Implementing Domain-Driven Design with Java

Premanand Chandrasekaran, Karthik Krishnan

Version 1.0.0-SNAPSHOT, Nov 24 2021 11:01 AM 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.

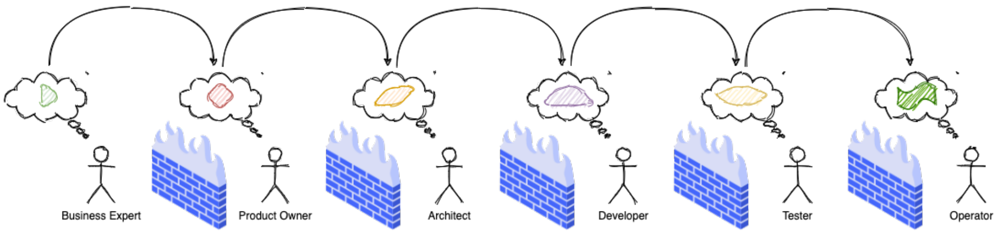


Figure 1- 1. Silo mentality and the loss of information fidelity

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:

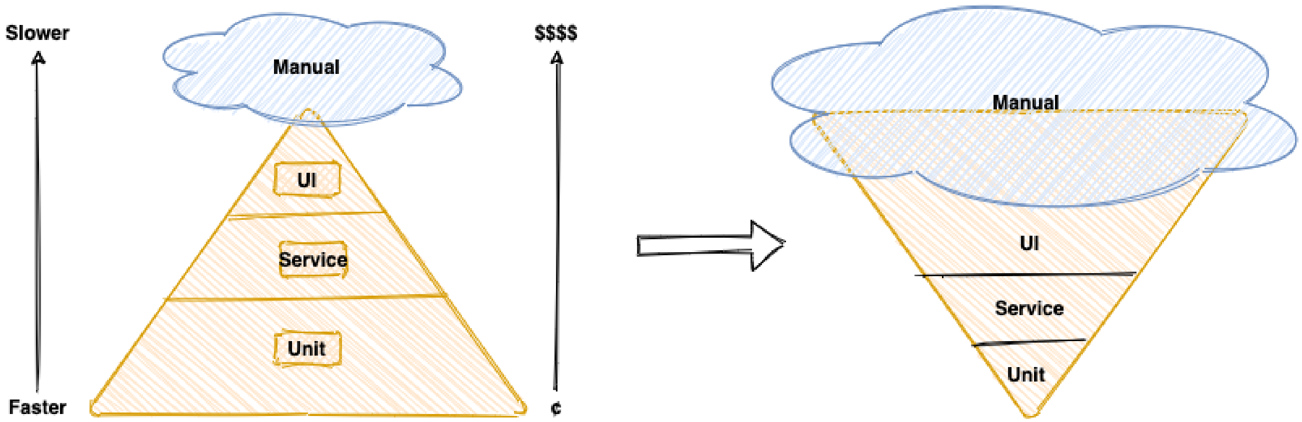


Figure 1- 2. Testing Strategy: Expectation vs. Reality

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:

