Test
    void blankClientReference(FxRobot robot) {
        robot.lookup("#client-reference")                          ④
            .queryAs(TextField.class)
            .setText("");

        verifyThat("#start-button", NodeMatchers.isDisabled());    ⑤
    }

    @Test
    void validClientReference(FxRobot robot) {
        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText("Test");

        verifyThat("#start-button", NodeMatchers.isEnabled());     ⑤
    }
}
```

① We have written a convenience `@UITest` extension to combine spring framework and TestFX testing. Please refer to the accompanying source code with the book for more details.

② We set up the spring context to act as the dependency injection provider for the mvvmFX framework and its injection annotations (for example, `@InjectViewModel`) to work.

③ We are using the `@Start` annotation provided by the TestFX framework to launch the UI.

④ The TestFX framework injects an instance of the `FxRobot` UI helper, which we can use to access UI elements.

⑤ We are using the The TestFX framework provided convenience matchers for test assertions.

Now, when we run the application, we can see that the start button is enabled when a valid client reference is entered:

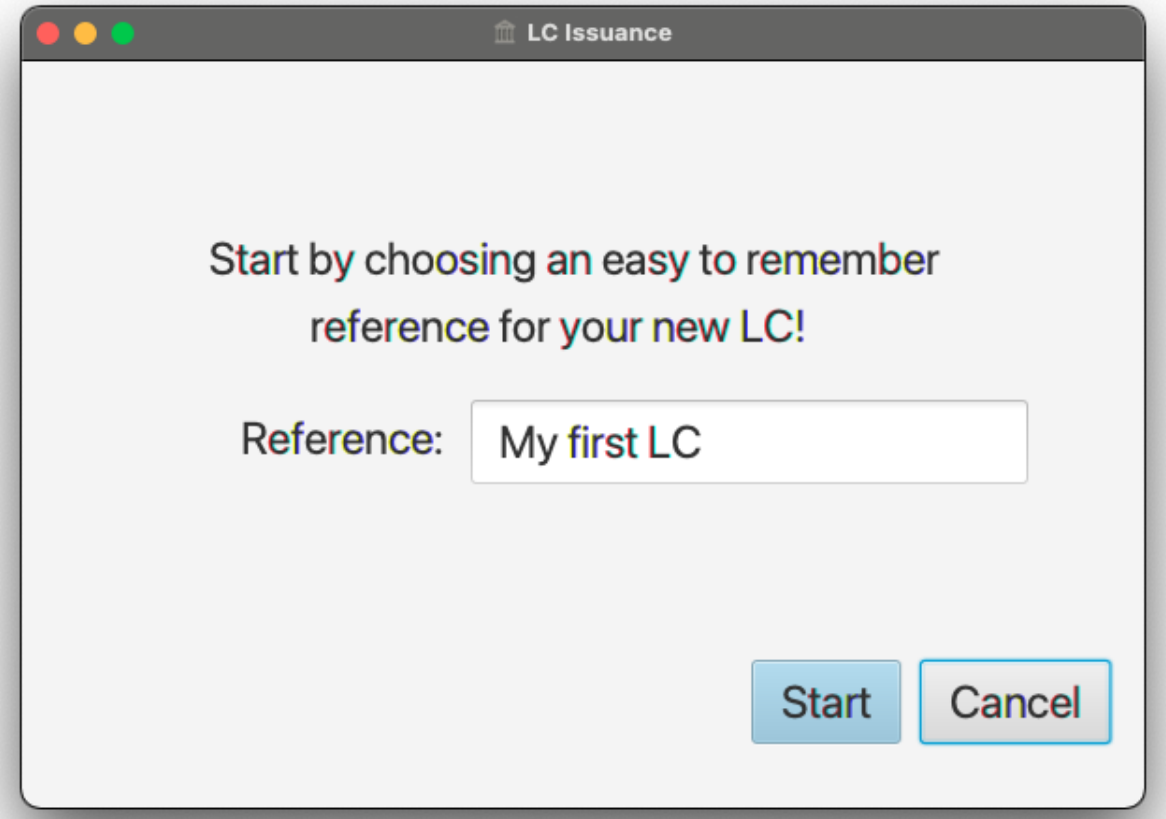


Figure 1- 72. The start button is enabled with a valid client reference

Now that we have the start button enabling correctly, let's implement the actual creation of the LC itself by invoking the backend API.

Integrating with the backend

LC creation is a complex process, requiring information about a variety of items as evidenced in figure Figure 1- 69 when we decomposed the LC creation process. In this section, we will integrate the UI with the command to start creation of a new LC. This happens when we press the *Start* button on the Figure 1- 71. The revised `StartNewLCApplicationCommand` looks as shown here:

```
@Data
public class StartNewLCApplicationCommand {
    private final String applicantId;
    private final LCApplicationId id;
    private final String clientReference;

    private StartNewLCApplicationCommand(String applicantId, String clientReference) {
        this.id = LCApplicationId.randomId();
        this.applicantId = applicantId;
        this.clientReference = clientReference;
    }

    public static StartNewLCApplicationCommand startApplication( ①
                    String applicantId,
                    String clientReference) {
        return new StartNewLCApplicationCommand(applicantId, clientReference);
    }
}
```

① To start a new LC application, we need an `applicantId` and a `clientReference`.

Given that we are using the MVVM pattern, the code to invoke the backend service is part of the view-model. Let's test-drive this functionality:

```
@ExtendWith(MockitoExtension.class)
class StartLCViewModelTests {

    @Mock
    private BackendService service;

    @BeforeEach
    void before() {
        int clientReferenceMinLength = 4;
        viewModel = new StartLCViewModel(clientReferenceMinLength, service);
    }

    @Test
    void shouldNotInvokeBackendIfStartButtonIsDisabled() {
        viewModel.setClientReference("");
        viewModel.startNewLC();

        Mockito.verifyNoInteractions(service);
    }
}
```

The view-model is enhanced accordingly to inject an instance of the `BackendService` and looks as shown here:

```
public class StartLCViewModel implements ViewModel {

    private final BackendService service;
    // Other members omitted for brevity

    public StartLCViewModel(int clientReferenceMinLength,
                            BackendService service) {
        this.service = service;
        // Other code omitted for brevity
    }

    public void startNewLC() {
        // TODO: invoke backend!
    }
}
```

Now a test to actually make sure that the backend gets invoked only when a valid client reference is input:

```
class StartLCViewModelTests {
    // ...

    @BeforeEach
    void before() {
        viewModel = new StartLCViewModel(4, service);
        viewModel.setLoggedInUser(new LoggedInUserScope("test-applicant"));   ①
    }

    @Test
    void shouldNotInvokeBackendIfStartButtonIsDisabled() {
        viewModel.setClientReference("");
        viewModel.startNewLC();

        Mockito.verifyNoInteractions(service);                                ②
    }

    @Test
    void shouldInvokeBackendWhenStartingCreationOfNewLC() {
        viewModel.setClientReference("My first LC");
        viewModel.startNewLC();

        Mockito.verify(service).startNewLC("test-applicant", "My first LC");  ③
    }
}
```

① We set the logged in user

② When the client reference is blank, there should be no interactions with the backend service.

③ When a valid value for the client reference is entered, the backend should be invoked with the

entered value.

The implementation to make this test pass, then looks like this:

```
public class StartLCViewModel {
    //...
    public void startNewLC() {
        if (!getStartDisabled()) {                  ①
            service.startNewLC(
                    userScope.getLoggedInUserId(),
                    getClientReference());          ②
        }
    }
    //...
}
```

① We check that the start button is enabled before invoking the backend.

② The actual backend call with the appropriate values.

Now let's look at how to integrate the backend call from the view. We test this in a UI test as shown here:

```
@UITest
public class StartLCViewTests {

    @MockBean
    private BackendService service;                                    ①

    //...

    @Test
    void shouldLaunchLCDetailsWhenCreationIsSuccessful(FxRobot robot) {
        final String clientReference = "My first LC";
        LCApplicationId lcApplicationId = LCApplicationId.randomId();

        when(service.startNewLC("test-applicant", clientReference))
                .thenReturn(lcApplicationId);                          ②

        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText(clientReference);                                 ③
        robot.clickOn("#start-button");                                ④

        Mockito.verify(service).startNewLC(
                "test-applicant", clientReference);                    ⑤

        verifyThat("#lc-details-screen", isVisible());                 ⑥
    }
}
```

① We inject a mock instance of the backend service.

② We stub the call to the backend to return successfully.

③ We type in a valid value for the client reference.

④ We click on the `start` button.

⑤ We verify that the service was indeed invoked with the correct arguments.

⑥ We verify that we have moved to the next screen in the UI (the LC details screen).

Let's also look at what happens when the service invocation fails in another test:

```
public class StartLCViewTests {
    //...
    @Test
    void shouldStayOnCreateLCScreenOnCreationFailure(FxRobot robot) {
        final String clientReference = "My first LC";
        when(service.startNewLC("test-applicant", clientReference))
            .thenThrow(new RuntimeException("Failed!!"));   ①

        robot.lookup("#client-reference")
            .queryAs(TextField.class)
            .setText(clientReference);
        robot.clickOn("#start-button");

        verifyThat("#start-lc-screen", isVisible());        ②
    }
}
```

① We stub the backend service call to fail with an exception.

② We verify that we continue to remain on the `start-lc-screen`.

The view implementation for this functionality is shown here:

```
import javafx.concurrent.Service;

public class StartLCView {
    //...
    public void start(ActionEvent event) {
        new Service<Void>() {                    ①
            @Override
            private Task<Void> createTask() {
                return new Task<>() {
                    @Override
                    private Void call() {
                        viewModel.startNewLC();  ②
                        return null;
                    }
                };
            }

            @Override
            private void succeeded() {
                Stage stage = UIUtils.getStage(event);
                showLCDetailsView(stage);        ③
            }

            @Override
            private void failed() {
                // Nothing for now. Remain on the same screen.
            }
        }.start();
    }
}
```

① JavaFX, like most frontend frameworks, is single-threaded and requires that long-running tasks not be invoked on the UI thread. For this purpose, it provides the `javafx.concurrent.Service` abstraction to handle such interactions elegantly in a background thread.

② The actual invocation of the backend through the view-model happens here.

③ We show the next screen to enter more LC details here.

Finally, the service implementation itself is shown here:

```
import org.axonframework.commandhandling.gateway.CommandGateway;

@Service
public class BackendService {

    private final CommandGateway gateway;                          ①

    public BackendService(CommandGateway gateway) {
        this.gateway = gateway;
    }

    public LCApplicationId startNewLC(String applicantId, String clientReference) {
        return gateway.sendAndWait(                                ②
                   startApplication(applicantId, clientReference)
               );
    }
}
```

① We inject the `org.axonframework.commandhandling.gateway.CommandGateway` provided by the Axon framework to invoke the command.

② The actual invocation of the backend using the `sendAndWait` method happens here. In this case, we are blocking until the backend call completes. There are other variations that do not require this kind of blocking. Please refer to the Axon framework documentation for more details.

We have now seen a complete example of how to implement the UI and invoke the backend API.

6.5. Summary

In this chapter, we looked the nuances of API styles and clarified why it is very important to design APIs that capture the users' intent closely. We looked at the differences between CRUD-based and task-based APIs. Finally, we implemented the UI making use of the MVVM design pattern and demonstrated how it aids in test-driving frontend functionality.

Now that we have implemented the creation of new LC, for implementing the subsequent commands we will require access to an existing LC. In the next chapter, we will look at how to implement the query side and how to keep it in sync with the command side.

6.6. Questions

- What kind of APIs do you come up with in your domain? CRUD-based? Task-based? Something else?
- How do consumers find your APIs? Do they have to implement further translations of your APIs to consume them meaningfully?
- Are you able to test-drive your front-end functionality? Do you see merit in this approach?

6.7. Further reading

Title	Author	Location
Task-drien user interfaces	Oleksandr Sukholeyster	https://www.uxmatters.com/mt/archives/2014/12/task-driven-user-interfaces.php
Business logic, a different perspective	Udi Dahan	https://vimeo.com/131757759
The Fallacies of Enterprise Computing	Ted Neward	http://blogs.tedneward.com/post/enterprise-computing-fallacies/
GUI architectures	Martin Fowler	https://martinfowler.com/eaaDev/uiArchs.html

[23] https://openjfx.com/

[24] https://en.wikipedia.org/wiki/Principle_of_least_privilege

[25] https://philcalcado.com/2015/09/18/the_back_end_for_front_end_pattern_bff.html

[26] https://vimeo.com/131757759

[27] http://blogs.tedneward.com/post/enterprise-computing-fallacies/

Chapter 7. Implementing Queries

> The best view comes after the hardest climb.
>
> — Anonymous

In the section on CQRS from Chapter 3 - Where and How Does DDD Fit?, we described how DDD and CQRS complement each other and how the query side (read models) can be used to create one or more representations of the underlying data. In this chapter, we will dive deeper into how we can construct read optimized representations of the data by listening to domain events. We will also look at persistence options for these read models.

When working with query models, we construct models by listening to events as they happen. We will examine how to deal with situations where:

- New requirements evolve over a period of time requiring us to build new query models.
- We discover a bug in our query model which requires us to recreate the model from scratch.

By the end of this chapter, you will learn to appreciate how to build query models by listening to domain events. You will also learn how to purpose build new query models to suit specific read requirements as opposed to being restricted by the data model that was chosen to service commands. You will finally look at how historic event replays work and how you can use it to create new query models to service new requirements.

7.1. Technical requirements

To follow the examples in this chapter, you will need access to:

- JDK 1.8+ (We have used Java 17 to compile sample sources)
- Spring Boot 2.4.x
- Axon framework 4.5.3
- JUnit 5.7.x (Included with spring boot)
- Project Lombok (To reduce verbosity)
- Maven 3.x

Please refer to the ch07 directory of the book's accompanying source code repository for complete working examples.

7.2. Continuing our design journey

In Chapter 4 - Domain analysis and modeling, we discussed eventstorming as a lightweight method to clarify business flows. As a reminder, this is the output produced from our eventstorming session:

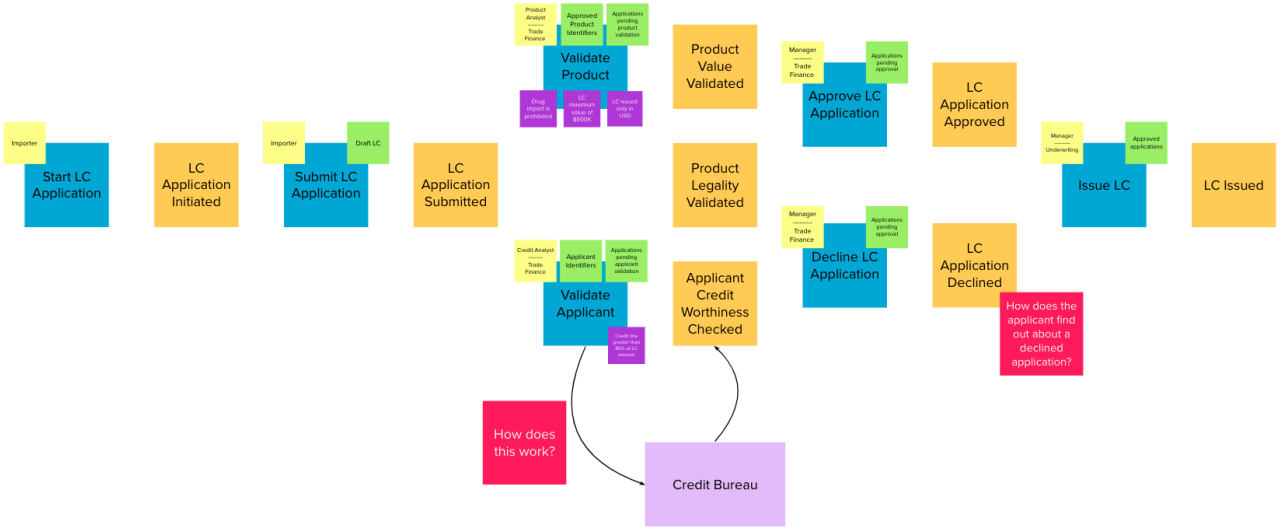


Figure 1- 73. Recap of eventstorming session

As mentioned previously, we are making use of the CQRS architecture pattern to create the solution. For a detailed explanation on why this is a sound method to employ, please refer to the "When to use CQRS" section in Chapter 3. In the diagram above, the **green** stickies represent **read/query models**. These query models are required when validating a command (for example: list of valid product identifiers when processing the `ValidateProduct` command) or if information is simply required to be presented to the user (for example: a list of LCs created by an applicant). Let's look at what it means to apply CQRS in practical terms for the query side.

7.3. Implementing the query side

In Chapter 5, we examined how to publish events when a command is successfully processed. Now, let's look at how we can construct a query model by listening to these domain events. Logically, this will look something like how it is depicted here:

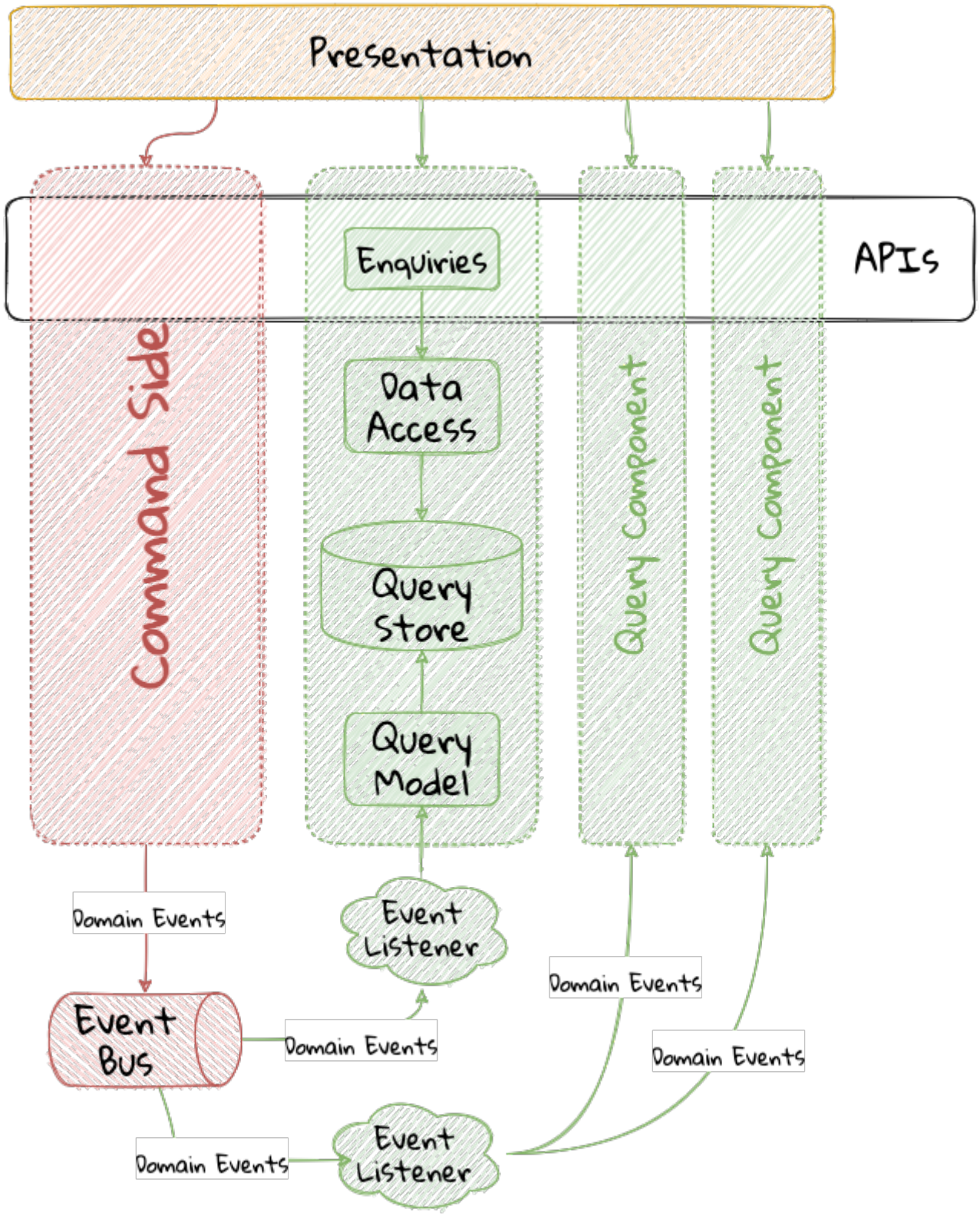


Figure 1- 74. CQRS application — query side

Please refer to the section on implementing the command side in Chapter 5 for a detailed explanation of how the command side is implemented.

The high level sequence on the query side is described here:

1. An event listening component listens to these domain events published on the event bus.
2. Constructs a purpose-built query model to satisfy a specific query use case.

3. This query model is persisted in a datastore optimized for read operations.
4. This query model is then exposed in the form of an API.

Note how there can exist more than one query side component to handle respective scenarios.

Let's implement each of these steps to see how this works for our LC issuance application.

7.3.1. Tooling choices

In a CQRS application, there is a separation between the command and query side. At this time, this separation is logical in our application because both the command and query side are running as components within the same application process. To illustrate the concepts, we will use conveniences provided by the Axon framework to implement the query side in this chapter. In Chapter 10, we will look at how it may not be necessary to use a specialized framework (like Axon) to implement the query side.

When implementing the query side, we have two concerns to solve for as depicted in the following picture :

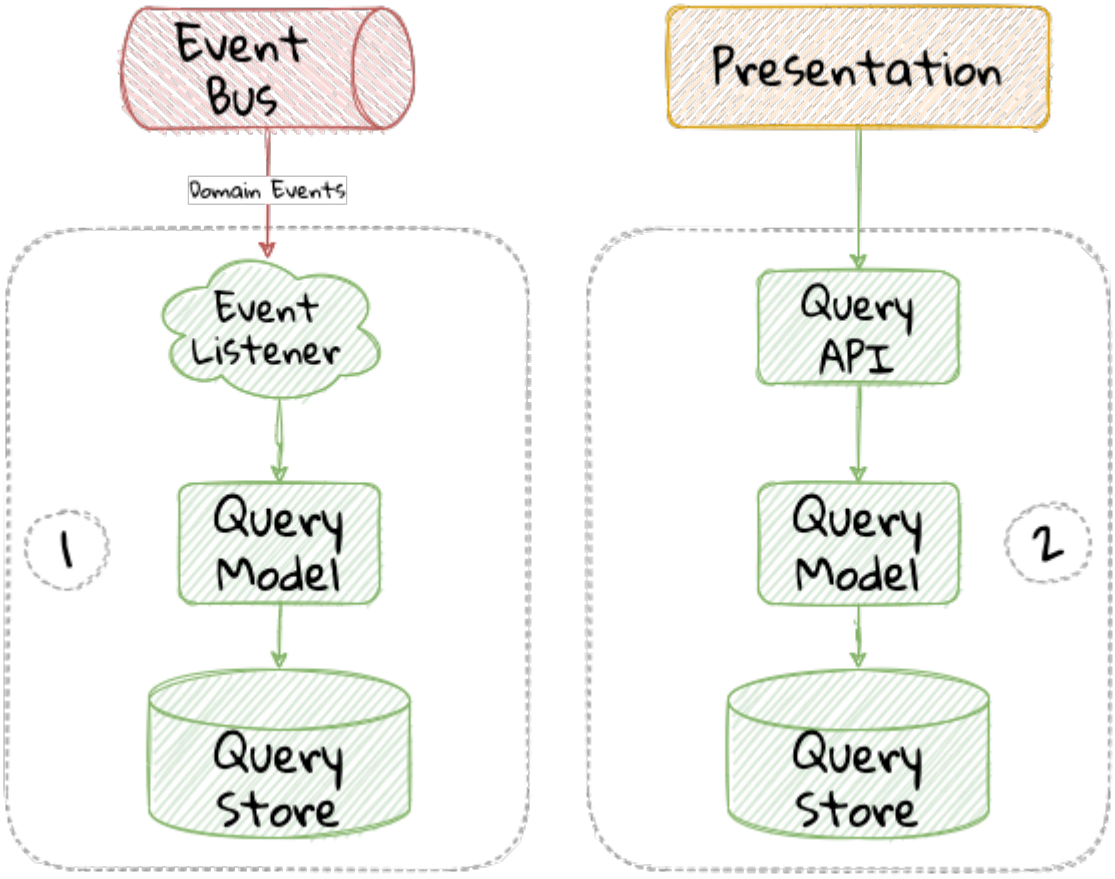


Figure 1- 75. Query side dissected

1. Consuming domain events and persisting one or more query models.
2. Exposing the query model as an API.

Before we start implementing these concerns, let's identify the queries we need to implement for

our LC issuance application.

7.3.2. Identifying queries

From the eventstorming session, we have the following queries to start with:

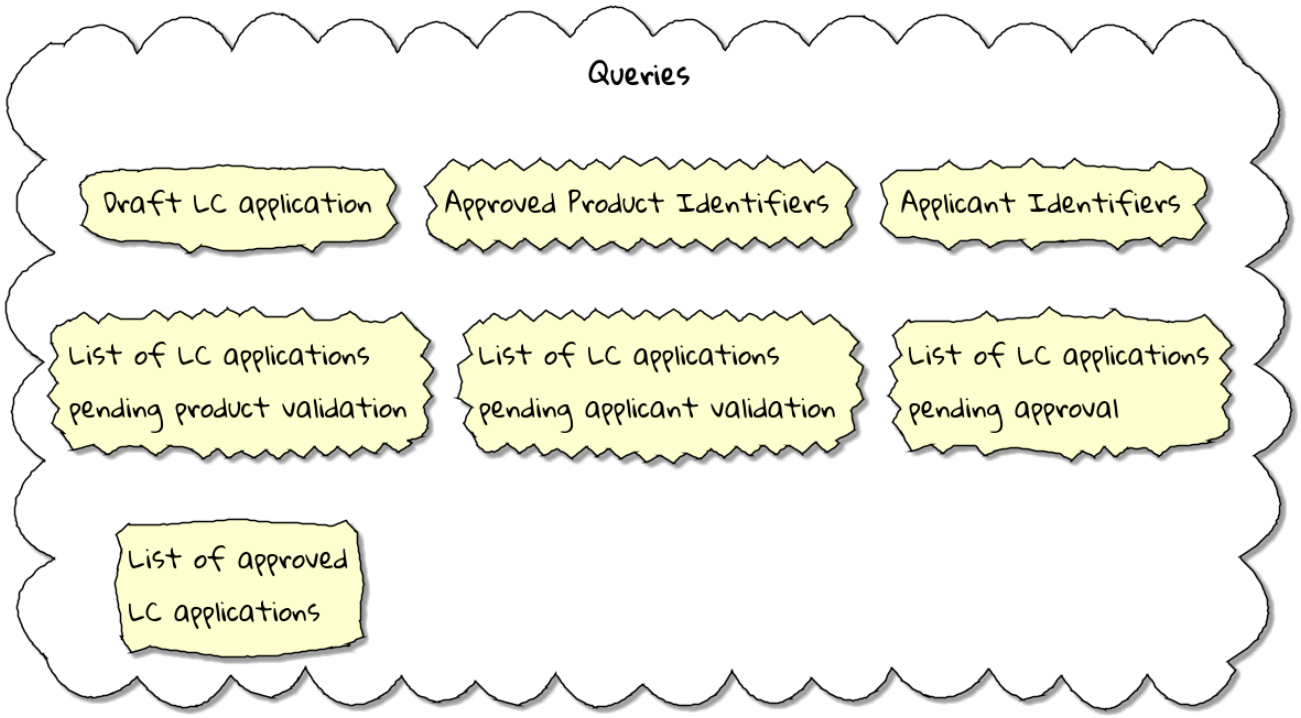


Figure 1- 76. Identified queries

The queries marked in green (in the output from eventstorming session), all require us to expose a collection of LCs in various states. To represent this, we can create an `LCView` as shown here:

The `LCView` class is an extremely simple object devoid of any logic.

```
public class LCView {

    private LCApplicationId id;
    private String applicantId;
    private String clientReference;
    private LCState state;

    // Getters and setters omitted for brevity
}
```

These query models are an absolute necessity to implement basic functionality dictated by business requirements. But it is possible and very likely that we will need additional query models as the system requirements evolve. We will enhance our application to support these queries as and when the need arises.

7.3.3. Creating the query model

As seen in chapter 5, when starting a new LC application, the importer sends a `StartNewLCApplicationCommand`, which results in the `LCApplicationStartedEvent` being emitted as shown here:

```
class LCApplication {
    //..
    @CommandHandler
    public LCApplication(StartNewLCApplicationCommand command) {
        // Validation code omitted for brevity
        // Refer to chapter 5 for details.
        AggregateLifecycle.apply(new LCApplicationStartedEvent(command.getId(),
                command.getApplicantId(), command.getClientReference()));
    }
    //..
}
```

Let's write an event processing component which will listen to this event and construct a query model. When working with the Axon framework, we have a convenient way to do this by annotating the event listening method with the `@EventHandler` annotation.

```
import org.axonframework.eventhandling.EventHandler;
import org.springframework.stereotype.Component;

@Component
class LCApplicationStartedEventHandler {

    @EventHandler                                          ①
    public void on(LCApplicationStartedEvent event) {
        LCView view = new LCView(event.getId(),
                        event.getApplicantId(),
                        event.getClientReference(),
                        event.getState());                 ②
        // Perform any transformations to optimize access
        repository.save(view);                             ③
    }
}
```

① To make any method an event listener, we annotate it with the `@EventHandler` annotation.

② The handler method needs to specify the event that we intend to listen to. There are other arguments that are supported for event handlers. Please refer to the Axon framework documentation for more information.

③ We finally save the query model into an appropriate query store. When persisting this data, we should consider storing it in a form that is optimized for data access. In other words, we want to reduce as much complexity and cognitive load when querying this data.

The `@EventHandler` annotation should not be confused with the `@EventSourcingHandler` annotation which we looked at in chapter 5. The `@EventSourcingHandler` annotation is used to replay events and restore aggregate state when loading event-sourced aggregates on the command side, whereas the `@EventHandler` annotation is used to listen to events outside the context of the aggregate. In other words, the `@EventSourcingHandler` annotation is used exclusively within aggregates, whereas the `@EventHandler` annotation can be used anywhere there is a need to consume domain events. In this case, we are using it to construct a query model.

7.3.4. Query side persistence choices

Segregating the query side this way enables us to choose a persistence technology most appropriate for the problem being solved on the query side. For example, if extreme performance and simple filtering criteria are prime, it may be prudent to choose an in-memory store like Redis or Memcached. If complex search/analytics requirements and large datasets are to be supported, then we may want to consider something like ElasticSearch. Or we may even simply choose to stick with just a relational database. The point we would like to emphasize is that employing CQRS affords a level of flexibility that was previously not available to us.

7.3.5. Exposing a query API

Applicants like to view the LCs they created, specifically those in the draft state. Let's look at how we can implement this functionality. Let's start by defining a simple object to capture the query criteria:

```
import org.springframework.data.domain.Pageable;

public class MyDraftLCsQuery {

    private ApplicantId applicantId;
    private Pageable page;

    // Getters and setters omitted for brevity
}
```

Let's implement the query using spring's repository pattern to retrieve the results for these criteria:

```
import org.axonframework.queryhandling.QueryHandler;

public interface LCViewRepository extends JpaRepository<LCView, LCApplicationId> {

    Page<LCView> findByApplicantIdAndState(          ①
            String applicantId,
            LCState state,
            Pageable page);

    @QueryHandler                                    ②
    default Page<LCView> on(MyDraftLCsQuery query) {
        return findByApplicantIdAndState(            ③
                query.getApplicantId(),
                LCState.DRAFT,
                query.getPage());
    }
}
```

① This is the dynamic spring data finder method we will use to query the database.

② The `@QueryHandler` annotation provided by Axon framework routes query requests to the respective handler.

③ Finally, we invoke the finder method to return results.

In the above example, we have implemented the `QueryHandler` method within the `Repository` itself for brevity. The `QueryHandler` can be placed elsewhere as well.

To connect this to the UI, we add a new method in the `BackendService` (originally introduced in Chapter 6) to invoke the query as shown here:

```
import org.axonframework.queryhandling.QueryGateway;

public class BackendService {

    private final QueryGateway queryGateway;                    ①

    public List<LCView> findMyDraftLCs(String applicantId) {
        return queryGateway.query(                              ②
                new MyDraftLCsQuery(applicantId),
                        ResponseTypes.multipleInstancesOf(LCView.class))
                .join();

    }
}
```

① The Axon framework provides the `QueryGateway` convenience that allows us to invoke the query.For more details on how to use the `QueryGateway`, please refer to the Axon framework documentation.

② We execute the query using the `MyDraftLCsQuery` object to return results.

What we looked at above, is an example of a very simple query implementation where we have a single `@QueryHandler` to service the query results.This implementation returns results as a one-time fetch.Let's look at more complex query scenarios.

7.3.6. Advanced query scenarios

Our focus currently is on active LC applications.Maintaining issued LCs happens in a different bounded context of the system.Consider a scenario where we need to provide a consolidated view of currently active LC applications and issued LCs.In such a scenario, it is necessary to obtain this information by querying two distinct sources (ideally in parallel) — commonly referred to as the scatter-gather[28] pattern.Please refer to the section on scatter-gather queries in the Axon framework documentation for more details.

In other cases, we may want to remain up to date on dynamically changing data.For example, consider a real-time stock ticker application tracking price changes.One way to implement this is by polling for price changes.A more efficient way to do this is to push price changes as and when they occur — commonly referred to as the publish-subscribe[29] pattern.Please refer to the section on subscription queries in the Axon framework documentation for more details.

7.4. Historic event replays

The example we have looked at thus far allows us to listen to events as they occur.Consider a scenario where we need to build a new query from historic events to satisfy an unanticipated new requirement.This new requirement may necessitate the need to create a new query model or in a more extreme case, a completely new bounded context.Another scenario might be when we may need to correct a bug in the way we had built an existing query model and now need to recreate it from scratch.Given that we have a record of all events that have transpired in the event store, we can use replay events to enable us to construct both new and/or correct existing query models with relative ease.

We have used the term *event replay* in the context of reconstituting state of event-sourced aggregate instances (discussed in event-sourced aggregates in Chapter 5).The event replay mentioned here, although similar in concept, is still very different.In the case of domain object event replay, we work with a single aggregate root instance and only load events for that one instance.In this case though, we will likely work with events that span more than one aggregate.

Let's look at how the different types of replays and how we can use each of them.

7.4.1. Types of replays

When replaying events, there are at least two types of replays depending on the requirements we need to meet.Let's look at each type in turn:

- **Full event replay** is one where we replay all the events in the event store. This can be used in a scenario where we need to support a completely new bounded context which is dependent on

this subdomain. This can also be used in cases where we need to support a completely new query model or reconstruct an existing, erroneously built query model. Depending on the number of events in the event store, this can be a fairly long and complex process.

- **Partial/Adhoc event replay** is one where we need to replay all the events on a subset of aggregate instances or a subset of events on all aggregate instances or a combination of both. When working with partial event replays, we will need to specify filtering criteria to select subsets of aggregate instances and events. This means that the event store needs to have the flexibility to support these use cases. Using specialized event store solutions (like Axon Server[30] and EventStoreDB[31] to name a few) can be extremely beneficial.

7.4.2. Event replay considerations

The ability to replay events and create new query models can be invaluable. However, like everything else, there are considerations that we need to keep in mind when working with replays. Let's examine some of these in more detail:

Event store design

As mentioned in Chapter 5, when working with event-sourced aggregates, we persist immutable events in the persistence store. The primary use-cases that we need to support are:

1. Provide consistent and predictable **write** performance when acting as an append-only store.
2. Provide consistent and predictable **read** performance when querying for events using the aggregate identifier.

However, replays (especially partial/adhoc) require the event store to support much richer querying capabilities. Consider a scenario where we found an issue where the amount is incorrectly reported for LCs that were approved during a certain time period only for a certain currency. To fix this issue, we need to:

1. Identify affected LCs from the event store.
2. Fix the issue in the application.
3. Reset the query store for these affected aggregates
4. Do a replay of a subset of events for the affected aggregates and reconstruct the query model.

Identifying affected aggregates from the event store can be tricky if we don't support querying capabilities that allow us to introspect the event payload. Even if this kind of adhoc querying were to be supported, these queries can adversely impact command handling performance of the event store. One of the primary reasons to employ CQRS was to make use of query-side stores for such complex read scenarios.

Event replays seem to introduce a chicken and egg problem where the query store has an issue which can only be corrected by querying the event store. A few options to mitigate this issue are discussed here:

- **General purpose store**: Choose an event store that offers predictable performance for both scenarios (command handling and replay querying).

- **Built-in datastore replication**: Make use of read replicas for event replay querying
- **Distinct datastores**: Make use of two distinct data stores to solve each problem on its own (for example, use a relational database/key-value store for command handling and a search-optimized document store for event replay querying).

Do note that the **distinct datastores** approach for replays is used to satisfy an operational problem as opposed to query-side business use-cases discussed earlier in this chapter. Arguably, it is more complex because the technology team on the command side has to be equipped to maintain more than one database technology.

7.4.3. Event design

Event replays are required to reconstitute state from an event stream. In this article on what it means to be event-driven[32], Martin Fowler talks about three different styles of events. If we employ the *event carried state transfer* approach (in Martin's article) to reconstitute state, it might require us to only replay the latest event for a given aggregate, as opposed to replaying all the events for that aggregate in order of occurrence. While this may seem convenient, it also has its downsides:

- All events may now require to carry a lot of additional information that may not be relevant to that event. Assembling all this information when publishing the event can add to the cognitive complexity on the command side.
- The amount of data that needs to be stored and flow through the wire can increase drastically.
- On the query side, it can increase cognitive complexity when understanding the structure of the event and processing it.

In a lot of ways, this leads back to the CRUD-based vs task-based approach for APIs discussed in Chapter 5. Our general preference is to design events with as lean a payload as possible. However, your experiences may be different depending on your specific problem or situation.

Application availability

In an event-driven system, it is common to accumulate an extremely large number of events over a period of time, even in a relatively simple application. Replaying a large number of events can be time-consuming. Let's look at the mechanics of how replays typically work:

1. We suspend listening to new events in preparation for a replay.
2. Clear the query store for impacted aggregates.
3. Start an event replay for impacted aggregates.
4. Resume listening to new events after replay is complete.

Based on the above, while the replay is running (step 3 above), we may not be able to provide reliable answers to queries that are impacted by the replay. This obviously has an impact on application availability. When using event replays, care needs to be taken to ensure that SLOs[33] (service level objectives) are continued to be met.

7.4.4. Event handlers with side effects

When replaying events, we re-trigger event handlers either to fix logic that was previously erroneous or to support new functionality. Invoking most (if not all) event handlers usually results in some sort of side effect (for example, update a query store). This means that some event handlers may not be running for the first time. To prevent unwanted side effects, it is important to undo the effects of having invoked these event handlers previously or code event handlers in an idempotent manner (for example, by using an *upsert* instead of a simple insert or an update). The effect of some event handlers can be hard (if not impossible) to undo (for example, invoking a command, sending an email or SMS). In such cases, it might be required to mark such event handlers as being ineligible to run during replay. When using the Axon framework, this is fairly simple to do:

```
import org.axonframework.eventhandling.DisallowReplay;

class LCApplicationEventHandlers {
    @EventHandler
    @DisallowReplay  ①
    public void on(CardIssuedEvent event) {
        // Behavior that we don't want replayed
    }
}
```

① The `@DisallowReplay` (or its counterpart `@AllowReplay`) can be used to explicitly mark event handlers ineligible to run during replay.

Events as an API

In an event-sourced system where events are persisted instead of domain state, it is natural for the structure of events to evolve over a period of time. Consider an example of an `BeneficiaryInformationChangedEvent` that has evolved over a period of time as shown here:

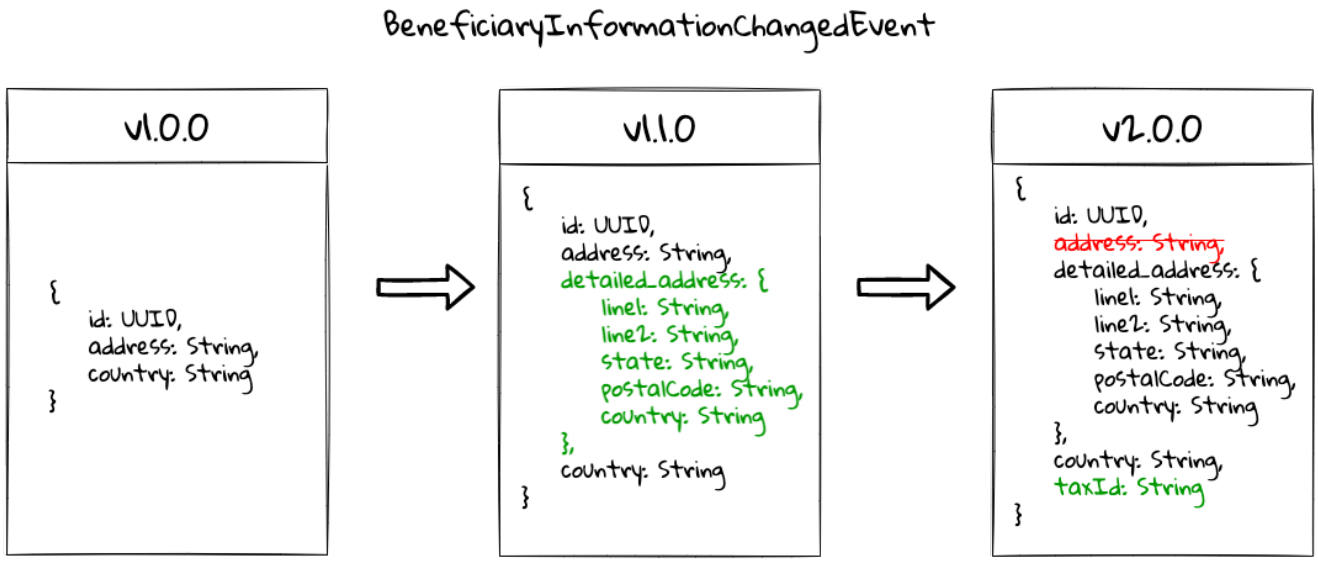


Figure 1- 77. Event evolution

Given that the event store is immutable, it is conceivable that we may have one or more combinations of these event versions for a given LC. This can present a number of decisions we will

need to make when performing an event replay:

- The producer can simply provide the historic event as it exists in the event store and allow consumers to deal with resolving how to deal with older versions of the event.
- The producer can upgrade older versions of events to the latest version before exposing it to the consumer.
- Allow the consumer to specify an explicit version of the event that they are able to work with and upgrade it to that version before exposing it to the consumer.
- Migrate the events in the event store to the latest version as evolutions occur. This may not be feasible given the immutable promise of events in the event store.

Which approach you choose really depends on your specific context and the maturity of the producer/consumer ecosystem. The axon framework makes provisions for a process they call **event upcasting**[34] that allows upgrading events just-in-time before they are consumed. Please refer to the Axon framework documentation for more details.

In an event-driven system, events are your API. This means that you will need to apply the same rigor that one applies to APIs when making lifecycle management decisions (for example, versioning, deprecation, backwards compatibility, etc.).

7.5. Summary

In this chapter we examined how to implement the query side of a CQRS-based system. We looked at how domain events can be consumed in real-time to construct materialized views that can be used to service query APIs. We looked at the different query types that can be used to efficiently access the underlying query models. We rounded off by looking at persistence options for the query side. Finally, we looked at historic event replays and how it can be used to correct errors or introduce new functionality in an event-driven system.

This chapter should give you a good idea of how to build and evolve the query side of a CQRS-based system to meet changing business requirements while retaining all the business logic on the command side.

Thus far, we have looked at how to consume events in a stateless manner (where no two event handlers have knowledge of each other's existence), in the next chapter, we will continue to look at how to consume events, but this time in a stateful manner in the form of long-running user transactions (also known as sagas).

7.6. Questions

- In your context, are you segregating commands and queries (even if the segregation is logical)?
- What read/query models are you able to come up with?
- What do you do if you build a query model, and it turns out to be wrong?

[28] https://www.enterpriseintegrationpatterns.com/BroadcastAggregate.html

[29] https://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html

[30] https://axoniq.io/product-overview/axon

[31] https://www.eventstore.com/eventstoredb

[32] https://martinfowler.com/articles/201701-event-driven.html

[33] https://sre.google/sre-book/service-level-objectives

[34] https://docs.axoniq.io/reference-guide/axon-framework/events/event-versioning#event-upcasting

Chapter 8. Long-Running Workflows

> In the long run, the pessimist may be proven right, but the optimist has a better time on the trip.
>
> — Daniel Reardon

In the previous chapters, we have looked at handling commands and queries within the context of a single aggregate. All the scenarios we have looked at thus far, have been limited to a single interaction. However, not all capabilities can be implemented in the form of a simple request-response interaction, requiring coordination across multiple external systems or human-centric operations or both. In other cases, there may be a need to react to triggers that are nondeterministic (occur conditionally or not at all) and/or be time-bound (based on a deadline). This may require managing business transactions across multiple bounded contexts that may run over a long duration of time, while continuing to maintain consistency (**saga**).

There are at least two common patterns to implement the saga pattern:

- **Explicit orchestration**: A designated component acts as a centralized coordinator — where the system relies on the coordinator to react to domain events to manage the flow.
- **Implicit choreography**: No single component is required to act as a centralized coordinator — where the components simply react to domain events in other components to manage the flow.

By the end of this chapter, you will have learned how to implement sagas using both techniques. You will also have learnt how to work with deadlines when no explicit events occur within the system. You will finally be able to appreciate when/whether to choose an explicit orchestrator or simply stick to implicit choreography without resorting to the use of potentially expensive distributed transactions.

8.1. Technical requirements

To follow the examples in this chapter, you will need access to:

- JDK 1.8+ (We have used Java 17 to compile sample sources)
- Spring Boot 2.4.x
- Axon framework 4.5.3
- JUnit 5.7.x (Included with spring boot)
- Project Lombok (To reduce verbosity)
- Maven 3.x

Please refer to the ch08 directory of the book's accompanying source code repository for complete working examples.

8.2. Continuing our design journey

In Chapter 4 - Domain analysis and modeling, we discussed eventstorming as a lightweight method to clarify business flows. As a reminder, this is the output produced from our eventstorming session:

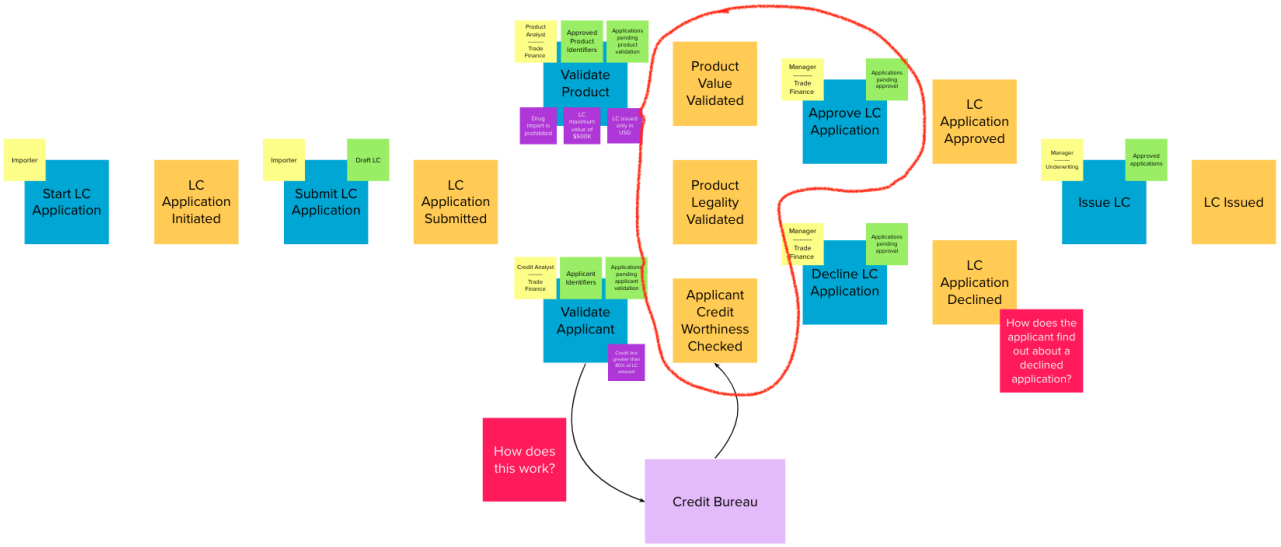


Figure 1- 78. Recap of eventstorming session

As depicted in the visual above, some aspects of Letter of Credit (LC) application processing happens outside our current bounded context), before the trade finance manager makes a decision to either approve or decline the application as listed here:

1. Product value is validated
2. Product legality is validated
3. Applicant's credit worthiness is validated

Currently, the final approval is a manual process. It is pertinent to note that the product value and legality checks happen as part of the work done by the product analysis department, whereas applicant credit worthiness checks happens in the credit analysis department. Both departments make use of their own systems to perform these functions and notify us through the respective events. An LC application is **not ready** to either be approved or declined until **each** of these checks are completed. Each of these processes happen mostly independently of the other and may take a nondeterministic amount of time (typically in the order of a few days). After these checks have happened, the trade finance manager manually reviews the application and makes the final decision.

Given the growing volumes of LC applications received, the bank is looking to introduce a process optimization to automatically approve applications with an amount below a certain threshold (**USD 10,000** at this time). The business has deemed that the three checks above are sufficient and that no further human intervention is required when approving such applications.

From an overall system perspective, it is pertinent to note that the product analyst system notifies us through the `ProductValueValidatedEvent` and `ProductLegalityValidatedEvent`, whereas the credit analyst system does the same through the `ApplicantCreditValidatedEvent` event. Each of these events can and indeed happen independently of the other. For us to be able to auto-approve

applications our solution needs to wait for all of these events to occur. Once these events have occurred, we need to examine the outcome of each of these events to finally make a decision.

In this context, we are using the term ***long-running*** to denote a complex business process that takes several steps to complete. As these steps occur, the process transitions from one state to another. In other words, we are referring to a **state machine**[35]. This is not to be confused with a long-running software process (for example, a complex SQL query or an image processing routine) that is computationally intensive.

As is evident from the diagram above, the LC auto-approval functionality is an example of a long-running business process where *some thing* in our system needs to keep track of the fact that these independent events have occurred before proceeding further. Such functionality can be implemented using the saga pattern. Let's look at how we can do this.

8.3. Implementing sagas

Before we delve into how we can implement this auto-approval functionality, let's take a look at how this works from a logical perspective as shown here:

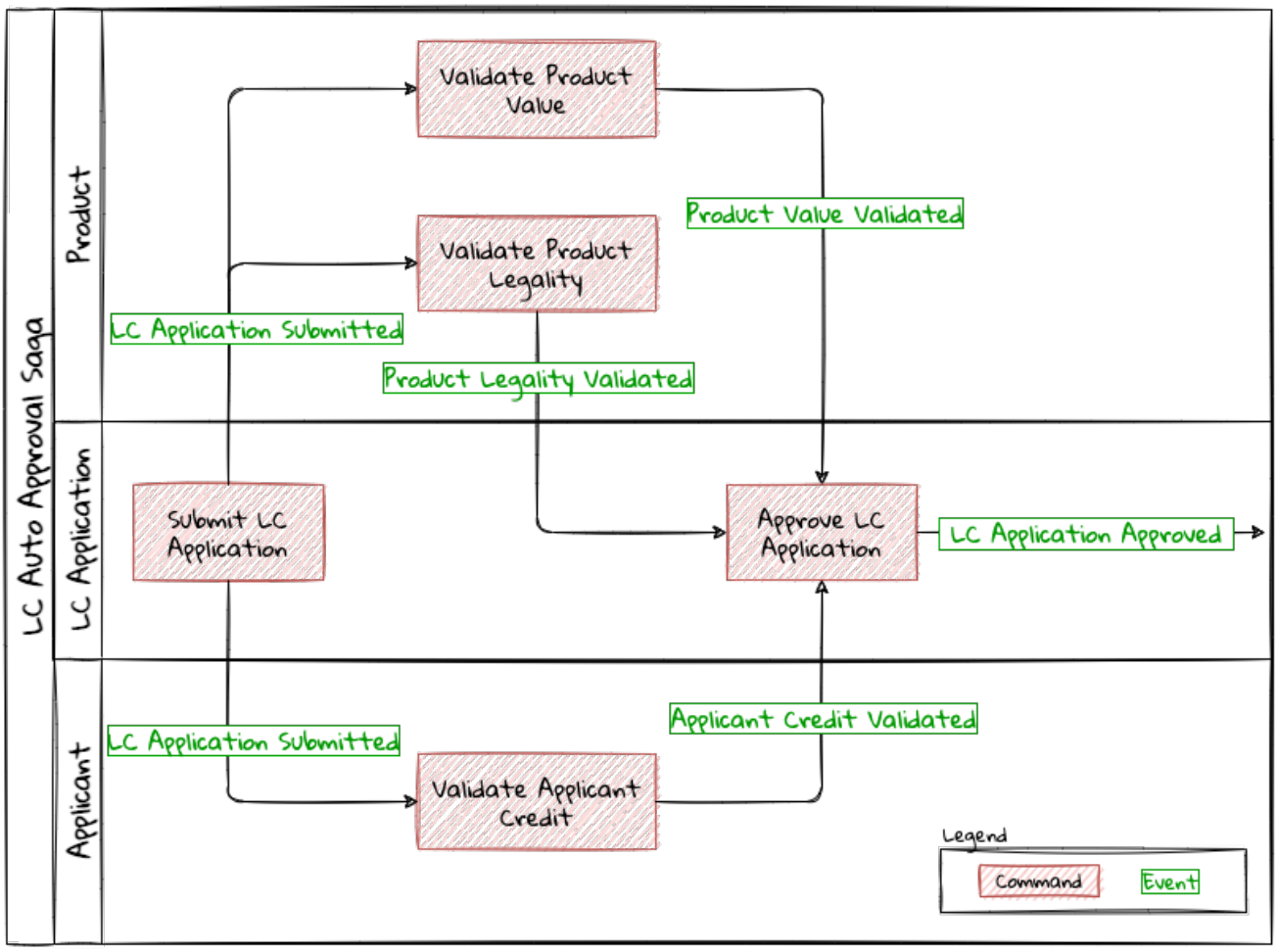


Figure 1- 79. Auto-approval process — logical view

As is depicted in the visual above, there are three bounded contexts in play:

1. LC Application (the bounded context we have been implementing thus far)
2. The Applicant bounded context
3. The Product bounded context

The flow gets triggered when the LC application is submitted. This in turn sets in motion three independent functions that establish the:

1. Value of the product being transacted
2. Legality of the product being transacted
3. Credit worthiness of the applicant

LC approval can proceed only after **all** of these functions have completed. Furthermore, to **auto-approve**, all of these checks have to complete **favorably** and as mentioned earlier, the LC amount has to be lesser than the **USD 10000** threshold.

As shown in the event storming artifact, the `LC Application` aggregate is able to handle an `ApproveLCApplicationCommand`, which results in a LCApplicationApprovedEvent`. To auto-approve, this command needs to be invoked automatically when all the conditions mentioned earlier are satisfied. We are building an event-driven system, and we can see that each of these validations produce events when their respective actions complete. There are at least two ways to implement this functionality:

1. **Orchestration**: where a single component in the system coordinates the state of the flow and triggers subsequent actions as necessary.
2. **Choreography**: where actions in the flow are triggered without requiring an explicit coordinating component.

Let's examine these methods in more detail:

8.3.1. Orchestration

When implementing sagas using an orchestrating component, the system looks similar to the one depicted here:

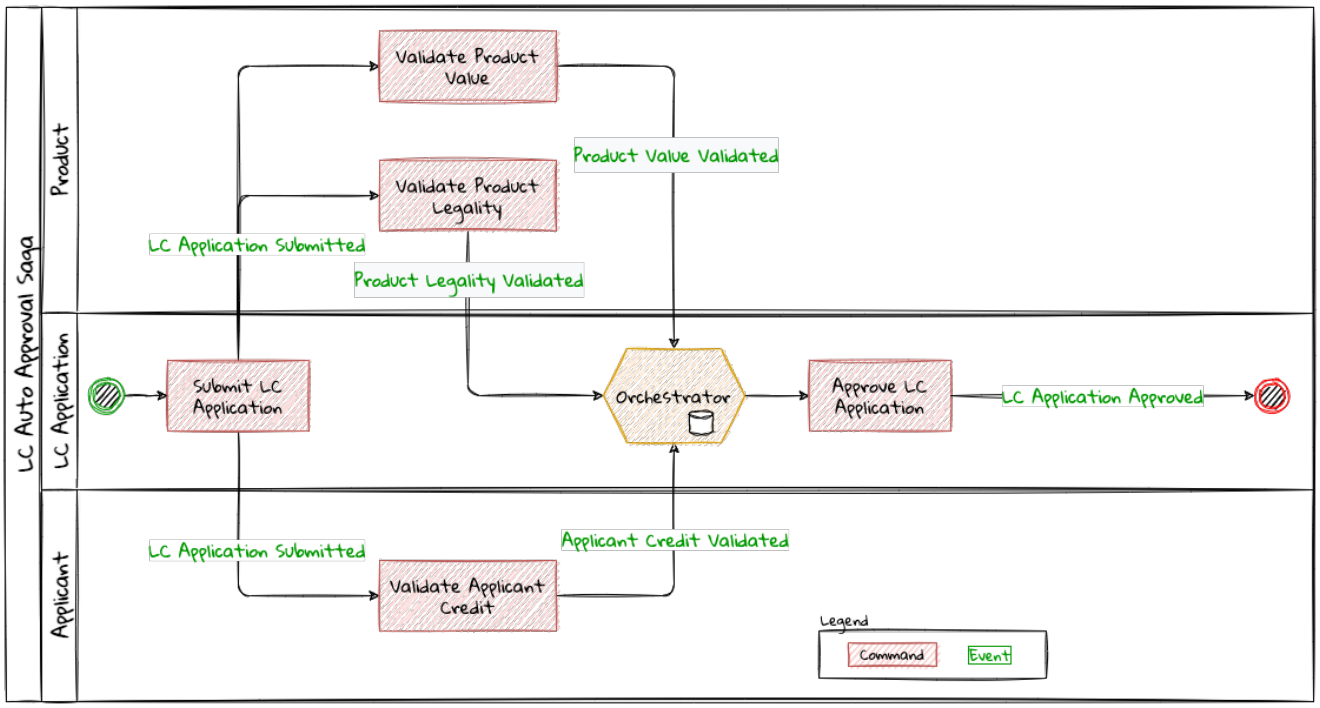


Figure 1- 80. Saga implementation using an orchestrator

The orchestrator starts tracking the flow when the LC application is submitted. It will then need to wait for each of the `ProductValueValidatedEvent`, `ProductLegalityValidatedEvent` and `ApplicantCreditValidatedEvent` events to occur and decide if it is appropriate to trigger the `ApproveLCApplicationCommand`. Finally, the saga lifecycle ends unconditionally when the LC application is approved. There are other conditions that may cause the saga to end abruptly. We will examine those scenarios in detail later. It is pertinent to note that there will be a **distinct** auto-approval saga instance for each LC application that gets submitted. Let's look at how to implement this functionality using the Axon framework. As usual, let's test drive this functionality that a new auto approval saga instance is created when an LC application is submitted:

```
import org.axonframework.test.saga.FixtureConfiguration;
import org.axonframework.test.saga.SagaTestFixture;

class AutoApprovalSagaTests {

    private FixtureConfiguration fixture;                                    ①

    @BeforeEach
    void setUp() {
        fixture = new SagaTestFixture<>(AutoApprovalSaga.class);             ①
    }

    @Test
    void shouldStartSagaOnSubmit() {
        final LCApplicationId lcApplicationId = LCApplicationId.randomId();
        fixture.givenNoPriorActivity()                                       ②
                .whenPublishingA(                                            ③
                        new LCApplicationSubmittedEvent(lcApplicationId,
                            AUTO_APPROVAL_THRESHOLD_AMOUNT
                               .subtract(ONE_DOLLAR)))
                .expectActiveSagas(1);                                       ④
    }

}
```

① We make use of the Axon provided `FixtureConfiguration` and `SagaTestFixture` that allow us to test saga functionality.

② Given no prior activity has occurred (from the perspective of the saga)

③ When a `LCApplicationSubmittedEvent` is published

④ We expect one active saga to exist

The implementation to make this test pass looks like:

```
import org.axonframework.modelling.saga.SagaEventHandler;
import org.axonframework.modelling.saga.StartSaga;
import org.axonframework.spring.stereotype.Saga;

@Saga                                                                ①
public class AutoApprovalSaga {

    @SagaEventHandler(associationProperty = "lcApplicationId") ②
    @StartSaga                                                       ③
    public void on(LCApplicationSubmittedEvent event) {
        //
    }
}
```

① When working with Axon and Spring, the orchestrator is annotated with the `@Saga` annotation to

mark it as a spring bean. In order to track each submitted LC application, the `@Saga` annotation is prototype-scoped (as opposed to singleton-scoped), to allow creation of multiple saga instances. Please refer to the Axon and Spring documentation for more information.

② The saga listens to the `LCApplicationSubmittedEvent` to keep track of the flow (as denoted by the `@SagaEventHandler` annotation). Conceptually, the `@SagaEventHandler` annotation is very similar to the `@EventHandler` annotation that we discussed previously in Chapter 7. However, the `@SagaEventHandler` annotation is used specifically for event listeners within a saga. The `associationProperty` attribute on the `@SagaEventHandler` annotation causes this event handler method to get invoked only for the saga with matching value of the `lcApplicationId` attribute in the event payload. Also, the `@SagaEventHandler` is a transaction boundary. Every time such a method completes successfully, the Axon framework commits a transaction, thereby allowing it to keep track of state stored in the saga. We will look at this in more detail shortly.

③ Every saga needs to have at least one `@SagaEventHandler` method that is also annotated with the `@StartSaga` annotation to denote the beginning of the saga.

We have a requirement that an LC cannot be auto-approved if its amount exceeds the threshold (USD 10000 in our case). The test for this scenario looks like:

```
class AutoApprovalSagaTests {
    //...

    @Test
    void shouldEndSagaImmediatelyIfAmountGreaterThanAutoApprovalThreshold() {
        final LCApplicationId lcApplicationId = LCApplicationId.randomId();
        fixture.givenAggregate(lcApplicationId.toString()).published()
                .whenPublishingA(
                        new LCApplicationSubmittedEvent(lcApplicationId,
                            AUTO_APPROVAL_THRESHOLD_AMOUNT.add(ONE_DOLLAR))) ①
                .expectActiveSagas(0);                                       ②
    }
}
```

① When the LC amount exceeds the auto approval threshold amount

② We expect no active sagas to exist for that LC

The implementation to satisfy this condition looks like this:

```
import org.axonframework.modelling.saga.SagaLifecycle;

@Saga
public class AutoApprovalSaga {

    @SagaEventHandler(associationProperty = "lcApplicationId")
    @StartSaga
    public void on(LCApplicationSubmittedEvent event) {
        if (AUTO_APPROVAL_THRESHOLD_AMOUNT.isLessThan(event.getAmount())) { ①
            SagaLifecycle.end();                                            ②
        }
    }
}
```

① We check for the condition of the LC amount being greater than the threshold amount

② If so, we end the saga using the framework provided `SagaLifecycle.end()` method. Here we end the saga programmatically. It is also possible to declaratively end the saga as well using the `@EndSaga` annotation when the `LCApplicationApprovedEvent` occurs. Please refer to the full code examples included with this chapter for more information.

We need to auto-approve the saga if all of `ApplicantCreditValidatedEvent`, `ProductLegalityValidatedEvent` and `ProductValueValidatedEvent` have occurred successfully. The test to verify this functionality is shown here:

```
class AutoApprovalSagaTests {

    @Test
    void shouldAutoApprove() {
        // Initialization code removed for brevity

        fixture.givenAggregate(lcApplicationId.toString())
            .published(submitted, legalityValidated, valueValidated)        ①
                .whenPublishingA(applicantValidated)                        ②
                .expectActiveSagas(1)                                       ③
                .expectDispatchedCommands(
                        new ApproveLCApplicationCommand(lcApplicationId));  ④
    }
}
```

① Given that the LC application has been submitted and the `ProductValueValidatedEvent` and the `ProductLegalityValidatedEvent` have occurred successfully.

② When the `ApplicantCreditValidatedEvent` is published

③ We expect one active saga instance AND

④ We expect the `ApproveLCApplicationCommand` to be dispatched for that LC

The implementation for this looks like:

```
class AutoApprovalSaga {

    private boolean productValueValidated;                              ①
    private boolean productLegalityValidated;                           ①
    private boolean applicantValidated;                                 ①

    @Autowired
    private transient CommandGateway gateway;                           ②

    // Other event handlers omitted for brevity

    @SagaEventHandler(associationProperty = "lcApplicationId")
    public void on(ApplicantCreditValidatedEvent event) {               ③
        if (event.getDecision().isRejected()) {                         ④
            SagaLifecycle.end();
        } else {
            this.applicantValidated = true;                             ⑤
            if (productValueValidated && productLegalityValidated) {    ⑥
                LCApplicationId id = event.getLcApplicationId();
                gateway.send(ApproveLCApplicationCommand.with(id));     ⑦
            }
        }
    }

    // Other event handlers omitted for brevity
}
```

① As mentioned previously, sagas can maintain state. In this case, we are maintaining three boolean variables, each to denote the occurrence of the respective event.

② We have declared the Axon `CommandGateway` as a `transient` member because we need it to dispatch commands, but not be persisted along with other saga state.

③ This event handler intercepts the `ApplicantCreditValidatedEvent` for the specific LC application (as denoted by the `associationProperty` in the @SagaEventHandler annotation).

④ If the decision from the `ApplicantCreditValidatedEvent` is rejected, we end the saga immediately.

⑤ Otherwise, we *remember* the fact that the applicant's credit has been validated.

⑥ We then check to see if the product's value and legality have already been validated.

⑦ If so, we issue the command to auto-approve the LC.

The logic in the `ProductValueValidatedEvent` and `ProductLegalityValidatedEvent` is very similar to that in the saga event handler for the `ApplicantCreditValidatedEvent`. We have omitted it here for brevity. Please refer to the source code for this chapter for the full example along with the tests.

Finally, we can end the saga when we receive the LCApplicationApprovedEvent for this application.

```
class AutoApprovalSagaTests {
    @Test
    @DisplayName("should end saga after auto approval")
    void shouldEndSagaAfterAutoApproval() {
        // Initialization code omitted for brevity

        fixture.givenAggregate(lcApplicationId.toString())
                .published(
                    submitted, applicantValidated,
                    legalityValidated, valueValidated)                          ①
                .whenPublishingA(new LCApplicationApprovedEvent(lcApplicationId)) ②
                .expectActiveSagas(0)                                           ③
                .expectNoDispatchedCommands();                                  ④
    }
}
```

① Given that the LC has been submitted and all the validations have been completed successfully.

② When a `LCApplicationApprovedEvent` is published.

③ We expect zero active sagas to be running.

④ And we also expect to not dispatch any commands.

Now that we have looked at how to implement sagas using an orchestrator, let's examine some design decisions that we may need to consider when working with them.

Pros

- **Complex workflows**: Having an explicit orchestrator can be very helpful when dealing with flows that involve multiple participants and have a lot of conditionals because the orchestrator can keep track of the overall progress in a fine-grained manner.
- **Testing**: As we have seen in the implementation above, testing flow logic in isolation is relatively straightforward.
- **Debugging**: Given that we have a single coordinator, debugging the current state of the flow can be relatively easier.
- **Handling exceptions**: Given that the orchestrator has fine-grained control of the flow, recovering gracefully from exceptions can be easier.
- **System knowledge**: Components in different bounded contexts do not need to have knowledge of each other's internals (e.g. commands and events) to progress the flow.
- **Cyclic dependencies**: Having a central coordinator allows avoiding accidental cyclic dependencies between components.

Cons

- **Single point of failure**: From an operational perspective, orchestrators can become single points of failure because they are the only ones that have knowledge of the flow. This means that these components need to exhibit higher resilience characteristics as compared to other components.

- **Leaking of domain logic**: In an ideal world, the aggregate will remain the custodian of all domain logic. Given that the orchestrator is also stateful, business logic may inadvertently shift to the orchestrator. Care should be taken to ensure that the orchestrator only has flow-control logic while business invariants remains within the confines of the aggregate.

The above implementation should give you a good idea of how to implement a saga orchestrator. Now let's look at how we can do this without the use of an explicit orchestrator.

8.3.2. Choreography

Saga orchestrators keep track of the current state of the flow, usually making use of some kind of data store. Another way to implement this functionality is without using any stateful component. Logically, this looks like the setup shown in the diagram here:

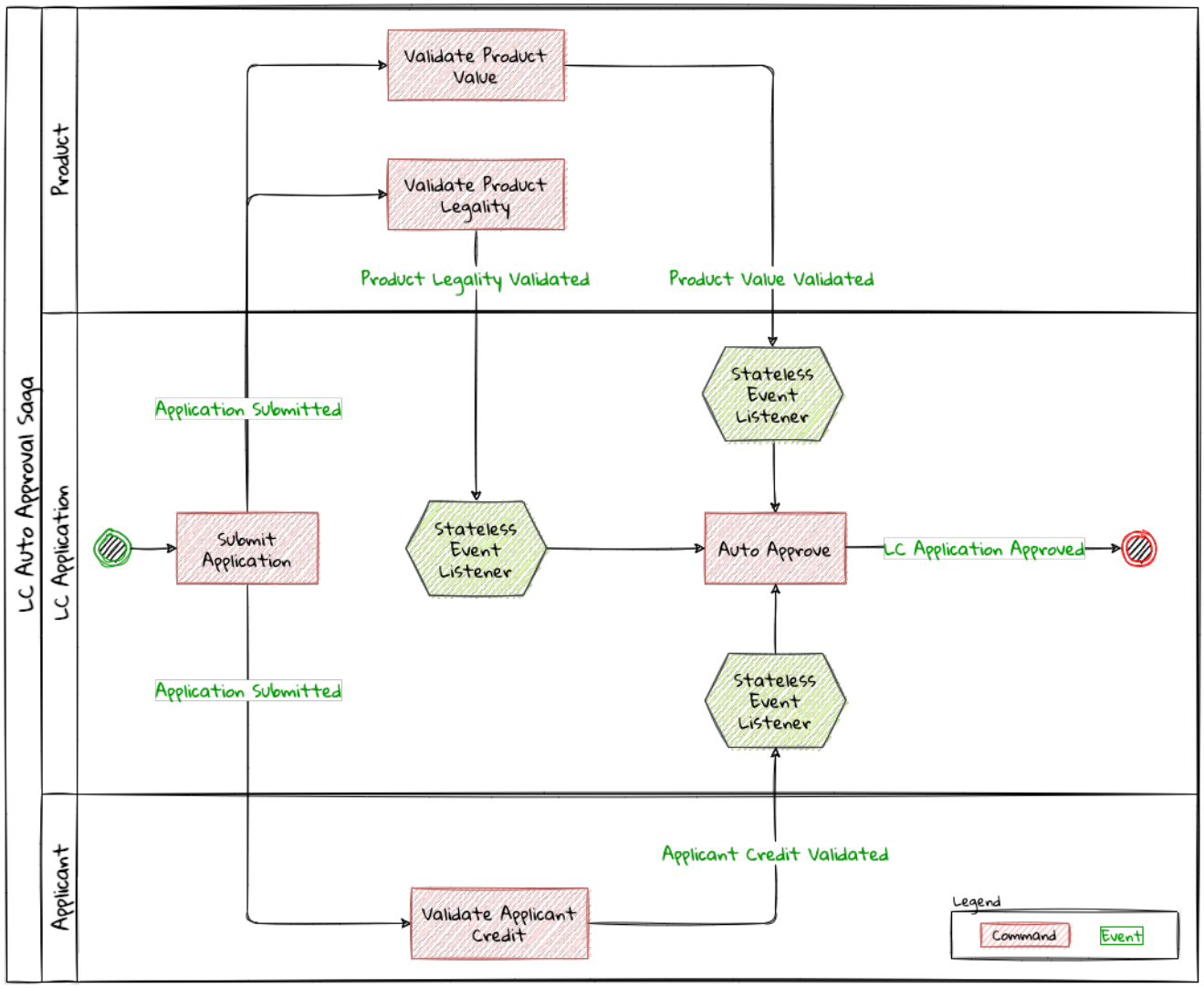


Figure 1- 81. Saga implementation using choreography

As you can see, there is no single component that tracks the saga lifecycle. However, to make the auto-approval decision, each of these stateless event handlers need to have knowledge of the same three events having occurred:

1. Product value is validated
2. Product legality is validated

3. Applicant's credit worthiness is validated

Given that the event listeners themselves are stateless, there are at least two ways to provide this information to them:

1. Each of the events carry this information in their respective payloads.
2. The event listeners query the source systems (in this case, the product and applicant bounded contexts respectively).
3. The LC application bounded context maintains a query model to keep track of these events having occurred.

Just like in the orchestrator example, when all events have occurred and the LC amount is below the specified threshold, these event listeners can issue the `ApproveLCApplicationCommand`.

We will skip covering code examples for the choreography implementation because this is no different from the material we have covered previously in this and prior chapters.

Now that we have looked at how to implement the choreography style of sagas, let's examine some design decisions that we may need to consider when working with them.

Pros

- **Simple workflows**: For simple flows, the choreography approach can be relatively straightforward because it does not require the overhead of an additional coordinating component.
- **No single points of failure**: From an operational perspective, there is one less high resilience component to worry about.

Cons

- **Workflow tracking**: Especially with complex workflows that involve numerous steps and conditionals, tracking and debugging the current state of the flow may become challenging.
- **Cyclic dependencies**: It is possible to inadvertently introduce cyclic dependencies among components when workflows become gnarly.

Sagas enable applications to maintain data and transactional consistency when more than one bounded context is required to complete the business functionality without having to resort to using *distributed transactions*[36]. However, it does introduce a level of complexity to the programming model, especially when it comes to handling failures. We will look at exception handling in a lot more detail when we discuss working with distributed systems in upcoming chapters. Let's look at how to progress flows when there are no explicit stimuli by looking at how deadlines work.

8.4. Handling deadlines

Thus far, we have looked at events that are caused by human (for example, the applicant

submitting an LC application) or system (for example, the auto-approval of an LC application) action. However, in an event-driven system, not all events occur due to an explicit human or system stimulus. Events may need to be emitted either due to inactivity over a period of time, or on a recurring schedule based on prevailing conditions.

For example, let's examine the case where the bank needs *submitted LC applications* to be decisioned as quickly as possible. When applications are not acted upon by the trade finance managers within ten calendar days, the system should send them reminders.

To deal with such inactivity, we need a means to trigger system action (read — emit events) based on the passage of time — in other words, perform actions when a *deadline* expires. In a happy path scenario, we expect either the user or the system to take certain action. In such cases, we will also need to account for cases we will need to cancel the trigger scheduled to occur on deadline expiry. Let's look at how to test-drive this functionality.

```
class LCApplicationAggregateTests {
    //...
    @Test
    void shouldCreateSubmissionReminderDeadlineWhenApplicationIsSubmitted() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                "My LC", LCState.DRAFT),
                        new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                        new MerchandiseChangedEvent(id, merchandise()))

                .when(new SubmitLCApplicationCommand(id)) ①
                .expectEvents(new LCApplicationSubmittedEvent(id,
                                THOUSAND_DOLLARS))

                .expectScheduledDeadlineWithName(
                        Duration.ofDays(10),
                        LC_APPROVAL_PENDING_REMINDER);    ②
    }
}
```

① When the LC application is submitted

② We expect a deadline for the reminder to be scheduled

The implementation for this is fairly straightforward:

```
import org.axonframework.deadline.DeadlineManager;

class LCApplication {
    //...
    @CommandHandler
    public void on(SubmitLCApplicationCommand command,
                   DeadlineManager deadlineManager) { ①
        assertPositive(amount);
        assertMerchandise(merchandise);
        assertInDraft(state);
        apply(new LCApplicationSubmittedEvent(id, amount));

        deadlineManager.schedule(Duration.ofDays(10),  ②
            "LC_APPROVAL_REMINDER",
            LCApprovalPendingNotification.first(id));  ③
    }
    //...
}
```

① To allow working with deadlines, the Axon framework provides a `DeadlineManager` that allows working with deadlines. This is injected into the command handler method.

② We use the `deadlineManager` to schedule a named deadline (`"LC_APPROVAL_REMINDER"` in this case) that will expire in 10 days.

③ When the deadline is met, it will result in a `LCApprovalPendingNotification` which can be handled just like a command. Except in this case, the behavior is triggered by the passage of time.

If no action is taken for ten days, this is what we expect:

```
class LCApplication {

    @Test
    void shouldTriggerApprovalPendingEventTenDaysAfterSubmission() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                "My LC", LCState.DRAFT),
                        new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                        new MerchandiseChangedEvent(id, merchandise()))
                .andGivenCommands(new SubmitLCApplicationCommand(id)) ①
                .whenThenTimeElapses(Duration.ofDays(10))             ②
                .expectDeadlinesMet(
                        LCApprovalPendingNotification.first(id))      ③
                .expectEvents(new LCApprovalPendingEvent(id));        ④
    }
}
```

① Given that the LC application is submitted.

② When the period of ten days elapses.

③ The deadline should be met.

④ And the `LCApprovalPendingEvent` should be emitted.

Let's look at how to implement this:

```
import org.axonframework.deadline.annotation.DeadlineHandler;

class LCApplication {

    @DeadlineHandler(deadlineName = "LC_APPROVAL_REMINDER")       ①
    public void on(LCApprovalPendingNotification notification) {  ②

        AggregateLifecycle.apply(new LCApprovalPendingEvent(id)); ③

    }
}
```

① Deadlines are handled by annotating handler methods with the `@DeadlineHandler` annotation. Note that the same deadline name used previously is being referenced here.

② This is the deadline handler method and uses the same payload that was passed along when it was scheduled.

③ We emit the `LCApprovalPendingEvent` when the deadline expires.

The deadline handling logic should only be triggered if no action is taken. However, if the LC is either approved or rejected within a duration of ten days, none of this behavior should be triggered:

```
class LCApplicationAggregateTests {
    //...
    @Test
    void shouldNotTriggerPendingReminderIfApplicationIsApprovedWithinTenDays() {
        final LCApplicationId id = LCApplicationId.randomId();
        fixture.given(new LCApplicationStartedEvent(id, ApplicantId.randomId(),
                                    "My LC", LCState.DRAFT),
                            new LCAmountChangedEvent(id, THOUSAND_DOLLARS),
                            new MerchandiseChangedEvent(id, merchandise()))
                .andGivenCommands(new SubmitLCApplicationCommand(id)) ①

                .when(new ApproveLCApplicationCommand(id))            ②
                .expectEvents(new LCApplicationApprovedEvent(id))
                .expectNoScheduledDeadlines();                        ③
    }

    @Test
    void shouldNotTriggerPendingReminderIfApplicationIsDeclinedWithinTenDays() {
        // Test code is very similar. Excluded for brevity
    }

}
```

① Given that the LC application is submitted

② When it is approved within a duration of ten days (in this case, almost immediately)

③ We expect no scheduled deadlines

And the implementation for this looks like:

```
class LCApplication {
    //...
    @CommandHandler
    public void on(ApproveLCApplicationCommand command,
                   DeadlineManager deadlineManager) {
        assertInSubmitted(state);
        AggregateLifecycle.apply(new LCApplicationApprovedEvent(id));
        deadlineManager.cancelAllWithinScope("LC_APPROVAL_REMINDER"); ①
    }

    @CommandHandler
    public void on(DeclineLCApplicationCommand command,
                   DeadlineManager deadlineManager) {
        assertInSubmitted(state);
        AggregateLifecycle.apply(new LCApplicationDeclinedEvent(id));
        deadlineManager.cancelAllWithinScope("LC_APPROVAL_REMINDER"); ①
    }

    //...
}
```

① We cancel all the deadlines with the name `LC_APPROVAL_REMINDER` (in this case, we only have one deadline with that name) within the scope of this aggregate.

8.5. Summary

In this chapter, we examined how to work with long-running workflows using sagas and the different styles we can use to implement them. We also looked at the implications of using explicit orchestration versus implicit choreography. We finally looked at how we can handle deadlines when there are no user-initiated actions.

You should have learnt how sagas can act as a first-class citizen in addition to aggregates when designing a system that makes use of domain-driven design principles.

In the next chapter, we will look at how we can interact with external systems while respecting bounded context boundaries between core and peripheral systems.

8.6. Questions

- Are you seeing the need to implement long-running workflows in your current ecosystem?
- Do you see yourself picking one style over the other most of the time?
- Are you using external deadline handlers (for instance, batch jobs) in your existing systems as opposed to embedding time-based logic in the core?

8.7. Further reading

Title	Author	Location
Saga persistence and event-driven architectures	Udi Dahan	https://udidahan.com/2009/04/20/saga-persistence-and-event-driven-architectures/
Sagas solve stupid transaction timeouts	Udi Dahan	https://udidahan.com/2008/06/23/sagas-solve-stupid-transaction-timeouts/
Microservices — when to react vs. orchestrate	Andrew Bonham	https://medium.com/capital-one-tech/microservices-when-to-react-vs-orchestrate-c6b18308a14c
Saga orchestration for microservices using the outbox pattern	Gunnar Morling	https://www.infoq.com/articles/saga-orchestration-outbox/
Patterns for distributed transactions within a microservices architecture	Keyang Xiang	https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture

[35] https://en.wikipedia.org/wiki/state_machine

[36] https://en.wikipedia.org/wiki/Distributed_transaction

Chapter 9. Integrating with External Systems

> Wholeness is not achieved by cutting off a portion of one's being, but by integration of the contraries.
>
> — Carl Jung

Thus far, we have used DDD to implement a robust core for our application. However, most solutions (by extension — bounded contexts) usually have both upstream and downstream dependencies which change at a pace that is different from these core components. To maintain both agility, reliability and enable loose coupling, it is important to integrate with peripheral systems in a manner that shields the core from everything else that surrounds it.

In this chapter, we will look at the LC application processing solution and examine means by which we can integrate with other components in the ecosystem. You will learn to recognize relationship patterns between components. We will round off by looking at common implementation patterns when integrating with other applications.

9.1. Continuing our design journey

From our domain analysis in the earlier chapters, we have arrived at four bounded contexts for our application as depicted here:

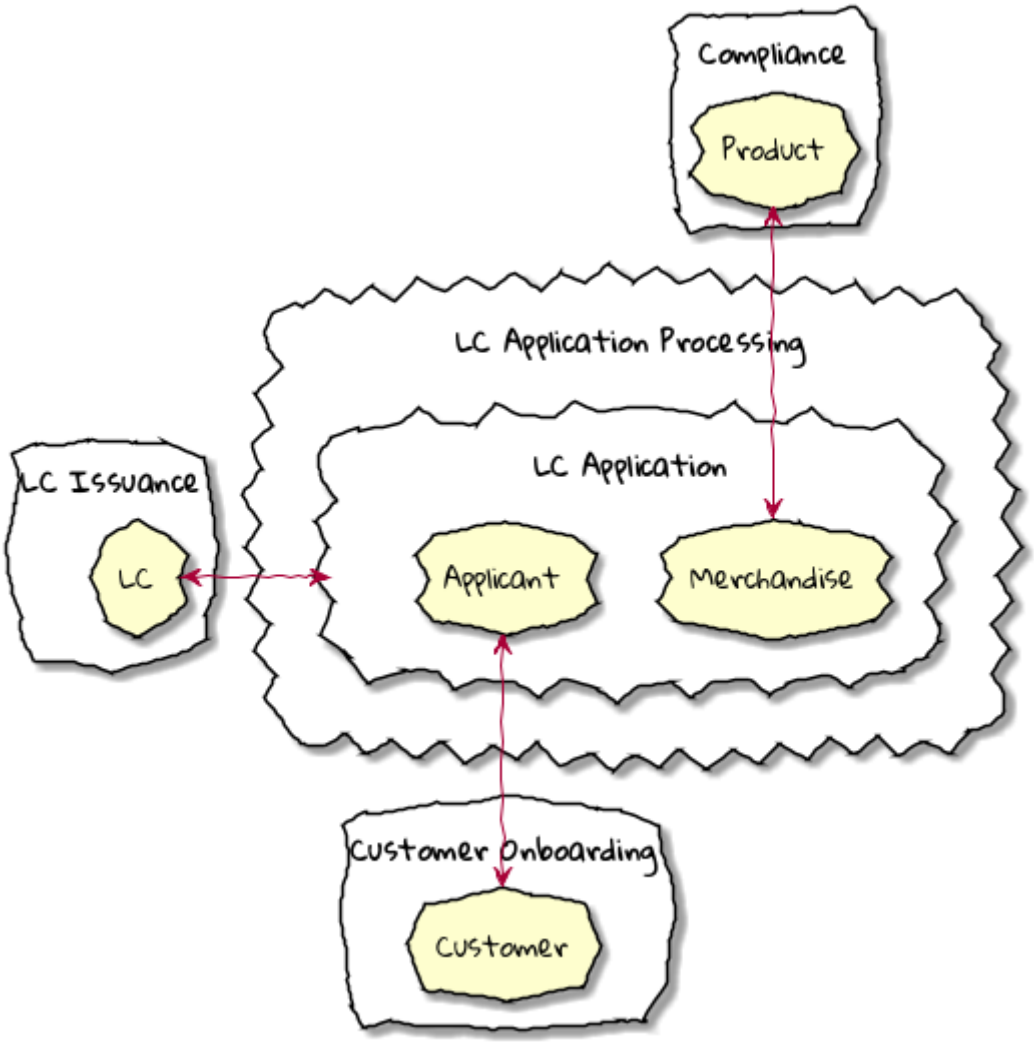


Figure 1- 82. Relationship between bounded contexts

Thus far, our focus has been on the implementation of the internals of the **LC Application** bounded context. While the LC Application bounded context is independent of the other bounded contexts, it is not completely isolated from them. For example, when processing an LC application, we need to perform merchandise and applicant checks which require interactions with the **Compliance** and **Customer Onboarding** bounded contexts respectively. This means that these bounded contexts have a relationship with each other. These relationships are driven by the nature of collaboration between the teams working on the respective bounded contexts. Let's examine how these team dynamics influence integration mechanisms between bounded contexts in a way that continues to preserve their individual integrity.

9.2. Bounded context relationships

We need bounded contexts to be as independent as possible. However, this does not mean that bounded contexts are completely isolated from each other. Bounded contexts need to collaborate with others to provide business value. Whenever there is collaboration required between two bounded contexts, the nature of their relationship is not only influenced by their individual goals and priorities, but also by the prevailing organizational realities. In a high performing environment, it is fairly common to have a single team assume ownership of a bounded context. The relationships between the teams owning these bounded contexts, play a significant role in influencing the integration patterns employed to arrive at a solution. At a high level, there are two

categories of relationships:

1. Symmetric
2. Asymmetric

Let's look at these relationship types in more detail.

9.2.1. Symmetric relationship patterns

Two teams can be said to have a symmetric relationship when they have an equal amount of influence in the decision-making process to arrive at a solution. Both teams are in a position to and indeed, do contribute more or less equally towards the outcome, as depicted here :

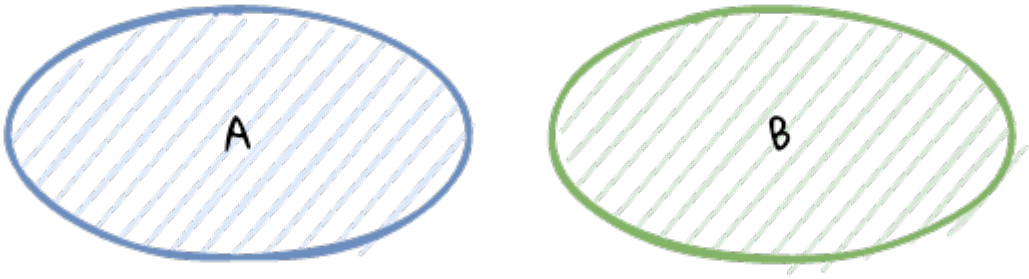


Figure 1- 83. Both teams have an equal say in influencing the solution

There are three variations of symmetric relationships, each of which we outline in more detail in the following sub-sections.

Partnership

In a partnership, both teams integrate in an ad hoc manner. There are no fixed responsibilities assigned when needing complete integration work. Each team picks up work as and when needed without the need for any specific ceremony or fanfare. The nature of the integration is usually two-way with both teams exchanging solution artifacts as and when needed. Such relationships require extremely high degrees of collaboration and understanding of the work done by both teams, as depicted here :

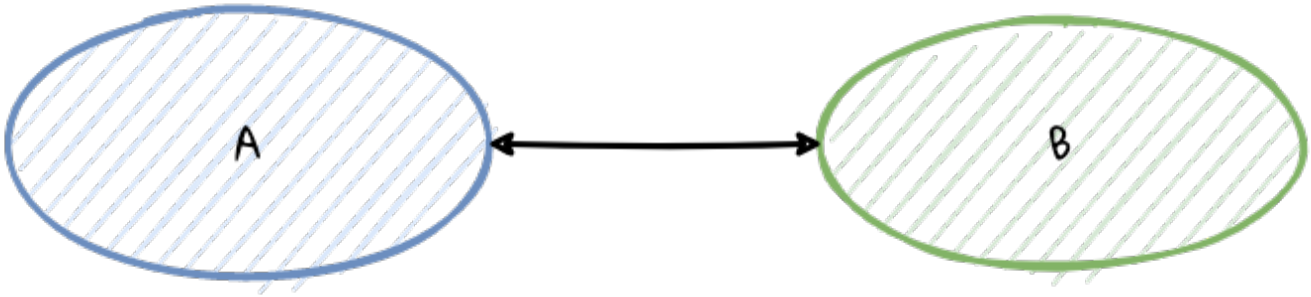


Figure 1- 84. There is an ad hoc, mutual dependency between teams in a partnership relationship

Example

A web front-end team working in close collaboration with the APIs team building the BFFs for the front-end. The BFF team creates experience APIs meant to be used exclusively by the front-end. To fulfill any functionality, the front-end team requires capabilities to be exposed by the APIs team. On the other hand, the APIs team is dependent on the front-end team to provide advice on what

capabilities to build and the order in which to build them. Both teams freely make use of each other's domain models (for example, the same set of request and response objects that define the API) to implement functionality. Such reuse happens mostly arbitrarily and when API changes happen, the both teams coordinate changes to keep things working.

When to use

Partnership between teams require high levels of collaboration, trust and understanding. Teams tend to use this when team boundaries are informal. It also helps if these teams are co-located and/or have a significant working time overlap.

Potential pitfalls

Partnership relationships between teams can lead to a situation where individual team responsibilities become very unclear leading the solution towards the dreaded *big ball of mud*.

Shared kernel

Unlike in a partnership, when using a shared kernel, teams have a clear understanding of the solution artifacts and models they choose to share between themselves. Both teams take equal responsibility in the upkeep of these shared artifacts.

Example

The *LC Application Processing* and *Customer Onboarding* teams in our LC application may choose to use a common model to represent the `CustomerCreditValidatedEvent`. Any enhancements or changes to the event schema can affect both teams. The responsibility to make any changes is owned by both teams. Intentionally, these teams do not share anything beyond this mutually agreed upon models and artifacts. Here is the representation of shared kernel relationships in teams.

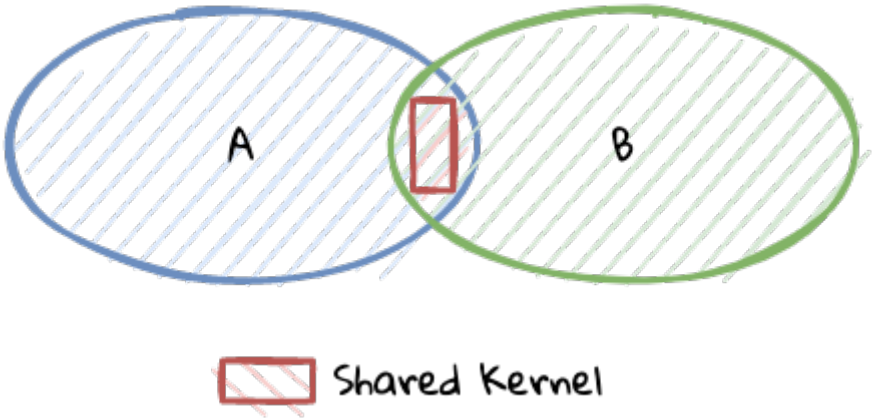


Figure 1- 85. Teams have an explicit understanding of shared models

When to use

The shared kernel form of collaboration works well if shared artifacts are required to be consumed in an identical fashion in both contexts. Furthermore, it is attractive for multiple teams to coordinate and continue sharing, as opposed to duplicating identical models in both contexts.

Potential pitfalls

Changes made to the shared kernel affect both bounded contexts. This means that any change made to the shared kernel needs to remain compatible for both teams. Needless to say, as the number of teams using the shared kernel increases, the cost of coordination goes up manifold.

Separate ways

When two teams choose to not share any artifacts or models between them, they go their own separate ways.

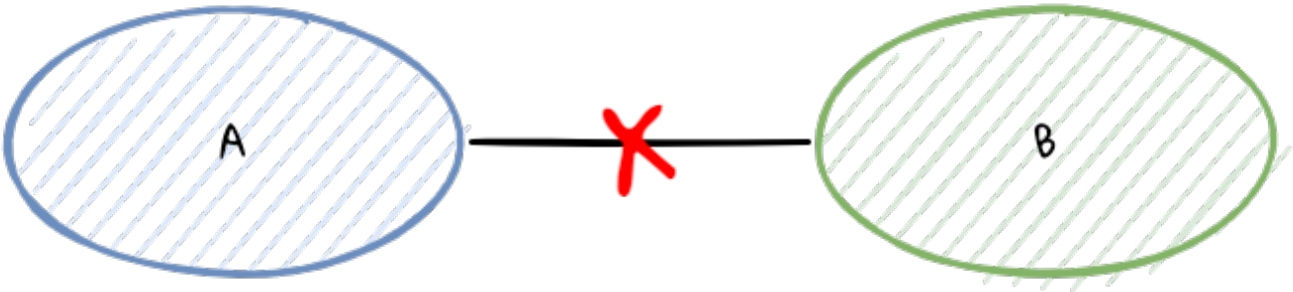


Figure 1- 86. Teams go separate ways and not share anything between them

Example

The *LC Application Processing* and the *Customer Onboarding* teams may start with sharing the same build/deployment scripts for their services. Over a period of time, deployment requirements may diverge to a point where the shared cost of maintaining these scripts becomes prohibitively expensive, causing these teams to fork their deployments to regain independence from the other team.

When to use

In some cases, two teams may be unable to collaborate for a variety of reasons, ranging from a drift in individual team requirements to organizational politics. Whatever the case may be, these teams may decide that the cost of collaboration is too high, resulting in them going their own separate ways.

Potential pitfalls

Choosing to go separate ways may result in duplicate work across affected bounded contexts. When working in bounded contexts that map to the core subdomains, this may prove counter-productive as it could lead to inconsistent behaviors unintentionally.

It is possible to transition from one relationship type to another over a period of time. In our experience, transitioning from any one of these relationships may not be straightforward. In cases where requirements are relatively clear at the outset, it may be easier to start with a *shared kernel*. On the contrary, if requirements are unclear, it may be prudent to start either with a loose *partnership* or go *separate ways* until requirements become clear. In any of these scenarios, it is important to keep evaluating the nature of the relationship and transition to a more appropriate type based on our enhanced understanding of the requirements and/or the relationship itself.

In each of the relationships characterized above, the teams involved have a more or less equal say

in how the relationship evolved and the resulting outcomes. However, this may not always be the case. Let's look at examples of cases where one team may have a clear upper hand in terms of how the relationship evolves.

9.2.2. Asymmetric relationship patterns

Two teams can be said to have an asymmetric relationship when one of the teams has a stronger influence in the decision-making process to arrive at a solution. In other words, there is a clear customer-supplier (or upstream-downstream) relationship where either the customer or the supplier plays a dominant role that affects solution design approaches. It is also likely that the customer and the supplier do not share common goals. Here is the representation of an asymmetric relationship between customer and supplier.

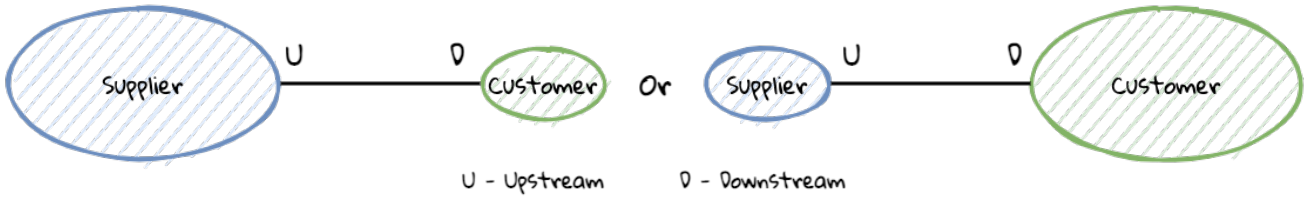


Figure 1- 87. One of the teams has a dominant say in influencing the solution

There are at least three solution patterns when teams are in an asymmetric relationship, each of which we outline in more detail in the following sub-sections.

Conformist (CF)

It is not unusual for the side playing the supplier role to have a dominant say in how the relationship with one or more customers is implemented. Furthermore, the customer may simply choose to conform with the supplier-provided solution as is, making it an integral part of their own solution. In other words, the supplier provides a set of models and the customer uses those same models to build their solution. In this case, the customer is termed to be a *conformist*.

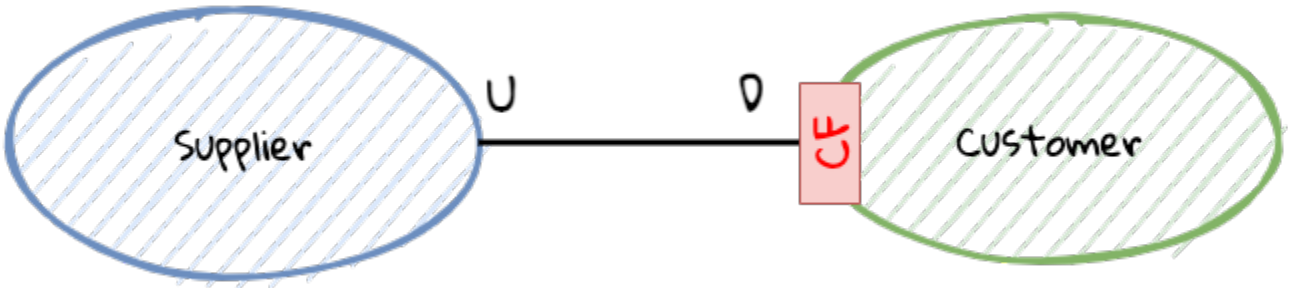


Figure 1- 88. Customer accepts dependency on supplier model

Example

When building a solution to validate United States postal addresses of LC applicants, we chose to conform to the USPS Web Tools address validation API schema. Given that the business started with just US-based applicants, this made sense. This means that any references to the address model in our bounded contexts mimic the schema prescribed by the USPS. This further means that we will need to keep up with changes that occur in the USPS API as and when they occur (regardless of whether that change is needed for our own functionality).

When to use

Being a conformist is not necessarily a negative thing. The supplier's models may be a well accepted industry standard, or they may simply be good enough for our needs. It may also be that the team may not have the necessary skills, motivation or immediate needs to do something different from what the supplier has provided. This approach also enables teams to make quick progress, leveraging work mostly done by other experts.

Potential pitfalls

An overuse of the conformist pattern may dilute the ubiquitous language of our own bounded contexts, resulting in a situation where there is no clear separation between the supplier and customer concepts. It may also be that concepts that are core to the supplier's context leaks into our own, despite those concepts carrying little to no meaning in our context. This may result in these bounded contexts being very tightly coupled with each other. And if a need arises to switch to another supplier or support multiple suppliers, the cost of change may be prohibitively expensive.

Anti-corruption layer (ACL)

There may be scenarios where a customer may need to collaborate with the supplier, but may want to shield itself from the supplier's ubiquitous language and models. In such cases, it may be prudent to redefine these conflicting models in the customer's own ubiquitous language using a translation layer at the time of integration, also known as an *anti-corruption layer* (ACL). This is depicted in the following figure :

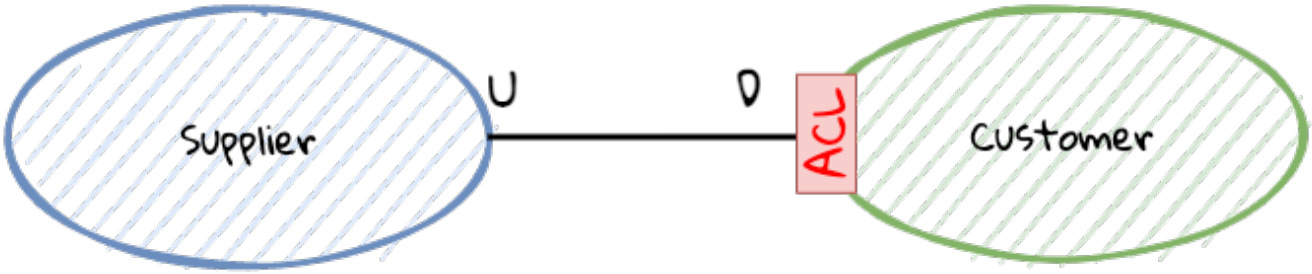


Figure 1- 89. Customer wants to protect itself from supplier models

Example

In the address validation example referenced in the Conformist section, the *LC Application Processing* team may need to support Canadian applicants as well. In such a case, being a conformist to a system that supports only US addresses may prove restrictive and even confusing. For example, the US *state* is analogous to a *province* in Canada. Similarly, *zip code* in the US is referred to *postal code* in Canada. In addition, US zip codes are numeric whereas Canadian postal codes are alphanumeric. Most importantly, we currently do not have the notion of a *country code* in our address model, but now we will need to introduce this concept to differentiate addresses within the respective countries. Let's look at the address models from the respective countries here:

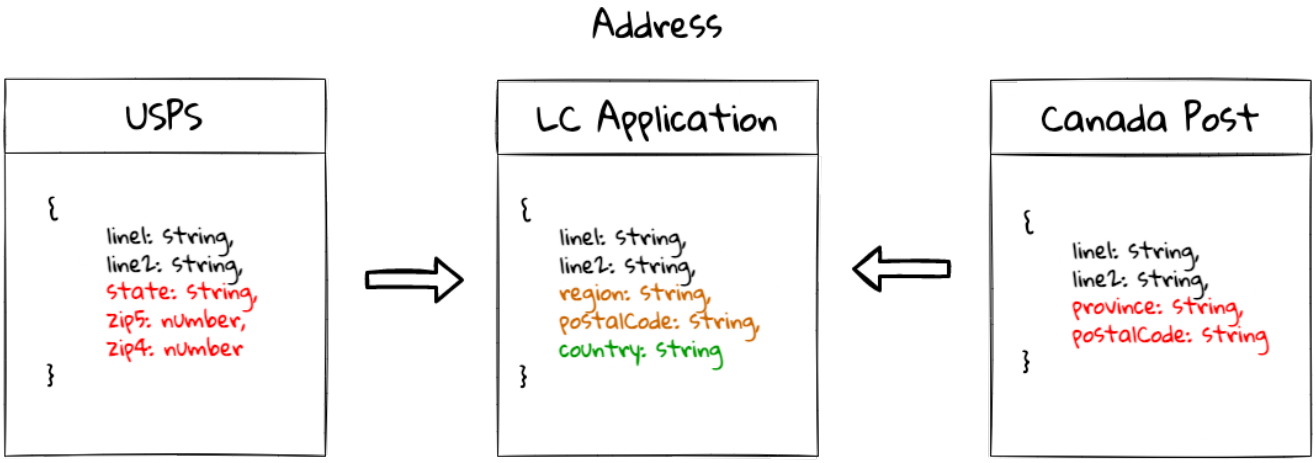


Figure 1- 90. Address Models of different countries

While we initially conformed to the USPS model, we have now evolved to support more countries. For example, *region* is used to represent the concept of *state/province*. Also, we have introduced the *country* value object, which was missing earlier.

When to use

Anti-corruption layers come in handy when the customer models are part of a core domain. The ACL shields the customer from changes in the supplier's models and can help produce more loosely coupled integrations. It may also be necessary when we are looking to integrate similar concepts from multiple suppliers.

Potential pitfalls

Using an anti-corruption layer may be tempting in a lot of cases. However, it is less beneficial when the concepts being integrated don't often change, or are defined by a well-known authority. Using an ACL with a custom language may only cause more confusion. Creating an ACL usually requires additional translations and thereby may increase the overall complexity of the customer's bounded context and may be considered premature optimization.

Open host service (OHS)

Unlike the conformist and the anti-corruption layer, where customers do not have a formal means to interface with the supplier, with the open host service, the supplier defines a clear interface to interact with its customers. This interface may be made available in the form of a well-known published language (for example, a REST interface or a client SDK):

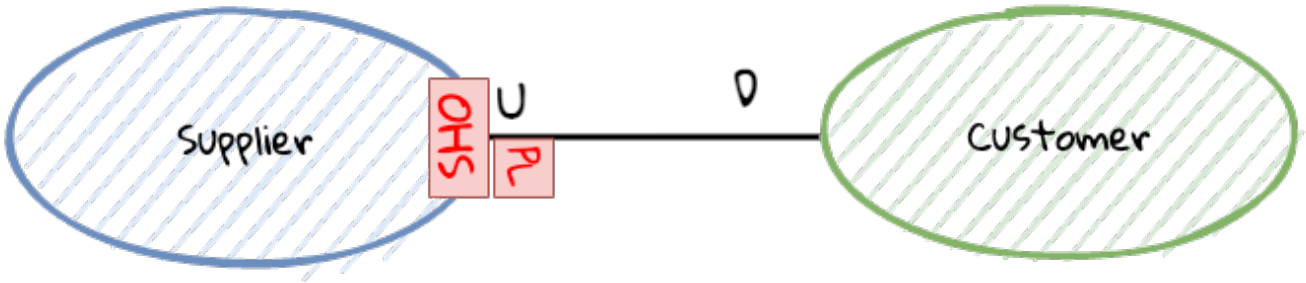


Figure 1- 91. Open host service (OHS) using a published language (PL).

Example

The LC Application Processing bounded context can expose an HTTP interface for each of its commands as shown here:

```
# Start a new LC application
curl POST /applications/start \
        -d '{"applicant-id": "8ed2d2fe", "clientReference": "Test LC"}' \
        -H 'content-type:application/vnd.lc-application.v2+json'

# Change the amount on an existing application
curl POST /applications/ac130002/change-amount \
        -d '{"amount": 100, "currency": "USD"}' \
        -H 'content-type:application/vnd.lc-application.v2+json'

# Other commands omitted for brevity
```

As an augment to the HTTP interface shown here, we may even provide a client SDK in some of the more popular languages used by our customers. This helps hide more implementation details such the MIME type and version from customers.

When to use

When the supplier wants to hide its internal models (ubiquitous language), making an open host service enables the supplier to evolve while providing a stable interface to its customers. In a sense, the open-host service pattern is a reversal of the anti-corruption layer pattern: instead of the customer, the supplier implements the translation of its internal model. Also, the supplier can consider providing an open host service when it is interested in providing a richer user experience for its customers.

Potential pitfalls

While suppliers may have good intentions by providing an open host service for its customers, it may result in increased implementation complexity (for example, there may be a need to support multiple versions of an API, or client SDKs in multiple languages). If the open host service does not take into account common usage patterns of its customers, it may result in a poor customer usability and also in degraded performance for the supplier.

It is important to note that the conformist and the anti-corruption layer are patterns that customers implement, whereas the open host service is a supplier-side pattern. For example, the following scenario with the supplier providing an *open host service* and one customer is a *conformist* while another has an *anti-corruption layer*, can be true as depicted here:

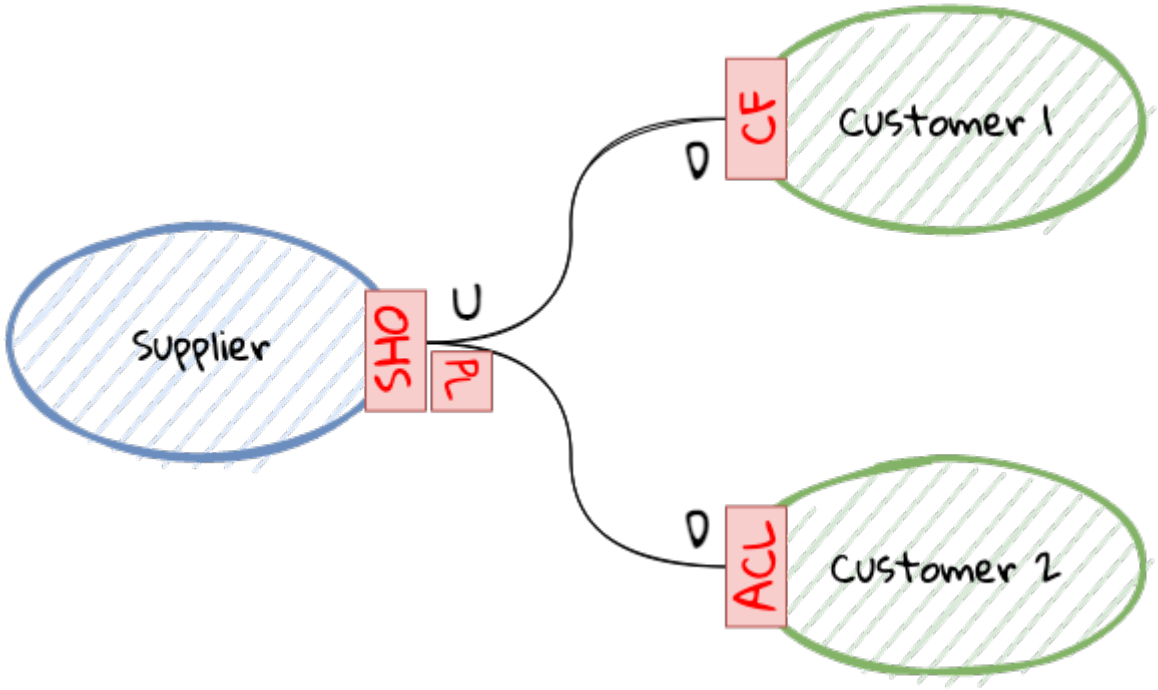


Figure 1- 92. Asymmetric relationships with multiple customers.

Now that we have seen the various ways in which bounded contexts can integrate with each other, here is one possible implementation for our LC application depicted in the form of a context map:

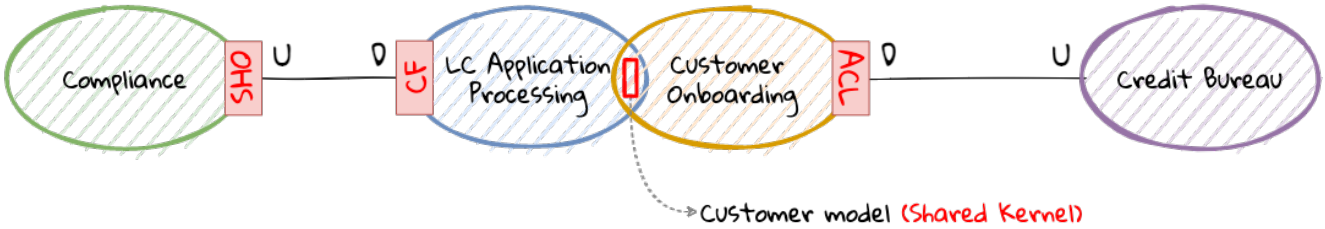


Figure 1- 93. Simplified context map for the LC application.

Thus far we have examined the various ways in which inter-team dynamics influence integration mechanisms. While having clarity at the conceptual level helps, let's see how these relationships manifest themselves at the implementation level.

9.3. Implementation patterns

We have looked integration between bounded contexts at a design level, but these concepts need to be translated into code. There are three broad categories that can be employed when integrating two bounded contexts:

1. Data-based
2. Code-based
3. API-based

Let's look at each method in more detail now.

9.3.1. Data-based

In this style of integration, the bounded contexts in question share data between each other. If the relationship is symmetric, the teams owning these bounded contexts may choose to share entire databases with free access to both read, write and change underlying structures. Whereas in an asymmetric relationship, the supplier may constrain the scope of access, based on the type of relationship.

Shared database

The simplest form of data integration is the use of a shared database. In this style of integration, all participating bounded contexts have unrestricted access to the schemas and the underlying data as shown here:

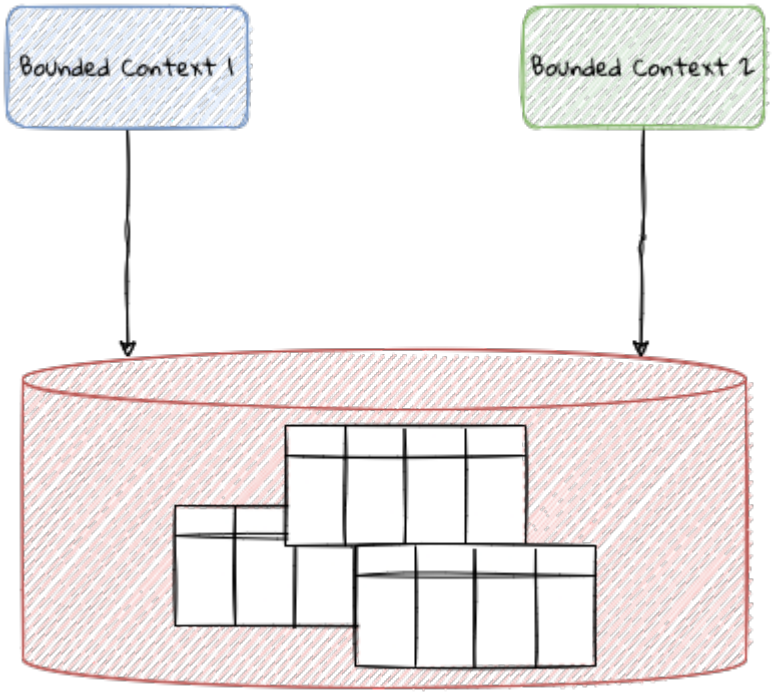


Figure 1- 94. Integration using a shared database

When to use

The shared database presents a very low barrier to entry for teams looking to quickly enable new or enhance existing functionality by providing ready access to data for read and/or write use-cases. More importantly, it also allows the use of local database transactions, which usually provides strong consistency, lower complexity and better performance (especially when working with relational databases).

Potential pitfalls

However, this symmetric integration style where multiple teams have shared ownership is usually frowned upon because it often leads to a situation where there is no clear ownership. Furthermore, the shared databases can become a source of tight coupling, accelerating the path towards the dreaded *big ball of mud*. Additionally, users of the shared database can suffer from the *noisy neighbor* effect where one co-tenant monopolizing resources adversely affects all other tenants. For these reasons, teams will be well advised to choose this style of integration sparingly.

Replicated data

In the case of asymmetric relationships, suppliers may be unwilling to provide direct access to their data. However, they may choose to integrate with customers using a mechanism based on data sharing. An alternate form of integration is to provide a copy of the data required by consumers. There are many variations on how this can be implemented, we depict the more common ways here:

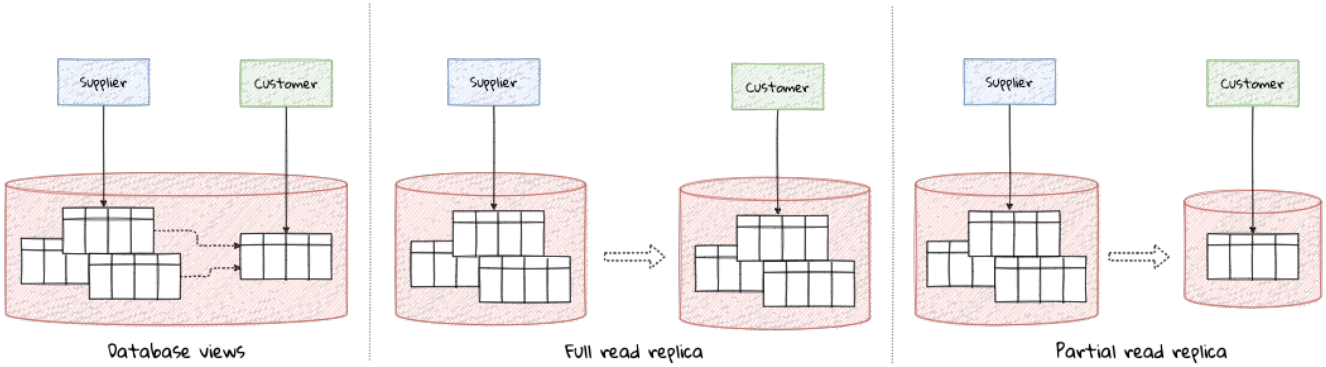


Figure 1- 95. Integration using data replication.

- **Database views**: In this form, the consumer gets or is provided access to a subset of the data using query-based or materialized views. In either case, the customer usually has read-only access to the data and both supplier and customer continue to share the same physical resources (usually the DB engine).
- **Full read replica**: In this form, the customer gets access to a read replica of the supplier's entire database, usually on physically disparate infrastructure.
- **Partial read replica**: In this form, the customer gets access to a read replica of a subset of the supplier's database, again on physically disparate infrastructure.

When to use

This style of integration may be required when there is an asymmetric relationship between the supplier and the customer. Like the shared database, this integration style usually requires less upfront effort to integrate. This is also apt when suppliers intend to provide read-only access to a subset of their data. It may also suffice to use data replication when customers only require to read a subset of the supplier's data.

Potential pitfalls

If we choose to use database views, we may continue to suffer from the noisy neighbor effect. On the other hand, if we choose to create physically disparate replicas, we will need to incur the cost of additional operational complexity. More importantly, the consumers remain tightly coupled to the supplier's domain models and ubiquitous language.

Next, let's look at some ways to make the most of data-based integrations.

Increasing effectiveness

When sharing data, the schema (the structure of the database) acts as a means to enforce contracts, especially when using databases that require specifying a formal structure (for example, relational

databases). When multiple parties are involved, managing the schema can become a challenge.

To mitigate undesirable changes, teams sharing data may want to consider the use of a schema migration tool. Relational databases work well with tools like liquibase[37] or flyway[38]. When working with databases that do not formally enforce a schema, it may be best to avoid employing this style of integration, especially when working in symmetric relationships where ownership is unclear.

In any case, if using one of the shared data styles of integration is unavoidable, teams may want to strongly consider employing one or more techniques mentioned in refactoring databases to make it more manageable.

9.3.2. Code-based

In this style of integration, teams coordinate by sharing code artifacts, either directly in the form of source code and/or binaries. At a high level, there are two forms:

1. Sharing source code
2. Sharing binaries

We describe each of these here:

Sharing source code

A fairly common practice within organizations is to share source code with the objective of promoting reuse and standardization. This may include utilities (like logging, authentication, etc.), build/deployment scripts, data transfer objects, etc. In other words, any piece of source code where the cost of duplication is seen to be higher than reuse.

When to use

Depending on the relationship type (symmetric/asymmetric), teams sharing code may have varied levels of influence in how the shared artifacts evolve. This works well in a symmetric relationship, both teams are empowered to make changes compatible with each other. Similarly, in an asymmetric relationship, the supplier may accept changes from customers, while retaining ownership and control of the shared artifacts. This also tends to work well in case of non-core, infrequently changing code artifacts. Sharing source code also enables higher levels of transparency and visibility into the internals of the shared artifacts (case in point - open source software).

Potential pitfalls

Sharing code artifacts means that individual teams take on responsibility to make sure that the process of converting source code into binary executables is uniform and compatible with requirements for all parties. This may include code conventions, static quality checks, tests (presence or lack thereof), compilation/build flags, versioning, and so on. When a relatively large number of teams are involved, maintaining this form of compatibility may become burdensome.

Sharing binary artifacts

Another relatively common practice is to share artifacts at the binary level. In this scenario, the consumers may or may not have direct access to source code artifacts. Examples include third-party libraries, client SDKs, API documentation, and so on. This form of integration is fairly common when the relationship between the coordinating parties is asymmetric. The supplier of the library has a clear ownership of maintaining the lifecycle of the shared artifacts.

When to use

Sharing just binary artifacts may be necessary when the supplier is unable/unwilling to share source artifacts, possibly because they may be proprietary and/or part of the supplier's intellectual property. Because the supplier takes ownership of the *build* process, it behooves the supplier to produce artifacts that are compatible with most potential consumers. Hence, this works well when the supplier is willing to do that. On the other hand, it requires that the customer place high levels of trust[39] in the supplier's software supply chain[40] when producing these artifacts.

Potential pitfalls

Integration through the use of binary artifact sharing reduces the visibility into the build process of the shared artifacts for the consumers. If consumers rely on slow-moving suppliers, this can become untenable. For example, if a critical security bug is discovered in the shared binary, the consumer is solely reliant on the supplier to remediate. This can be a huge risk if such dependencies are in critical, business-differentiating aspects of the solution (especially in the core subdomain). This risk can be exacerbated without the use of appropriate anti-corruption layers (ACLs) and/or service level agreements (SLAs).

Increasing effectiveness

When sharing code artifacts, it becomes a lot more important to be explicit in how changes are made while continuing to maintain high levels of quality — especially when multiple teams are involved. Let's examine some of these techniques in more detail:

1. **Static analysis**: This can be as simple as adhering to a set of coding standards using a tool like checkstyle. More importantly, these tools can be used to conform to a set of naming conventions to allow the firmer use of the ubiquitous language throughout the codebase. In addition, tools like spotbugs, PMD/CPD can be used to statically analyze code for the presence of bugs and duplicate code.
2. **Code architecture tests**: While static inspection tools are effective at operating at the level of a single compilation unit, runtime inspection can take this one level further to identify package cycles, dependency checks, inheritance trees, etc. to apply lightweight architecture governance. The use of tools like JDepend and ArchUnit can help here.
3. **Unit tests**: When working with shared codebases, team members are looking to make changes in a safe and reliable manner. The presence of a comprehensive suite of fast-running unit tests can go a long way to wards increasing confidence. We strongly recommend employing test-driven design to further maximize creating a codebase that is well-designed and one that enables easier refactoring.
4. **Code reviews**: While automation can go a long way, augmenting the process where a human

reviews changes can be highly effective for multiple reasons. This can take the form of offline reviews (using pull requests) or active peer reviews (using paired programming). All of these techniques serve to enhance collective understanding, thereby reducing risk when changes are made.

5. **Documentation**: Needless to say, well-structured documentation can be invaluable when making contributions and also when consuming binary code artifacts. Teams will be well advised to proliferate the use of the ubiquitous language by striving to write self-documenting code all throughout to maximize benefits derived.
6. **Dependency management**: When sharing binary code artifacts, managing dependencies can become fairly complicated due to having too many dependencies, long dependency chains, conflicting/cyclic dependencies, and so on. Teams should strive to reduce afferent (incoming) coupling as much as possible to mitigate the problems described above.
7. **Versioning**: In addition to minimizing the amount of afferent coupling, using an explicit versioning strategy can go a long way towards making dependency management easier. We strongly recommend considering the use of a technique like semantic versioning for shared code artifacts.

9.3.3. IPC-based

In this style of integration, the bounded contexts exchange messages using some form of inter-process communication (IPC) to interact with each other. This may take the form of synchronous or asynchronous communication.

Synchronous messaging

Synchronous messaging is a style of communication where the sender of the request waits for a response from the receiver, which implies that the sender and the receiver need to be active for this style to work. Usually, this form of communication is point-to-point. HTTP is one of the commonly used protocols for this style of communication. A visual representation of this form of communication is shown here:

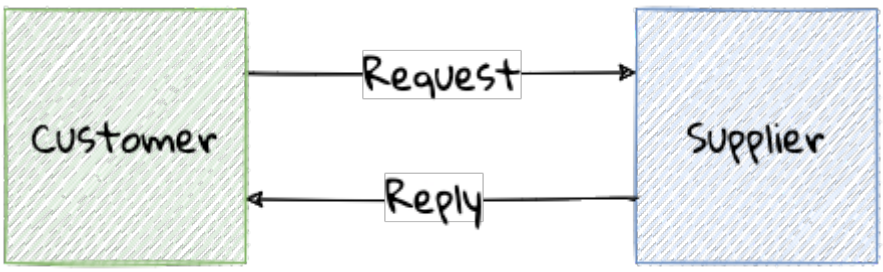


Figure 1- 96. Synchronous messaging

Please take a look at the HTTP APIs for the commands used during LC application processing included with the code examples for this chapter.

When to use

This form of integration is used when the customer is interested in the supplier's response to the request. The response is then used to determine whether the request was successful or not. Given

that the customer needs to wait for the response, it is advisable to use this style of messaging for low latency operations. This form of integration is popular when exposing public APIs over the internet (for example, Github's REST API[41]).

Potential pitfalls

When using synchronous messaging, the customer's ability to scale is heavily dependent on the supplier to satisfy the customer's requirements. On the flip side, customers making requests at too high a rate may compromise the supplier's ability to serve customers in a predictable manner. If there is a chain of synchronous messaging, the probability of cascading failure becomes much higher.

Asynchronous messaging

Asynchronous messaging is a style of communication where the sender does not wait for an explicit response from the receiver.

We are using the terms sender and receiver, instead of customer and supplier because they both can play either role of sender or receiver.

This is typically achieved by introducing an intermediary in the form of a message channel. The presence of the intermediary enables both one-to-one and one-to-many modes of communication. Typically, the intermediary can take the form of a shared filesystem, database or a queueing system.

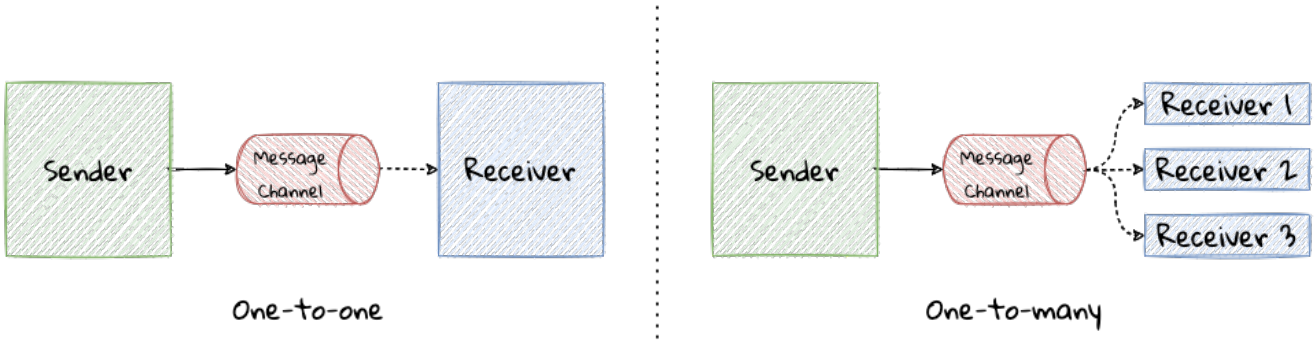


Figure 1- 97. Asynchronous messaging

Please take a look at the event APIs for the commands used during LC application processing included with the code examples for this chapter.

When to use

This form of integration is used when the sender does not care about receiving an **immediate** response from the receiver(s), resulting in the respective systems becoming a lot more decoupled from each other. This further enables these systems to scale independently. This also makes it possible to have the same message to be processed by multiple receivers. For example, in our LC application processing system, the `LCApplicationSubmittedEvent` is received by both the *Compliance* and *Customer Onboarding* systems.

Potential pitfalls

The introduction of the intermediary component adds complexity to the overall solution. The non-functional characteristics of the intermediary can have a profound effect on the resilience characteristics of the system as a whole. It can also be tempting to add processing logic to the intermediary, thereby coupling the overall system very tightly to this component. To ensure reliable communication between the sender and the receiver, the intermediary may have to support a variety of enhanced capabilities (such as ordering, producer flow control, durability, transactions, and so on.)

Increasing effectiveness

When implementing integration using some form of IPC, a lot of the techniques discussed in the code-based implementation patterns section continue to apply. As discussed earlier, API documentation plays a significant role in reducing friction for customers. In addition, here are a few more techniques that apply specifically when using IPC-based integration. We outline some of these techniques here:

1. **Typed protocols**: When working with this form of integration, it is important to minimize the amount of time taken to gather feedback on structural validations. This is especially important given that the supplier and the customer may be in a constant state of independent evolution. The use of typed protocols such as protocol buffers, Avro, Netflix's Falcor, GraphQL, and so on. can make it easier for customers to interact with suppliers while maintaining a lightweight mechanism to validate correctness of requests.

> The operative word here is **lightweight**. It is pertinent to note that we are not advising against the use of JSON-based HTTP APIs (typically advertised as being RESTful) which do not enforce the use of an explicit schema. Neither are we promoting the use of (arguably) legacy protocols like SOAP, WSDL, CORBA, etc. Each of these, while being well-meaning suffered from being fairly heavyweight.

2. **Self discovery**: As outlined above, when working with a IPC-based integration mechanism, we should look to reduce the barrier to entry. When working with RESTful APIs, the use of HATEOAS[42], although difficult for suppliers to implement, can make it easier for customers to understand and consume APIs. In addition, making use of a service registry and/or a schema registry can further reduce consumption friction.
3. **Contract tests**: In the spirit of failing fast and shifting left, the practice of contract testing and consumer-driven contracts can further increase the quality and speed of integration. Tools such as Pact[43] and Spring Cloud Contract[44] make the adoption of these practices relatively simple.

Thus far, we discussed implementation patterns, broadly categorized into data-based, code-based and IPC-based integrations. Hopefully, this gives you a good idea to consciously choose the appropriate approach considering the benefits and the caveats that they bring along with them.

9.4. Summary

In this chapter, we looked at the different types of bounded context relationships. We also examined common integration patterns that can be used when implementing these bounded

context relationships.

You have learned when specific techniques can be used, potential pitfalls and ideas on how to increase effectiveness when employing these methods.

In the next chapter, we will explore means to distribute these bounded contexts into independently deployable components (in other words, employ a microservices-based architecture).

9.5. Questions

1. In your ecosystem, can you identify bounded context boundaries and the integration mechanism in use?
2. Will you be able to draw a context map for your system?
3. Will you recommend any changes to the existing styles of integration based on the learnings from this chapter?

9.6. Further reading

Title	Author	Location
Integration database	Martin Fowler	https://martinfowler.com/bliki/IntegrationDatabase.html
REST APIs must be hypertext-driven	Roy T. Fielding	https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[37] https://www.liquibase.org/

[38] https://flywaydb.org/

[39] https://www.thoughtworks.com/en-us/insights/podcasts/technology-podcasts/securing-software-supply-chain

[40] https://blog.sonatype.com/software-supply-chain-a-definition-and-introductory-guide

[41] https://docs.github.com/en/rest

[42] https://restfulapi.net/hateoas

[43] https://pact.io/

[44] https://spring.io/projects/spring-cloud-contract

Part 3: Advanced Patterns

In Part 3, we will extend the application we built in Part 2 to utilize more modern, cloud native technologies. We will look at implementing an ecosystem of microservices and further extend these to be expressed to employ a serverless architecture.

Chapter 10. Distributing into remote components

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> — Leslie Lamport

Thus far, we have a working application for LC application processing, which is bundled along with other components as a single package. Although, we have discussed the idea of subdomains and bounded contexts, the separation between these components is logical, rather than physical. Furthermore, we have focused primarily on the *LC Application Processing* aspect of the overall solution. In this chapter, we will look at extracting the LC Application Processing bounded context into components that are physically disparate, and hence enable us to deploy them independently of the rest of the solution. We will discuss various options available to us, the rationale for choosing a given option, along with the implications that we will need to be cognizant of.

10.1. Continuing our design journey

From a logical perspective, our realization of the Letter of Credit application looks like the visual depicted here:

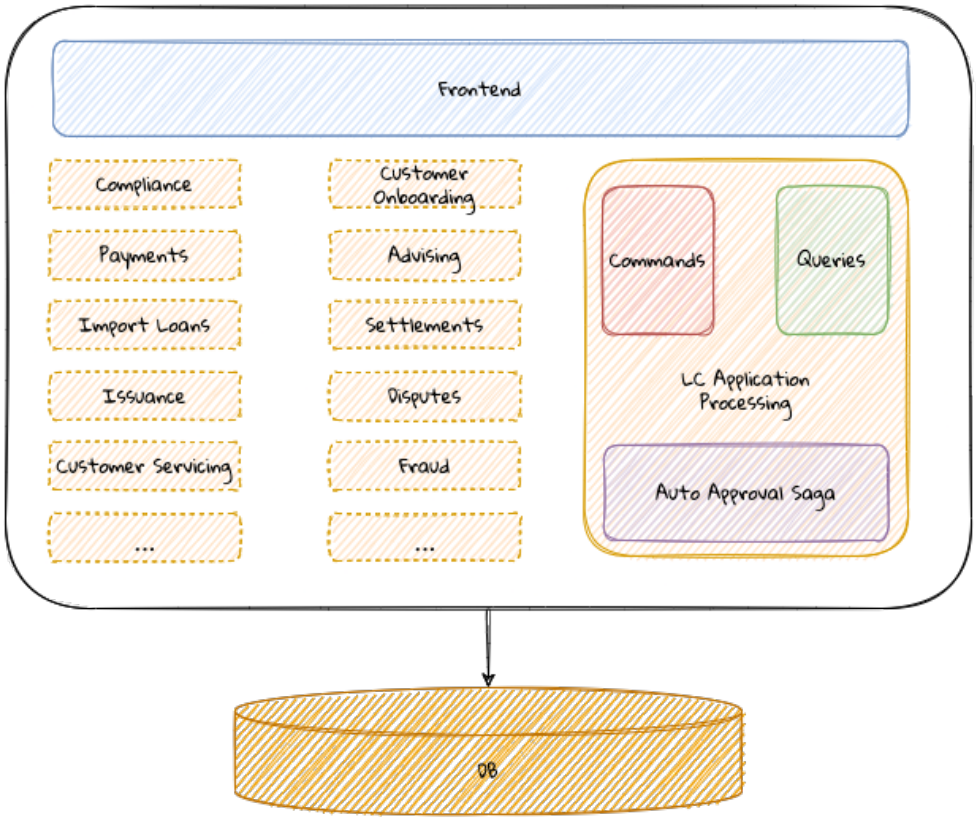


Figure 1- 98. Current view of the LC application monolith

Although the *LC Application Processing* component is loosely coupled from the rest of the application, we are still required to coordinate with several other teams to realize business value.

This may inhibit our ability to innovate at a pace faster than the slowest contributor in the ecosystem. This is because all teams need to be production ready before a deployment can happen. This can be further exacerbated by the fact that individual teams may be at different levels of engineering maturity. Let's look at some options on how we can achieve a level of independence from the rest of the ecosystem by physically decomposing our components into distinctly deployable artifacts.

10.2. Decomposing our monolith

First and foremost, the LC Application Processing component exposes only in-process APIs when other components interact with it. This includes interactions with:

1. Frontend
2. Published/consumed events
3. Database

To extract LC application processing functionality out into its own independently deployable component, remotely invokable interfaces will have to be supported instead of the in-process ones we have currently. Let's examine remote API options for each:

10.2.1. Changes for frontend interactions

Currently, the JavaFX frontend interacts with the rest of the application by making request-response style in-process method calls (`CommandGateway` for commands and `QueryGateway` for queries) as shown here:

```
@Service
public class BackendService {

    private final QueryGateway queryGateway;
    private final CommandGateway commandGateway;

    public BackendService(QueryGateway queryGateway,
                          CommandGateway gateway) {
        this.queryGateway = queryGateway;
        this.commandGateway = gateway;
    }

    public LCApplicationId startNewLC(ApplicantId applicantId, String clientReference)
{
        return commandGateway.sendAndWait(
                startApplication(applicantId, clientReference));
    }

    public List<LCView> findMyDraftLCs(ApplicantId applicantId) {
        return queryGateway.query(
                new MyDraftLCsQuery(applicantId),
                        ResponseTypes.multipleInstancesOf(LCView.class))
                .join();

    }
}
```

One very simple way to replace these in-process calls will be to introduce some form of remote procedure call (RPC). Now our application looks like this:

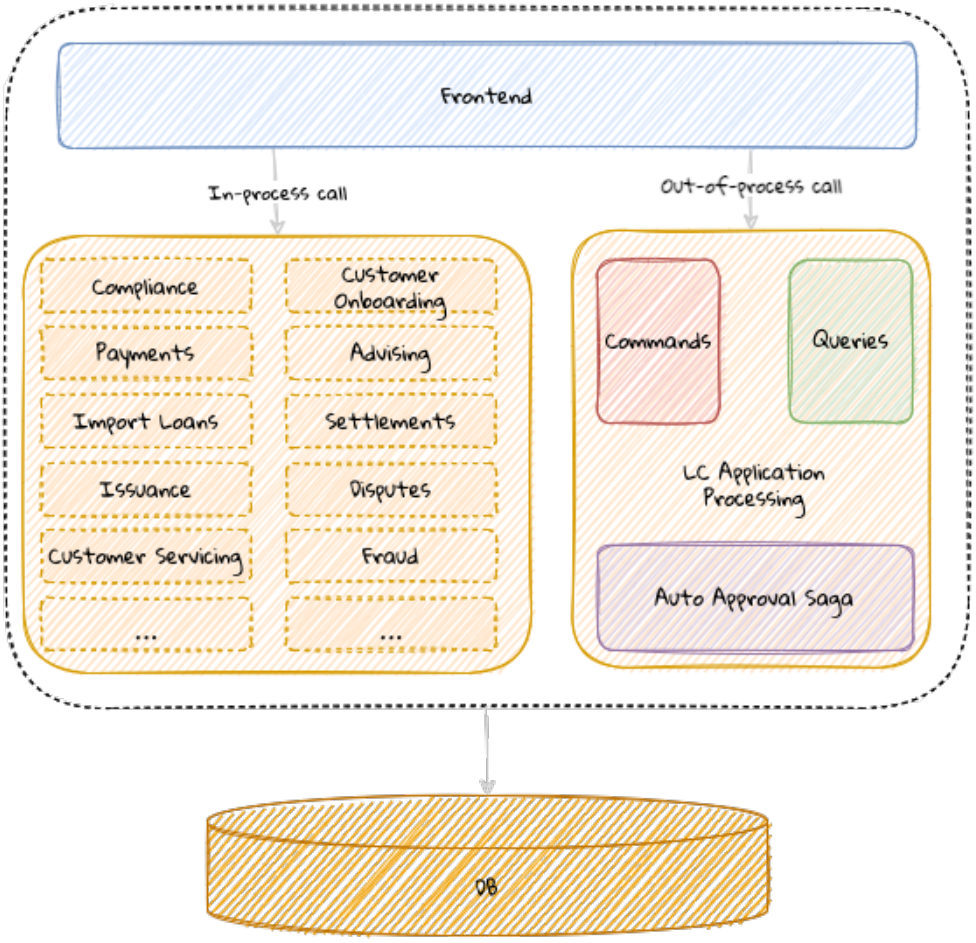


Figure 1- 99. Remote interaction with the frontend introduced

When working with in-process interactions, we are simply invoking methods on objects within the confines of the same process. However, when we switch to using out-of-process calls, we have quite a few considerations. These days when working with remote APIs, we have several popular choices in the form of JSON-based web services, GraphQL, gRPC, etc. While it is possible to make use of a completely custom format to facilitate the communication, DDD advocates the use of the open host service pattern using a published language that we covered in chapter 9. Even with the open host service style of communication, there are a few considerations, some of which we discuss here:

Protocol options

There are several options available to us when exposing remote APIs. These days using a JSON-based API (often labeled as REST) seems to be quite popular. However, this isn't the only option available to us. In a resource-based approach, the first step is to identify a resource (noun) and then map the interactions (verbs) associated with the resource as a next step. In an action-based approach, the focus is on the actions to be performed. Arguably, REST takes a resource-based approach, whereas graphQL, gRPC, SOAP, etc. seem to be action-based. Let's take an example of an API where we want to submit an LC. In a RESTful world, this may look something like below:

```
# Start a new LC application
curl POST /lc-applications/start \
        -d '{"applicant-id": "8ed2d2fe", "clientReference": "Test LC"}' \
        -H 'content-type:application/vnd.lc-application.v2+json'
```

whereas with a graphQL implementation, this may look like:

```
mutation StartLCApplication {
  startLCApplication(applicantId: "8ed2d2fe",
                     clientReference: "Test LC") {
    lcApplicationId
  }
}
```

In our experience, designing APIs using REST does result in some form of dilution when attempting to mirror the language of the domain — because the focus is first and foremost on resources. Purists will be quick to point out that the example above is not RESTful because there is no resource named `start`. Our approach is to place more importance to remaining true to the ubiquitous language as opposed to being dogmatic about adherence to technical purity.

Transport format

Here we have two broad choices: text-based (for example, JSON or XML) versus binary (for example, protocol buffers or avro). If non-functional requirements (like performance) are met, our preference is to use text-based protocols as a starting point, because it can afford the flexibility of not needing any additional tools to visually interpret the data (when debugging).

When designing a remote API, we have the option of choosing a format that enforces a schema (for example, protocol buffers or avro) or something less formal like plain JSON. In such cases, in order to stay true to the ubiquitous language, the process may have to include additional governance in the form of more formal design and code reviews, documentation, etc.

Compatibility and versioning

As requirements evolve, there will be a need to enhance the interfaces to reflect these changes. This will mean that our ubiquitous language will also change over time, rendering old concepts to become obsolete. The general principle is to maintain backwards compatibility with consumers for as long as possible. But this does come with a cost of having to maintain old and new concepts together — leading to a situation where it can become hard to tell what is relevant versus what is not. Using an explicit versioning strategy can help in managing this complexity to an extent — where newer versions may be able to break backwards compatibility with older ones. But it is also not feasible to continue supporting a large number of incompatible versions indefinitely. Hence, it is important to make sure that the versioning strategy makes deprecation and retirement agreements explicit.

REST APIs

We recognize that there are several options when exposing web-based APIs, claims of using a REST (Representation State Transfer) approach seem quite common these days. REST was coined by Roy Fielding as part of his doctoral dissertation. The idea of what constitutes REST has been a matter of debate and arguably remains ambiguous even today. Leonard Richardson introduced the notion of a maturity model for HTTP-based REST APIs that somewhat helped provide some clarity. The model describes broad conformance to REST in three levels, with each level being more mature than the

preceding one:

1. **Resources**: TODO
2. **HTTP Verbs**: TODO
3. **Hypermedia controls**: TODO

Most web service based solutions that claim to be RESTful seem to stop at level 2. Roy Fielding, the inventor or REST seems to claim that REST APIs must be hypertext-driven[45]. In our opinion, the use of hypertext controls in APIs allows them to become self-documenting and thereby promotes the use of the ubiquitous language more explicitly. More importantly, it also indicates what operations are applicable for a given resource at that time in its lifecycle. For example, let's look at a sample response where all pending LC applications are listed:

```
GET /lc-applications?status=pending HTTP/1.1
Content-Type: application/json

HTTP/1.1 200 OK
Content-Type: application/prs.hal-forms+json
{
  "_embedded" : {
    "lc-applications" : [
      {
       "clientReference" : "Test LC",
       "_links" : {
         "self" : {
           "href" : "/lc-applications/582fe5f8"
         },
         "submit" : {
           "href" : "/lc-applications/582fe5f8/submit"
         }
       }
      },
      {
       "clientReference" : "Another LC",
       "_links" : {
         "self" : {
           "href" : "/lc-applications/7689da3e"
         },
         "approve" : {
           "href" : "/lc-applications/7689da3e/approve"
         },
         "reject" : {
           "href" : "/lc-applications/7689da3e/reject"
         }
       }
      }
    ]
  }
}
```

In the example above, the first LC application needs to be **submit**ted, whereas the second application needs to either be **approved** or **rejected** (presumably because it has already been submitted). Notice how the response also does not need to include a `status` attribute so that they can use this to deduce which operations are relevant for LC application at that time. While this may be a subtle nuance, we felt that it is valuable to point out in the context of our DDD journey.

We have looked at a few considerations when moving from an in-process out-of-process API. There are quite a few other considerations, specifically pertaining to non-functional requirements (such as performance, resilience, error handling, etc.) We will look at these in more detail in Chapter 11.

Now that we have a handle on how we can work with APIs that interact with the front-end, let's look at how we can handle event publication and consumption **remotely**.

10.2.2. Changes for event interactions

Currently, our application publishes and consumes domain events over an in-process bus that the Axon framework makes available.

We publish events when processing commands:

Event publishing when processing a command

```
class LCApplication {

    // Boilerplate code omitted for brevity
    @CommandHandler
    public LCApplication(StartNewLCApplicationCommand command) {
        //...
        AggregateLifecycle.apply(new LCApplicationStartedEvent(command.getId(),
                command.getApplicantId(), command.getClientReference(), LCState.DRAFT
));
    }
}
```

and consume events to expose query APIs:

Event consumption to populate the query store

```
class LCApplicationSummaryEventHandler {

    // Boilerplate code omitted for brevity

    @EventHandler
    public void on(LCApplicationStartedEvent event) {
        //...
    }
}
```

In order to process events remotely, we need to introduce an explicit infrastructure component in the form of an event bus. Common options include message brokers like ActiveMQ, RabbitMQ or a distributed event streaming platform like Apache Kafka. Application components can continue to publish and consume events as before — only now they will happen using an out-of-process invocation style. Logically, this causes our application to now look something like this:

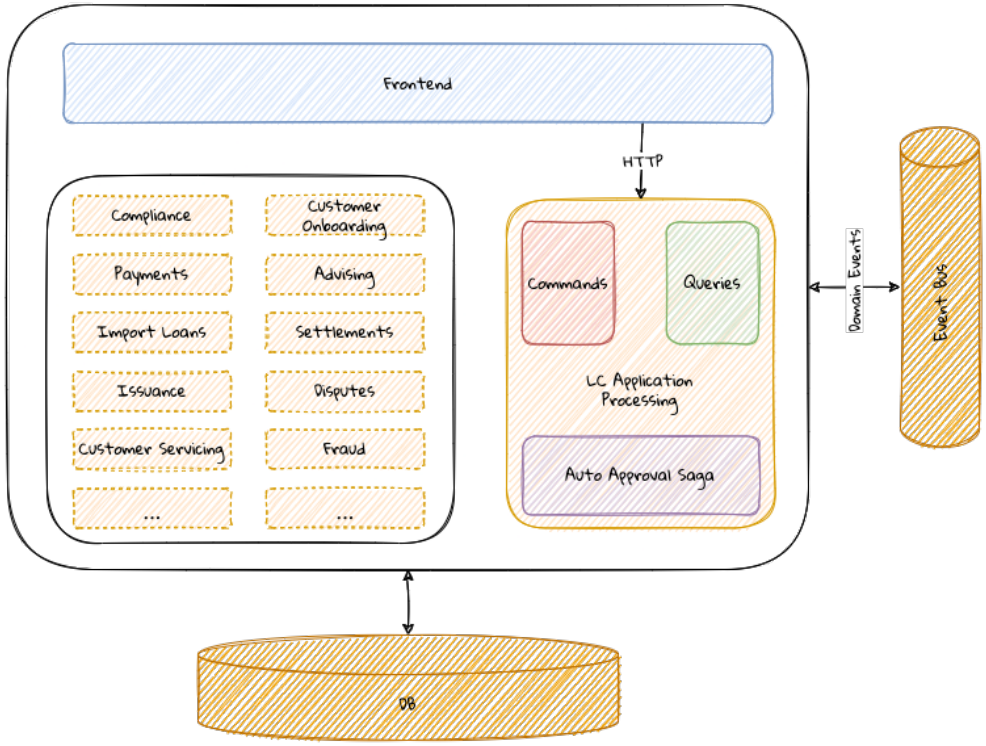


Figure 1- 100. Out of process event bus introduced

When working with events within the confines of a single process, assuming synchronous processing (event publishing and consumption on the same thread), we do not encounter a majority of problems that only become apparent when the publisher and the consumer are distributed across multiple processes. Let's examine some of these in more detail next.

Atomicity guarantees

Previously, when the publisher processed a command by publishing an event and the consumer(s) handled it, transaction processing occurred as a single atomic unit as shown here:

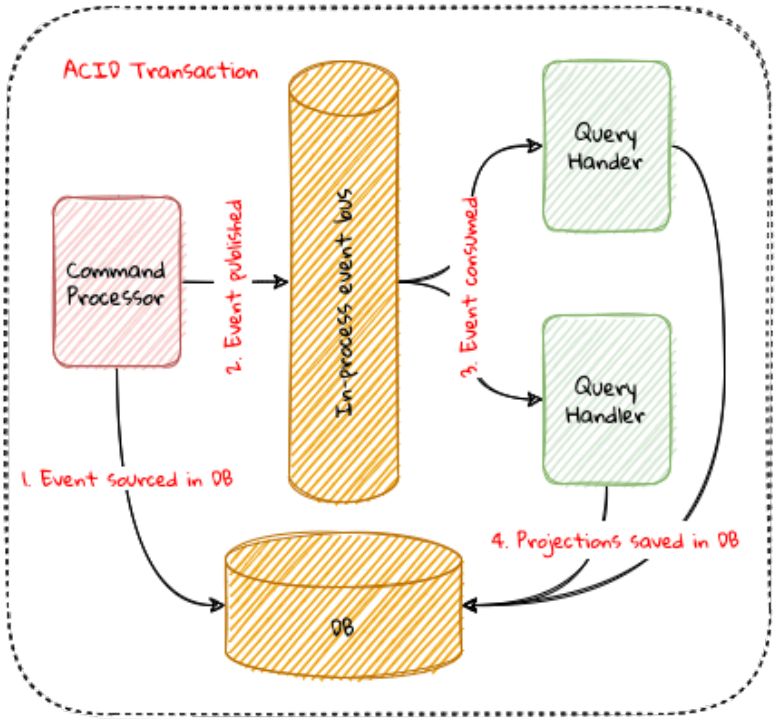


Figure 1- 101. ACID transaction processing within the monolith

Notice how all the highlighted operations in the diagram above happen as part of a single database transaction. This allowed the system to be strongly consistent end-to-end. When the event bus is distributed to work within its own process, atomicity cannot be guaranteed like it was previously. Each of the above numbered operations work as independent transactions. This means that they can fail independently, which can lead to data inconsistencies.

To solve this problem, let's look at each step in the process in more detail, starting with command processing as shown here:

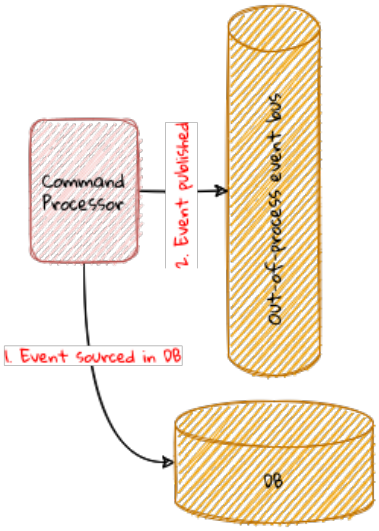


Figure 1- 102. Command processing transaction semantics

Consider the situation where we save to the database, but fail to publish the event. Consumers will remain oblivious of the event occurring and become inconsistent. On the flip side, if we publish the event but fail to save in the database, the command processing side itself becomes inconsistent. Not to mention that the query side now thinks that a domain event occurred, when in fact it did not.

Again, this leads to inconsistency. This **dual-write** problem is fairly common in distributed event-driven applications. If command processing has to work in a foolproof manner, saving to the database and the publishing to the event bus have to happen atomically - both operations should succeed or fail in unison. Here are a few solutions that we have used to solve this issue (in increasing order of complexity):

1. **Do nothing**: It is arguable that this approach is not really a solution, however it may be the only placeholder until a more robust solution is in place. While it may be puzzling to see this being listed as an option, we have seen several occasions where this is indeed how event-driven systems have been implemented. We leave this here as a word of caution so that teams become cognizant of the pitfalls.
2. **Transaction synchronization**: In this approach, multiple resource managers are synchronized in a way that failure in any one system, triggers a cleanup in the others where the transaction has already been committed. It is pertinent to note that this may not be foolproof in that it may lead to cascading failures.

The spring framework provides support for this style of behavior through the `TransactionSynchronization` and the now deprecated `ChainedTransactionManager` interfaces. Please refer to the framework documentation for more details. Needless to say, this interface should not be used without careful consideration to business requirements.

3. **Distributed transactions**: Another approach is to make use of distributed transactions and two-phase commit[46]. Typically, this functionality is implemented using pessimistic locking on the underlying resource managers (databases) and may present scaling challenges in highly concurrent environments.
4. **Transactional outbox**: All the above methods are not completely foolproof in the sense that there still exists a window of opportunity where the database and the event bus can become inconsistent (this is true even with two-phase commits). One way to circumvent this problem is by completely eliminating the dual-write problem. In this solution, the command processor writes to its database and the intended event to an *outbox* table in a local transaction. A separate poller component polls the outbox table and writes to the event bus. Polling can be computationally intensive and may again lead back to the dual write problem because the poller has to keep track of the last written event. This may be avoided by making event processing idempotent on the consumer so that processing duplicate events do not cause issues. In extremely high Another way to mitigate this issue is to use a change data capture (CDC) tool (like Debezium[47]). Most modern databases ship with tools to make this easier and may be worth exploring. Here is one way to implement this using the *transactional outbox pattern* as shown here:

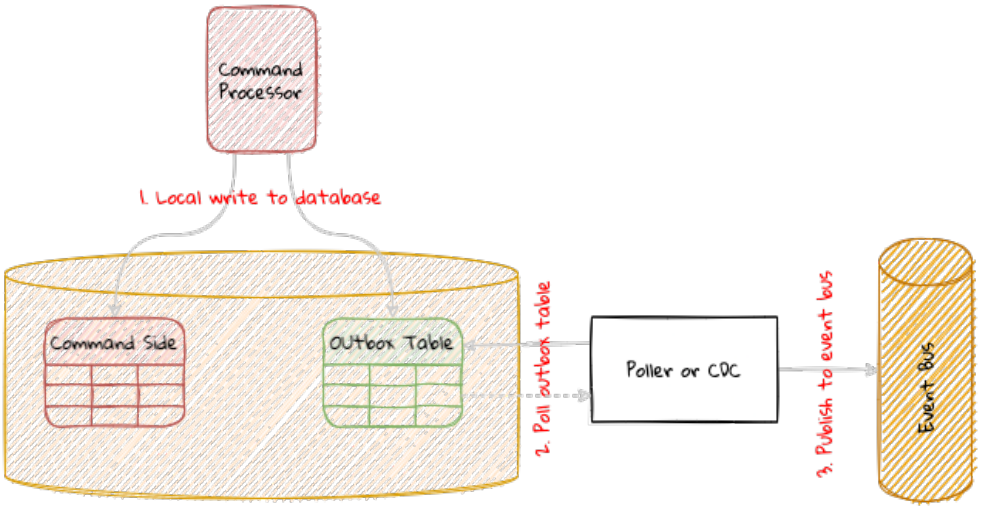


Figure 1- 103. Transactional outbox

The transactional outbox is a robust approach to dealing with the dual write problem. But it also introduces a non-trivial amount of operational complexity. In one of our previous implementations, we made use of the transactional synchronization to ensure that we never missed writes to the database. We also ensured that the event bus was highly available through redundancy on both the compute and storage tiers, and most importantly by avoiding **any** business logic on the event bus.

Delivery guarantees

Previously, because all of our components worked within a single process, delivery of events to the consumers was guaranteed at least as long as the process stayed alive. Even if event processing failed on the consumer side, it was fairly straightforward to detect the failure because exception handling was fairly straightforward. Furthermore, rollbacks were straightforward because the production and consumption of events happened as part of a single database transaction. With the LC processing application now becoming a remote component, event delivery becomes a lot more challenging. When it comes to message delivery semantics, there are three basic categories:

1. **At-most once** delivery means that each message may be delivered once or not at all. This style of delivery is arguably the easiest to implement because the producer creates messages in a fire and forget fashion. This may be okay in environments where loss of some messages may be tolerated. For example, data from click-stream analytics or logging might fall in this category.
2. **At-least once** delivery means that each message may be delivered more than once with no messages being lost. Undelivered messages are retried to be delivered — potentially infinitely. This style of delivery may be required when it is not feasible to lose messages, but where it may be tolerable to process the same message more than once. For example, analytical environments may tolerate duplicate message delivery or have duplicate detection logic to discard already processed messages.
3. **Exactly once** delivery means that each message is delivered exactly once without either being lost or duplicated. This style of message delivery is extremely hard to implement and a lot of solutions may approach exactly once semantics with some implementation help from the consumers where duplicate messages are detected and discarded with the producer sticking to at least once delivery semantics.

For the purposes of domain event processing, most teams will obviously prefer to have exactly once

processing semantics, given that they would not want to lose any of these events. However, given the practical difficulties guaranteeing *exactly once* semantics, it is not unusual to approach exactly once processing by having the consumers process events in an idempotent manner or designing events to make it easier to detect errors. For example, consider a `MonetaryAmountWithdrawn` event, which includes the `accountId` and the `withdrawalAmount`. This event may carry an additional `currentBalance` attribute so that the consumer may know if they are out of sync with the producer when processing the withdrawal. Another way to do this might be for the consumer to keep track of the last "n" events processed. When processing an event, the consumer can check if this event has already been processed. If so, they can detect it as a duplicate and simply discard it. All the above methods again add a level of complexity to the overall system. Despite all these safeguards, consumers may still find themselves out of sync with the system of record (the command side that produces the event). If so, as a last resort, it may be required to use a partial or full event replays which was discussed in Chapter 7.

Ordering guarantees

In an event-driven system like the one we are building, it is desirable for consumers to receive events in a deterministic order. Not knowing the order or receiving it in the wrong order may result in inaccurate outcomes. Consider the example of an `LCApplicationAddressChangedEvent` occurring twice in *quick succession*. If these changes are processed in the wrong order, we may end up displaying the wrong address as their current one. This does not necessarily mean that events need to be ordered for all use cases. Consider another example where we receive an `LCApplicationSubmittedEvent` more than once erroneously when it is not possible to submit a given LC application more than once. All such notifications after the first may be ignored.

As a consumer it is important to know if events will be ordered or not, so that we can make design considerations for out-of-order events. One default might be to accommodate for out-of-order events as a default. In our experience, this does tend to make the resulting design more complicated, especially in cases where the order does matter. We discuss three event ordering strategies and their implications for both the producer and the consumer here:

Strategy	Producer	Event Bus	Consumer
No ordering	Arguably the easiest to implement because there is no expectation from the producer to support ordering.	Without additional metadata, the event bus may only be able to guarantee ordering in the sequence of receipt (FIFO order).	If the consumer depends on ordering, it may have to implement ordering through some form of special processing.
Per aggregate ordering	The producer needs to make sure that each event includes an identifier to enable grouping by aggregate.	The event bus needs to support the notion of grouping (in this case by the aggregate identifier). For events belonging to the same aggregate instance, messages are emitted in FIFO order.	To guarantee ordering, events originating from the same aggregate instance need to be processed by the same consumer instance.

Strategy	Producer	Event Bus	Consumer
Global ordering	The producer needs to make sure that each event includes a sequence number.	Either the event bus or the consumer need to implement ordering logic.	

Durability and persistence guarantees

When an event is published to the event bus, the happy path scenario is that the intended consumer(s) are able to process it successfully. However, there are scenarios that may cause message processing to be impacted adversely. Let's examine each of these scenarios:

- **Slow consumer**: The consumer is unable to process events as fast as the producers are publishing them.
- **Offline consumer**: The consumer is unavailable (down) at the time of the events being published.
- **Failing consumer**: The consumer is experiencing errors when trying to process events.

In each of these cases, there can develop a backlog of unprocessed events. Because these are domain events, we need to prevent the loss of these events until the consumer has been able to process them successfully. There are two communication characteristics that need to be true for this to work successfully:

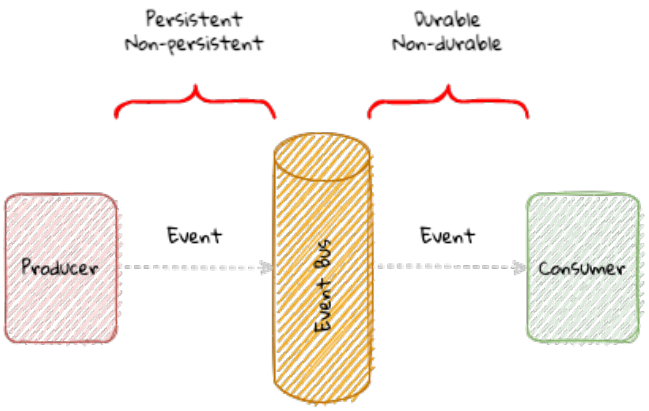


Figure 1- 104. Persistence versus durability

1. **Persistence**: the communication style between the publisher and the event bus.
2. **Durability**: the communication style between the event bus and the consumer.

Firstly, messages need to be persistent (stored on disk) and secondly the message subscription (relationship between the consumer and the event bus) needs to be durable (persist across event bus restarts). It is important to note that events have to be made persistent by the producer for them to be consumed durably by the consumer.

Processing guarantees

When an event is processed by the query side component as shown here, the following steps occur:

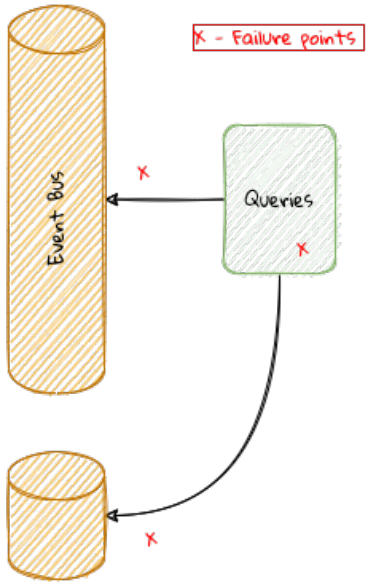


Figure 1- 105. Event processing failure scenarios

1. The event is consumed (either through a push or a pull) from the event bus
2. Transformation logic is applied on the payload o the event
3. The transformed payload is saved in the query side store.

Each of these steps can encounter failures. Irrespective of the cause of failure, the event should be durable (as discussed above) so that it can be processed later when the issue is fixed. These errors can be broadly segregated into four categories:

Error Cause	Example	Remediation
Transient	Network blip resulting in temporary connectivity issues to either the event bus or the query store.	A finite number of retries potentially with a backoff strategy before giving up with either a fatal error.
Configuration	Event bus or database URL misconfiguration.	Manual intervention with updated configuration and/or restart.
Code Logic	Implementation bugs either in the transformation logic.	Manual intervention with updated logic and redeployment
Data	Unexpected or erroneous data in the event payload.	Manual intervention that requires segregating spurious data (for example, by automatically moving problematic events to a *dead letter queue*) and/or fixing code logic.

We have now looked at the changes that we need to make because of the introduction of an out-of-process event bus.Having done this allows us to actually extract the LC application processing component into its own independently deployable unit, which will look something like this:

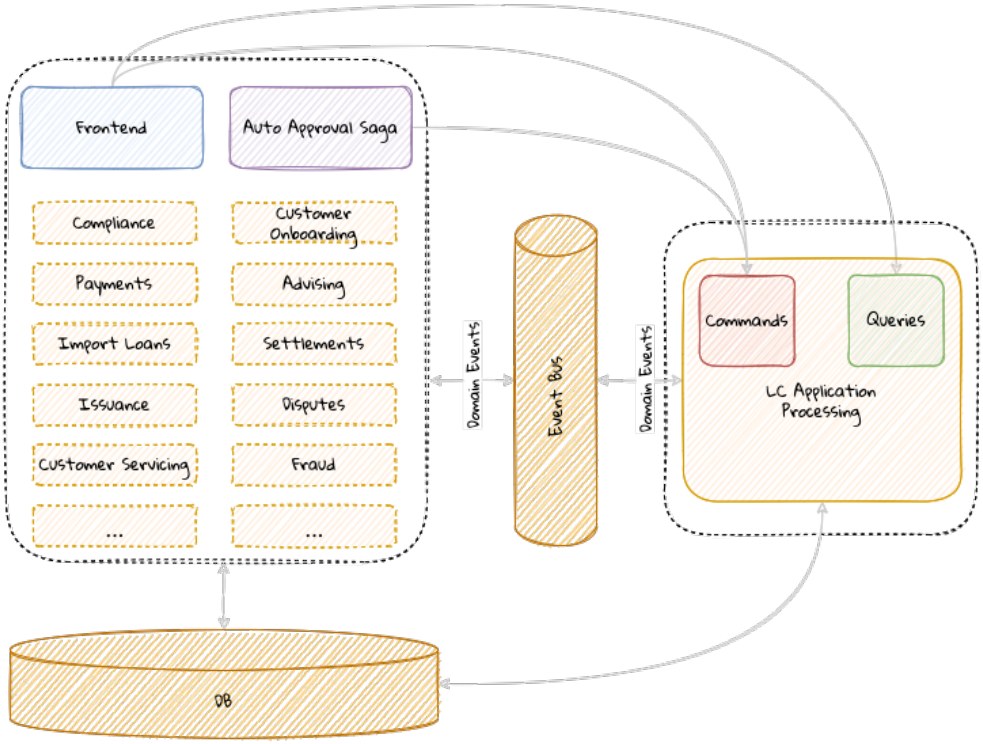


Figure 1- 106. LC Application processing deployed independently

However, we are continuing to use a common datastore for the LC application processing component.Let's look at what is involved in segregating this into its own store.

10.2.3. Changes for database interactions

While we have extracted our application component into its own unit, we continue to be coupled at the database tier.If we are to achieve true independence from the monolith, we need to break this database dependency.Let's look at the changes involved in making this happen.

Data migration

As a first step to start using a database of our own, we will need to start migrating data from the command side event store and the query store(s) as shown here:

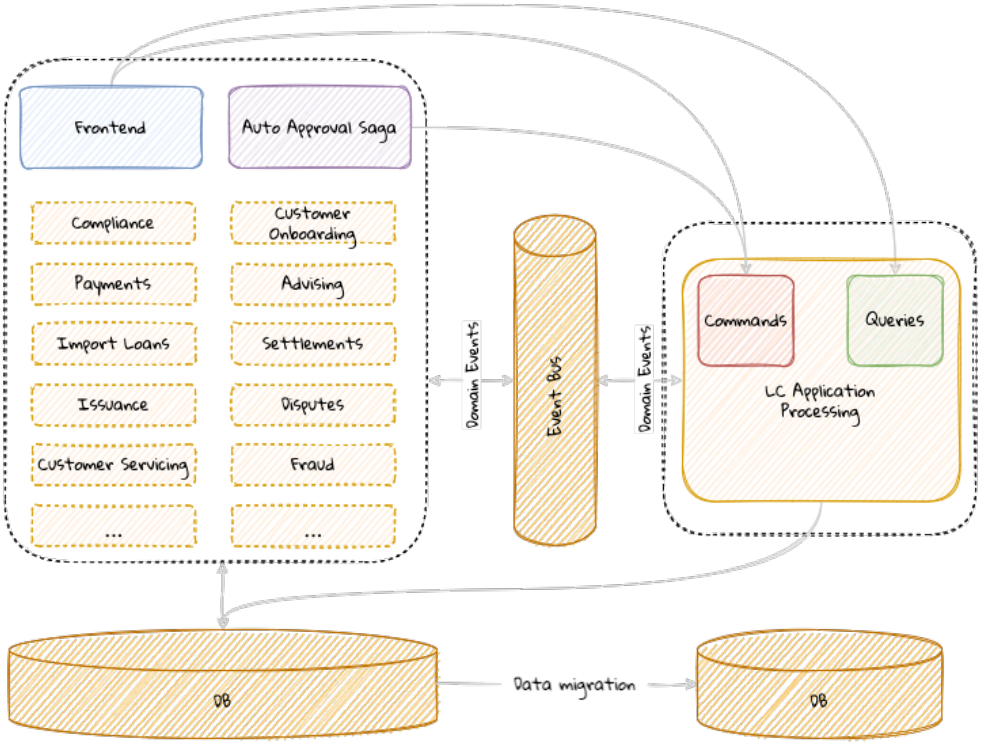


Figure 1- 107. Data migration

In our case, we have the command side event store and the query store(s) that will need to be migrated out. To minimize effort at the outset, it might be prudent to do a simple homogenous migration by keeping the source and target database technologies identical. In advance of the cut-over, among other things, it will be essential to:

- **Profile** to make sure that latency numbers are within tolerable limits.
- **Test** to make sure that the data has migrated correctly.
- **Minimize downtime** by understanding and agreeing on **SLAs** such as RTO (Recovery Time Objective) and RPO (Recovery Point Objective).

Cut-over

If we have made it thus far, we are now ready to complete the migration of the LC application processing from the rest of the monolith. The logical architecture of our solution now looks like the diagram shown here:

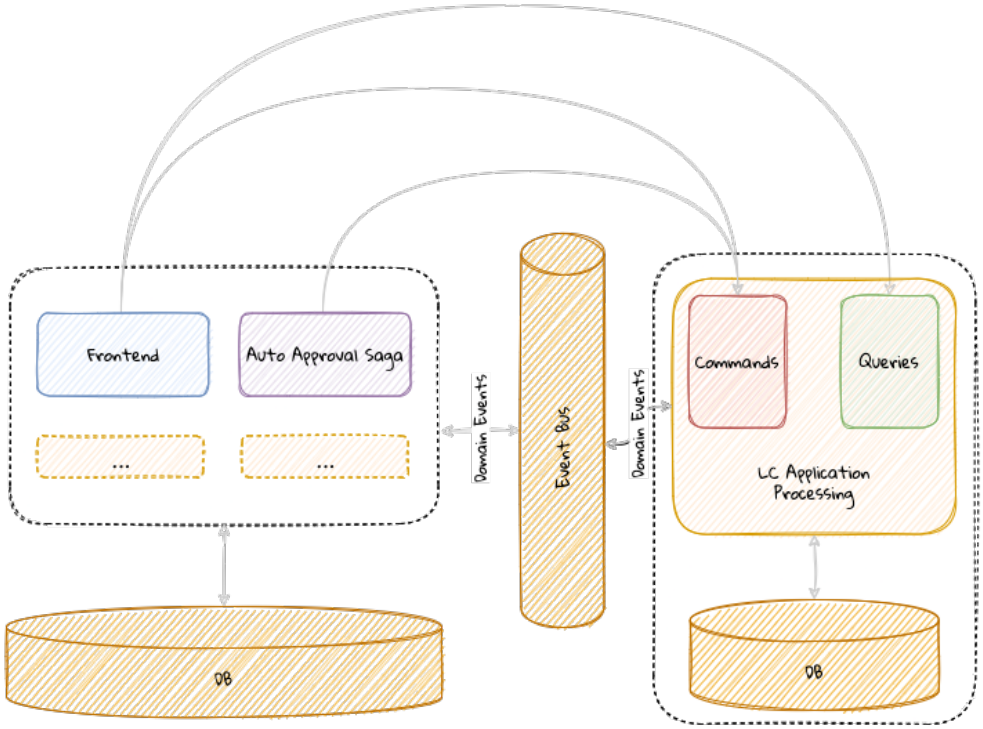


Figure 1- 108. Independent data persistence

With this step, we have successfully completed the migration of our first component. There is still quite a lot of work to do. Arguably, our component was already well-structured and loosely coupled from the rest of the application. Despite that, moving from an in-process to an out-of-process model between bounded contexts is quite an involved process — as should be evident from the work we have done in this chapter.

10.3. Summary

In this chapter, we learnt how we can extract a bounded context from an existing monolith, although one could argue that this was from a reasonably well-structured one. You should have an understanding of what it takes to go from in-process event-driven application to a distributed one.

In the next chapter, we will look at how to extract pieces out of a monolith that may not be as well-structured, possibly very close to the dreaded big ball of mud.

[45] https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

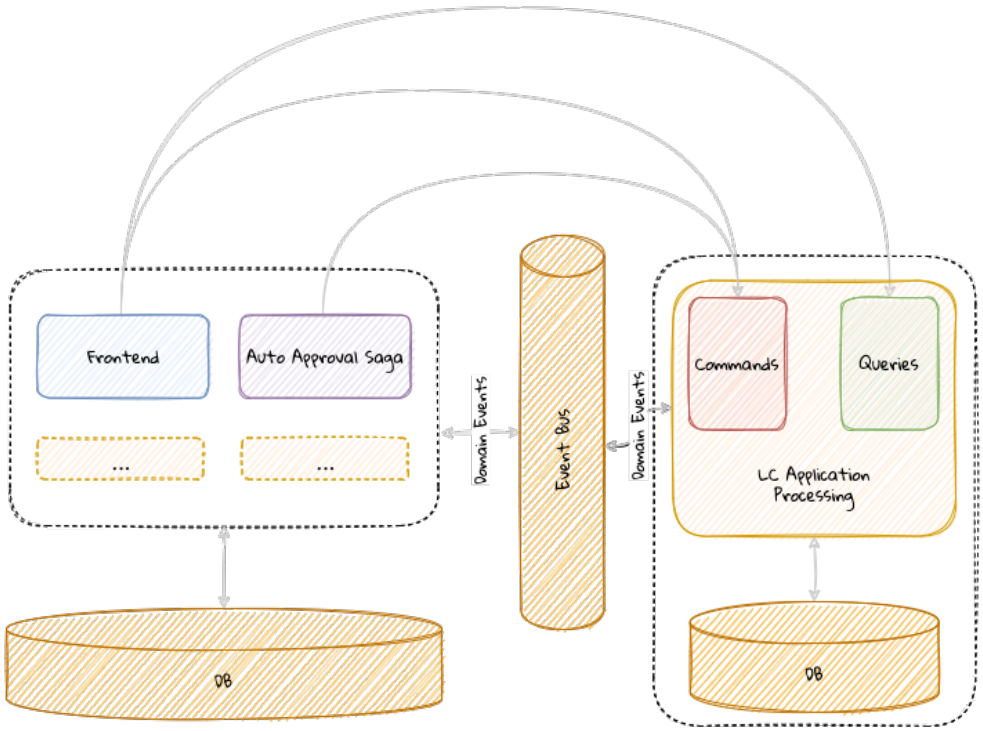
[46] https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html

[47] https://debezium.io/

Chapter 11. Distributing into finer grained components

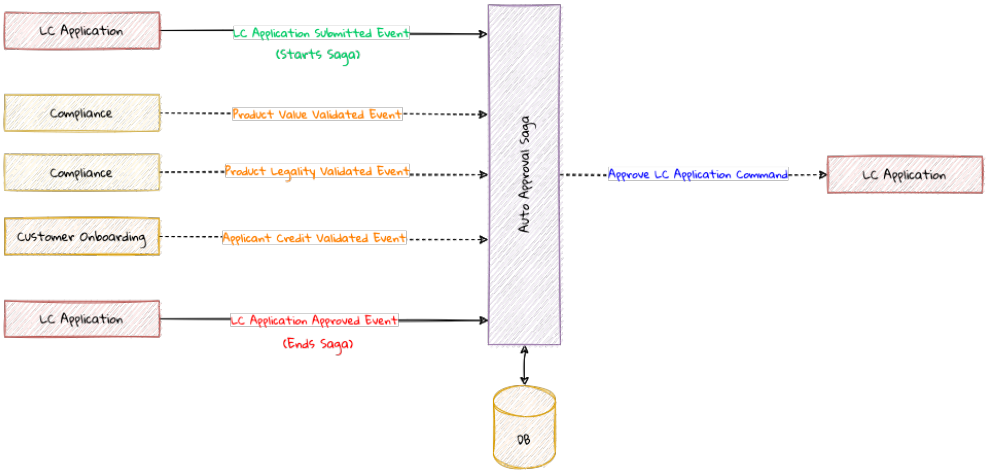
In this chapter, we will migrate the components developed thus far to adopt a serverless style of architecture.

11.1. Potential next steps



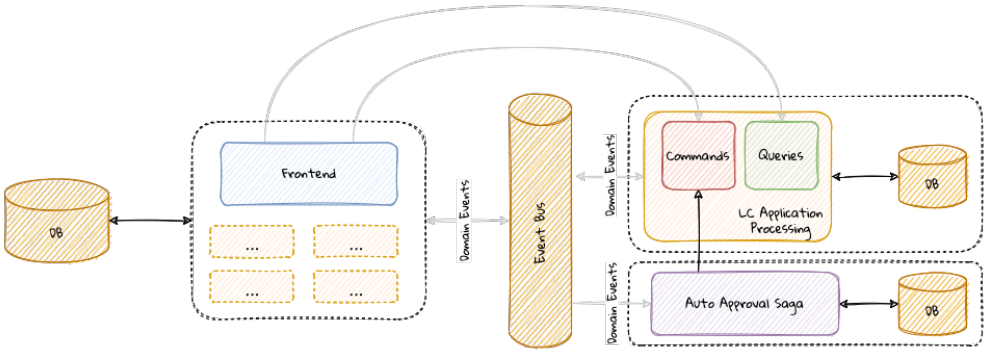
11.1.1. Saga as standalone components

Currently, the `AutoApprovalSaga` component (discussed in detail in Chapter 8) works by listening to domain events as shown here:



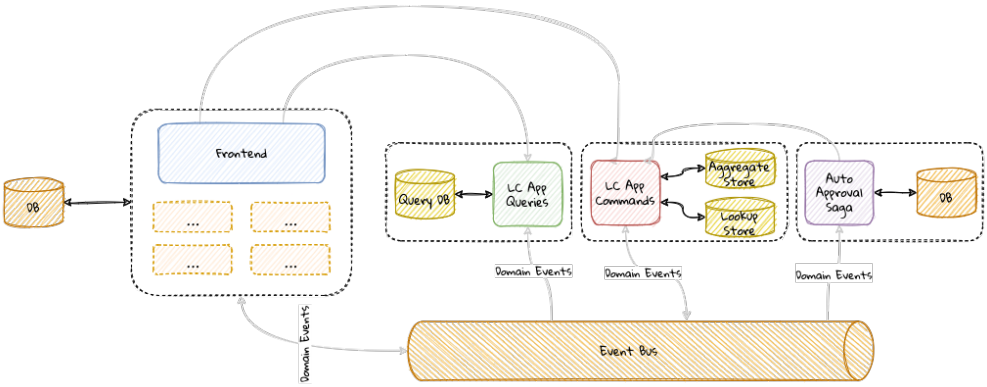
Given that these events are published by different bounded contexts onto the event bus, there is no need for `AutoApprovalSaga` to be embedded within the monolith. This means that it can be safely

pulled out into its own deployable unit along with its private datastore. This means that our system now looks like the visual depicted here:



11.1.2. Commands and queries as standalone components

As we have seen in the section on the CQRS pattern in earlier chapters, the primary benefit that we derive is gaining the ability to evolve and scale these components independently of each other. This is important because commands and queries have completely different usage patterns and thus require the use of distinct domain models. This makes it fairly natural to further split our bounded contexts along these boundaries. Thus far, the segregation is logical. A physical separation will enable us to truly scale these components independently as shown here:



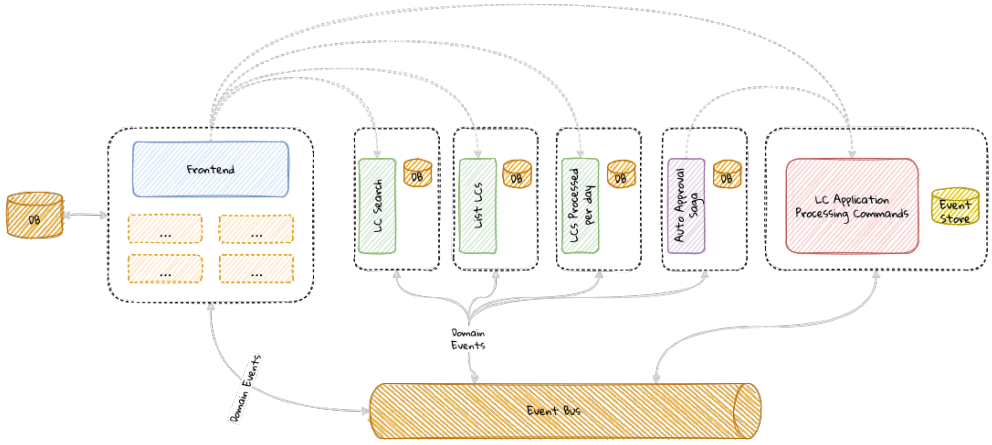
It is pertinent to note that the command processing component is now shown to have access to two distinct datastores:

- The **aggregate store**: which stores either an event-sourced or state-stored representation of aggregate state
- The **lookup store**: which can be used to store lookup data when performing business validations when processing commands. This is applicable when requiring to access data that is/cannot be stored as part of aggregate state.

The reason we bring this up is that we may have to continue making lookups for data that still remain in the monolith. To achieve full independence, this lookup data must also be migrated using techniques like a historic event replay (discussed in Chapter 7) or other conventional data migration techniques (discussed in Chapter 10).

11.1.3. Distributing individual query components

At this point, we have achieved segregation along command and query boundaries. But we do not need to stop here. Each of the queries we service need not necessarily remain a single component. Let's consider an example where we need to implement a fuzzy LC search feature for the UI and a view of LC facts for analytical use cases. It is conceivable that these requirements may be implemented by a different set of teams, thereby necessitating the need for distinct components. Even if these are not distinct teams, the disparity in usage patterns may warrant the use of different persistence stores and APIs, again requiring us to look at implementing at least a subset of these as distinct components as shown here:



Owning domains should strive to create query APIs that exhibit the characteristics of a good domain data product[48].

11.2. Even more fine-grained distribution

At this stage, is there any further decomposition that is required and feasible? These days, whether rightfully or otherwise, the serverless architecture (specifically, *Functions-As-A-Service*) is arguably becoming quite the rage. As we pointed in Chapter 2, this means that we may be able to decompose our command side in a manner that each command becomes its own independently deployable unit (hence a bounded context). In other words, the `LCApplicationSubmitCommand` and the `LCApplicationCancelCommand` can be deployed independently.

But just because this is technically possible, should we do it? While it is easy to dismiss this as a passing fad, there may be good reasons to split applications along command boundaries:

- **Risk profile**: Certain pieces of functionality present a higher risk when changes are made. For example, submit an LC application may be deemed a lot more critical than the ability to cancel it. But that is not to say that *cancel* is unimportant. Being decoupled from *submit* allows *cancel* changes to be made with a lot less scrutiny. This may make it easier to innovate at pace for more experimental features with minimal fear of causing large disruptions.
- **Scalability needs**: Scaling needs can differ wildly for various commands in the system. For example, *submit* may need to scale a lot more than *cancel*. But being coupled will force us to treat them as equals, which can be inefficient.
- **Cost attribution**: Having fine-grained components allows us to more accurately measure the amount of effort and the resulting ROI dedicated to each individual command. This can make it

easier to focus our efforts on the most critical functionality (the "core" of the core) and minimize waste.

11.2.1. Effects on the domain model

These finer grainer components are leading us to a point where it may appear that the deployment model is starting to have a big influence on the design. The fact that it is now feasible to deploy individual "tasks" independently, requires us to re-examine how we arrive at bounded contexts. For example, we started by working on the *LC Application Processing* bounded context. And our aggregate design was based on all functionality included in the scope of application processing. Now, our aggregate design can be a lot more fine-grained. This means that we can have an aggregate specifically for *start* functionality and another for *cancel* as shown here:

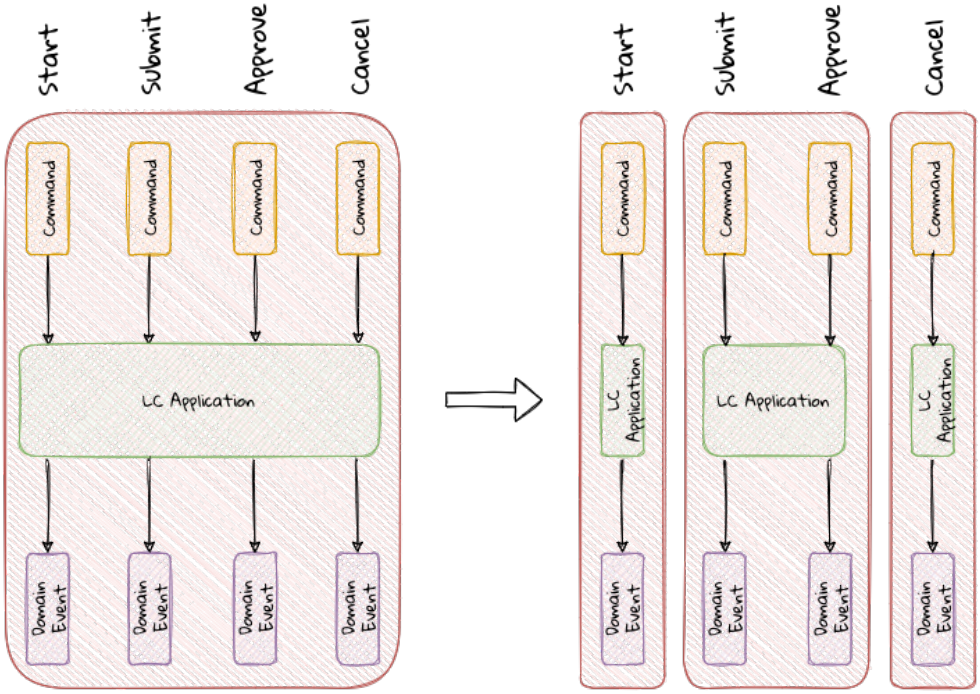


Figure 1- 109. Fine-grained bounded contexts example

The most fine-grained decomposition may lead us to a bounded context per command. But that does not necessarily mean that we have to decompose the system this way. In the above example, we have chosen to create a single bounded context for the *submit* and *approve* commands. However, *start* and *cancel* have their own bounded contexts. The actual decision that you make in your own ecosystems will depend on maintaining a balance among reuse, coupling, transactional consistency and other considerations that we discussed earlier. It is important to note that the aggregate labeled as `LCApplication`, although named identically, is distinct from a domain model perspective in its respective bounded context. The only attribute they will need to share is a **common identifier**. If we choose to decompose the system into a bounded context per command, our overall solution will look like the visual shown here:

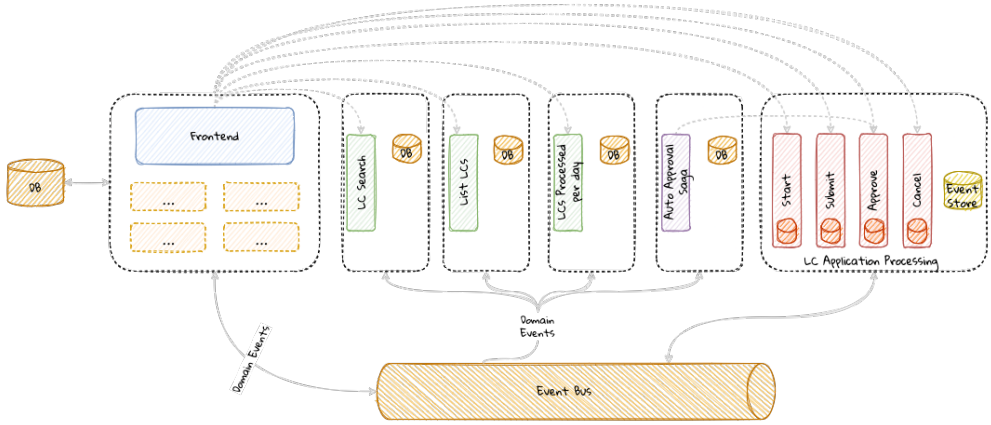


Figure 1- 110. Decomposition per command

It is pertinent to note that the *command* functions continue to share a single event store, although they may make use of their own individual lookup stores. We understand that this decomposition likely feels unnecessary and forced. However, this does allow us to focus our energies on the ***core** of the core*. For example, LC application processing may be our business differentiator. But an even more careful examination may reveal that it is our ability to *decision* LCs near real-time that is our real business differentiator. This means that it may be prudent to isolate that functionality from the rest of the system. In fact, doing so may enable us to optimize our business process without adding additional risk to the overall solution. While it is not strictly necessary to decompose the system in this way to arrive at such insights, the fine-grained decomposition may enable us to refine the idea of what is most important to our business. Having to share a persistent store can be a wrinkle to achieve complete independence. So a final decomposition may look something like what we show here:

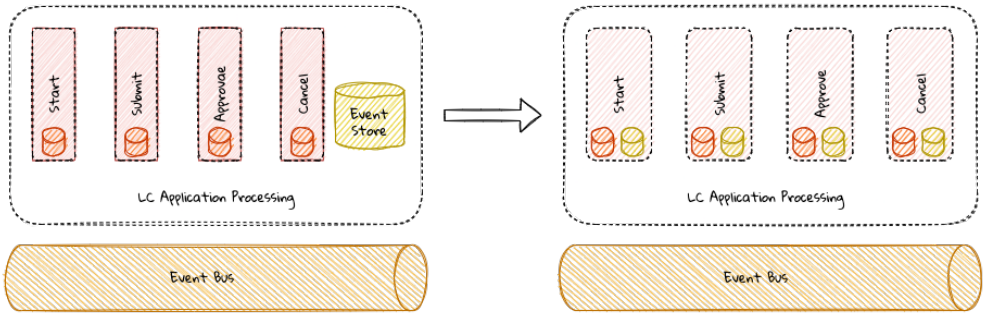


Figure 1- 111. Command components with individual event stores.

Obviously, there is no free lunch! This fine-grained decomposition may require additional coordination and duplication of data among these components — to a point where it may not be attractive anymore. But we felt that it is important to illustrate the art of the possible.

11.3. Customer experiences and frontends

Thus far, we have focussed on decomposing and distributing the backend components while keeping the frontend untouched as part of the existing monolithic system. It is worth considering breaking down the frontend to align more closely along functional boundaries. Patterns like micro-frontends[49][50] extend the concepts of microservices to the frontend. Micro-frontends promote team structures to support end-to-end ownership of a set of features. It is conceivable that a cross-functional, polyglot team owns both the experience (frontend) and the business logic (backend) functions eliminating communication overheads drastically (along the lines of the vertical slice

architecture conversation in Chapter 2). Even if such a team organization where the frontend and backend being one team is not feasible in your current ecosystem, this approach still has many merits such as:

Increased Customer focus?

Autonomy/Technology agnostic

Decoupled code-bases allow for parallelization

Independent Deployment?

While there are many advantages in considering the micro-frontend approach, it does bring some challenges :

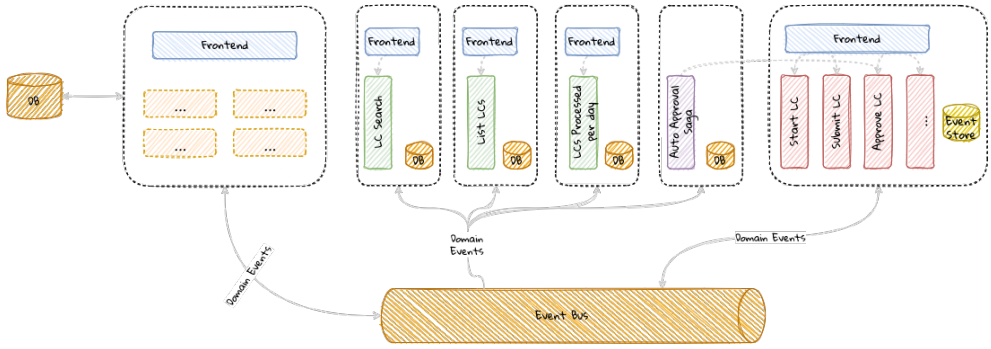
Deployment complexity - TODO: Mobile Apps? Single SPA

Build time integration

Duplication

Runtime Inefficiency (more code downloaded to browser)

Consistency (UI & User Experience, different framework versions)



11.4. Where to draw the line?

In general, the smaller the size of our bounded contexts, the easier it becomes to manage complexity. Does that mean we should decompose our system into as fine-grained a granularity as possible? On the other hand, having extremely fine-grained components can increase coupling among them to an extent where it becomes very hard to manage operational complexity. Hence, decomposing a system into collaborating components can be a bit tricky, seeming to work more like an art, rather than an exact science. There is no right or wrong answer here. In general, if things feel and become painful, you most likely got it more wrong than right. Here are some non-technical heuristics that might help guide this process:

- **Existing organization boundaries**: Look to align along current org structures. Identify which applications your business unit/department/team already owns and assign responsibilities in a manner that causes minimal disruption.
- **Domain expert roles and responsibilities**: What work do your domain expert carry out? What

enables them to do their work with the least friction possible?

- **Change in vernacular**: Look for subtle changes in the usage of common terms. Does someone call something that is/feels the same in the physical world by different names? For example, a credit card may be called "plastic", "payment instrument", "account" by different people or the same people in a different context.
- **Existing (modular/monolithic/distributed) applications**: How are your current applications segregated logically? How are they segregated physically? This might provide some inspiration.

But what if one or more of the above are wrong/cumbersome/suboptimal? In such a case, our work as developers/architects is a bit more involved. It is not uncommon to get domain boundaries wrong. Come up with an initial breakdown that seems to make more sense and apply a series of *what if* questions to assess suitability. If the reasoning is able to stand up to scrutiny by domain experts, architects and other stakeholders, you might have your answer. If you do choose to go this route, it may be prudent to adjust existing organization structures to match your proposed architecture. This will help reduce friction (in other words, you have applied the **inverse Conway maneuver**[51]).

Despite all our due diligence and noble intentions, it is still possible to get these boundaries wrong. Or a change in business priorities or competitor offerings may render decisions that appeared perfectly valid at the time to become incorrect. Instead of looking to arrive at the perfect decomposition, it might be prudent to embrace change and invest in building designs that are flexible while being prepared to evolve and refactor the architecture iteratively. The book on building evolutionary architectures[52] has some great advice on how to do precisely that.

[48] https://martinfowler.com/articles/data-monolith-to-mesh.html#DomainDataAsAProduct

[49] https://micro-frontends.org/

[50] https://martinfowler.com/articles/micro-frontends.html

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

[52] https://evolutionaryarchitecture.com/

Chapter 12. Migrating to Serverless (15 pages)

In this chapter, we will migrate the components developed thus far to adopt a serverless style of architecture.

12.1. Serverless Primer

12.2. Services as Functions

12.3. Serverless Persistence

12.4. Next Steps

12.5. Legacy Application Migration Patterns

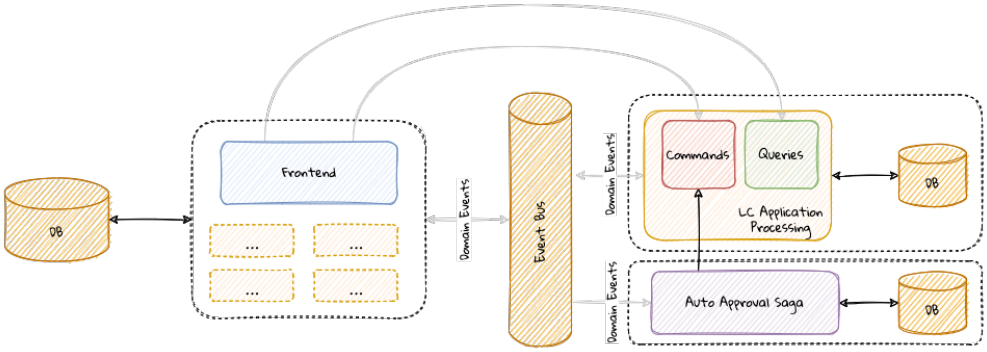
12.6. Decomposing and distributing big balls of mud

12.6.1. Code-first

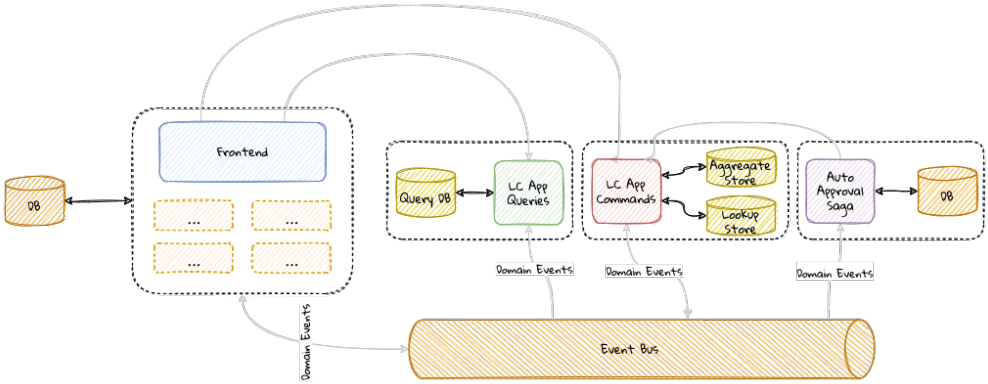
12.6.2. Data-first

12.6.3. Potential next steps

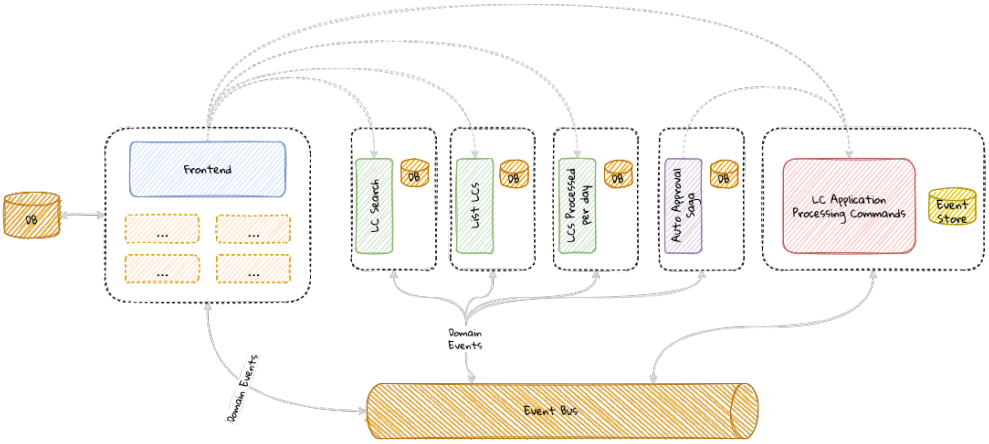
Sagas as standalone components



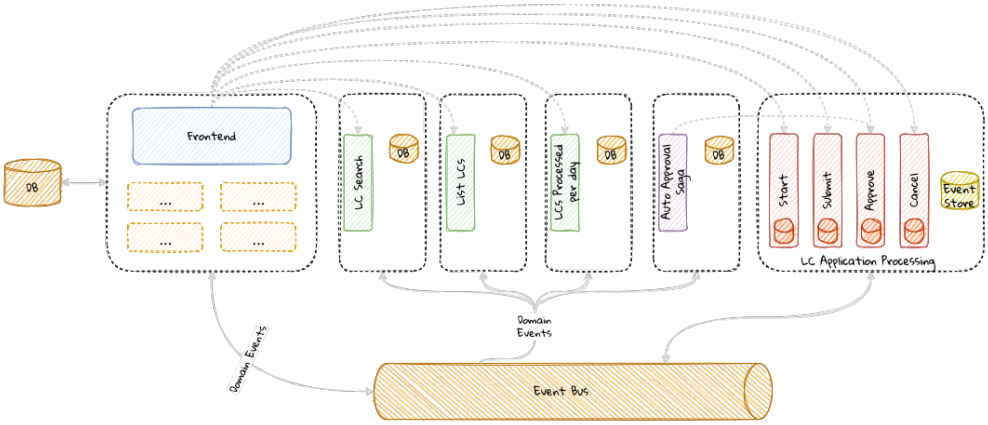
Commands and queries as standalone components



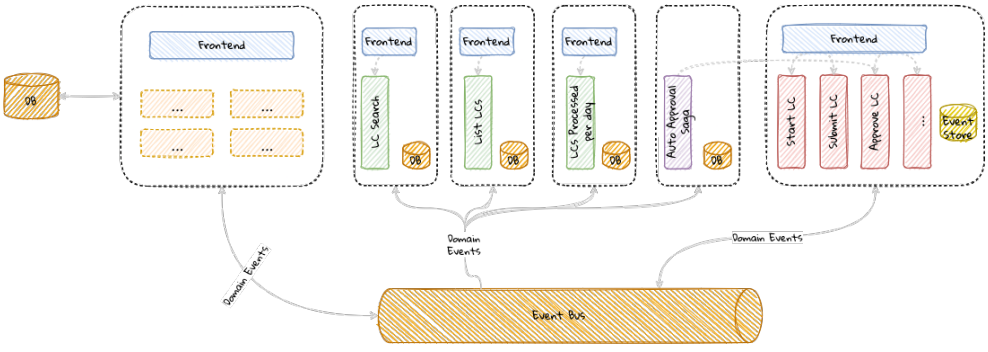
Distributing individual query components



12.6.4. Even more fine-grained distribution



12.6.5. Customer experiences and frontends



12.6.6. Technical implications of distribution

Performance

Resilience

Exception handling

> Everything fails all the time.
>
> — Werner Vogels, CTO -- Amazon Web Services

Unexpected failures in software systems are bound to happen. Instead of expending all our energies trying to prevent them from occurring, it is prudent to embrace and design for failure as well. Let's look at the scenario in the `AutoApprovalSaga` and identify things that can fail:

```
class AutoApprovalSaga {
    //...
    @SagaEventHandler(associationProperty = "lcApplicationId")
    public void on(ProductLegalityValidatedEvent event) {
        //..
        productLegalityValidated = true;
        if (productValueValidated && applicantValidated) {
               gateway.send(new ApproveLCApplicationCommand(lcApplicationId)); ①
        }
    }
}
```

① When dispatching commands, we have a few styles of interaction with the target system (in this case, the LC application bounded context):

- **Fire and forget**: This is the style we have used currently. This style works best when system scalability is a prime concern. On the flip side, this approach may not be the most reliable because we do not have definitive knowledge of the outcome.
- **Wait infinitely**: We wait infinitely for the dispatch and handling of the `ApproveLCApplicationCommand`.
- **Wait with timeout**: We wait for a certain amount of time before concluding that the handling of the command has likely failed.

Which interaction style should we use? While this decision appears to be a simple one, it has quite a few, far-reaching consequences:

- If command dispatching itself fails, the `CommandGateway#send` method will fail with an exception. Given that the CommandGateway is an infrastructure component, this will happen because of technical reasons (like network blips, etc.)
- If the command handling for the `ApproveLCApplicationCommand` fails, and we will not know about it because the `#send` method does not wait for handling to complete. One way to mitigate that problem is to wait for the command to be handled using the `CommandGateway#sendAndWait`

method. However, this variation waits infinitely for the handler to complete — which can be a scalability concern.

- We can choose to only wait

Recovery

Automated recovery

Manual recovery

Compensating actions

Retries

Integration strategies

Testing strategies

12.7. Not yet finalized

12.7.1. Transitional architecture

12.8. Understanding the costs of distribution

12.8.1. Handling exceptions

12.8.2. Testing the Overall System

12.8.3. Compatibility

12.8.4. Non-technical implications of distribution

Team topologies

Tech stack

Chapter 13. Non-Functional Requirements (25 pages)

> Sometimes I feel like I am being forgotten.
>
> — Anonymous

While the functional requirements of the core of the system may be met adequately, it is just as important to place focus on the operational characteristics of the system. In this chapter, we will look at common pitfalls and how to get past them.

13.1. Dealing With Eventual Consistency

13.2. Scaling the Event Store with Snapshots

13.3. Event Versioning and Upcasting

13.4. Monitoring, Metrics and Tracing

13.5. Enhancing Performance

13.5.1. Scaling the event bus