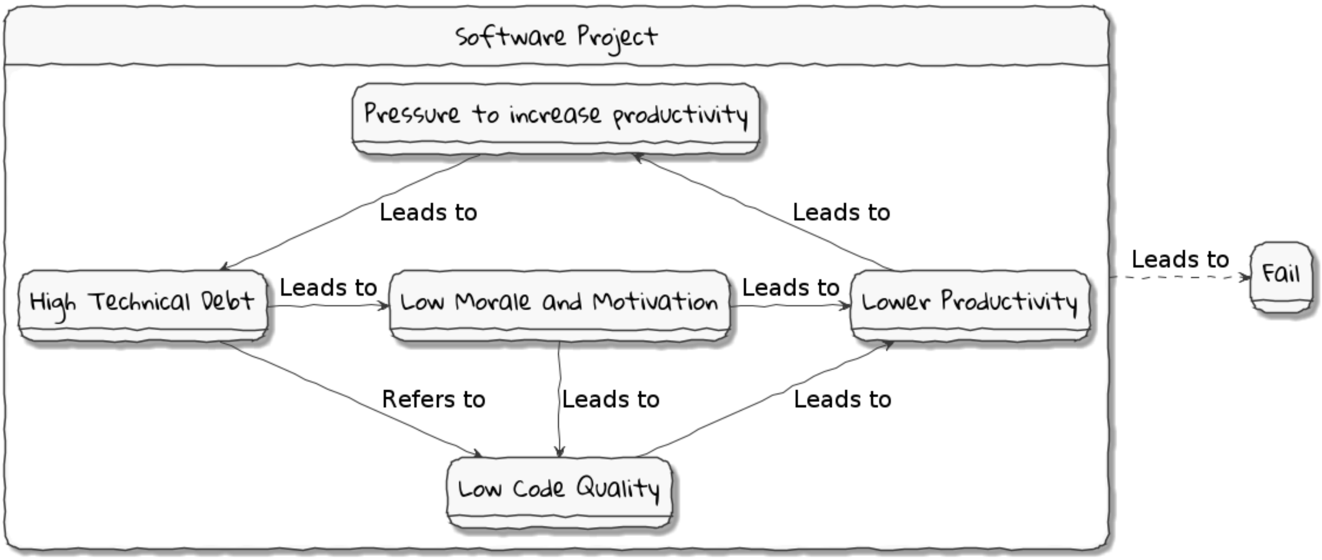


Figure 1- 3. Technical Debt — Implications

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:

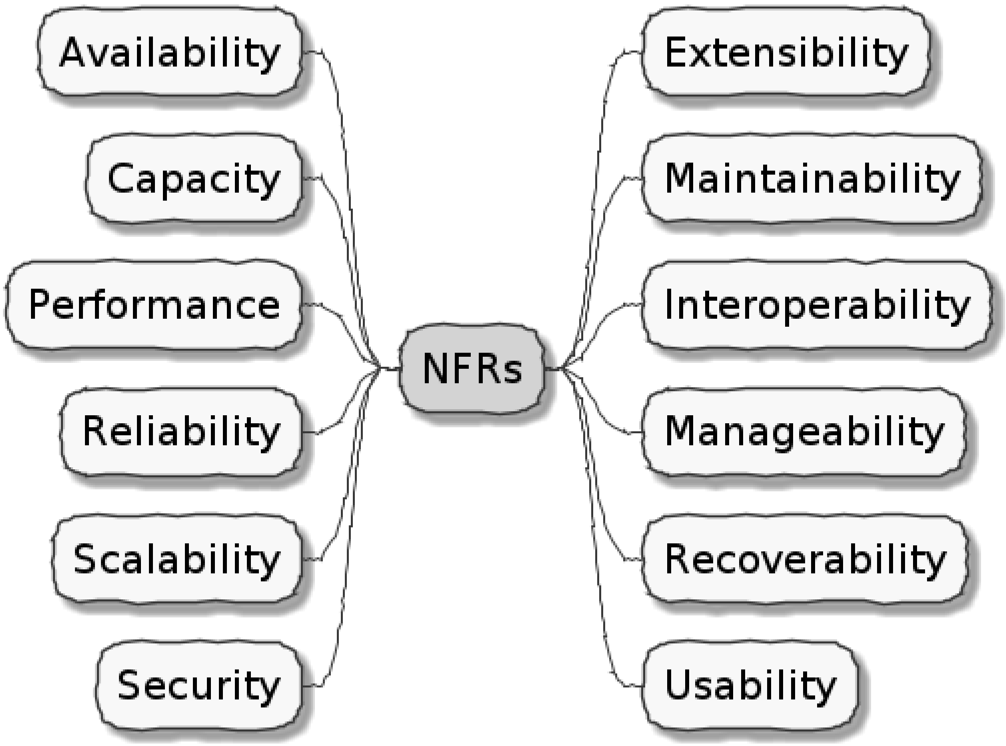


Figure 1- 4. Non-Functional Requirements

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.

1.2.7. Where To From Here?

In this section we examined some common reasons that cause software projects to fail. In the upcoming section, we will look at characteristics of modern systems and look at more effective ways to deal with software complexity. In upcoming chapters, we will look at how applying domain-driven design helps mitigate these causes of failure.

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

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,

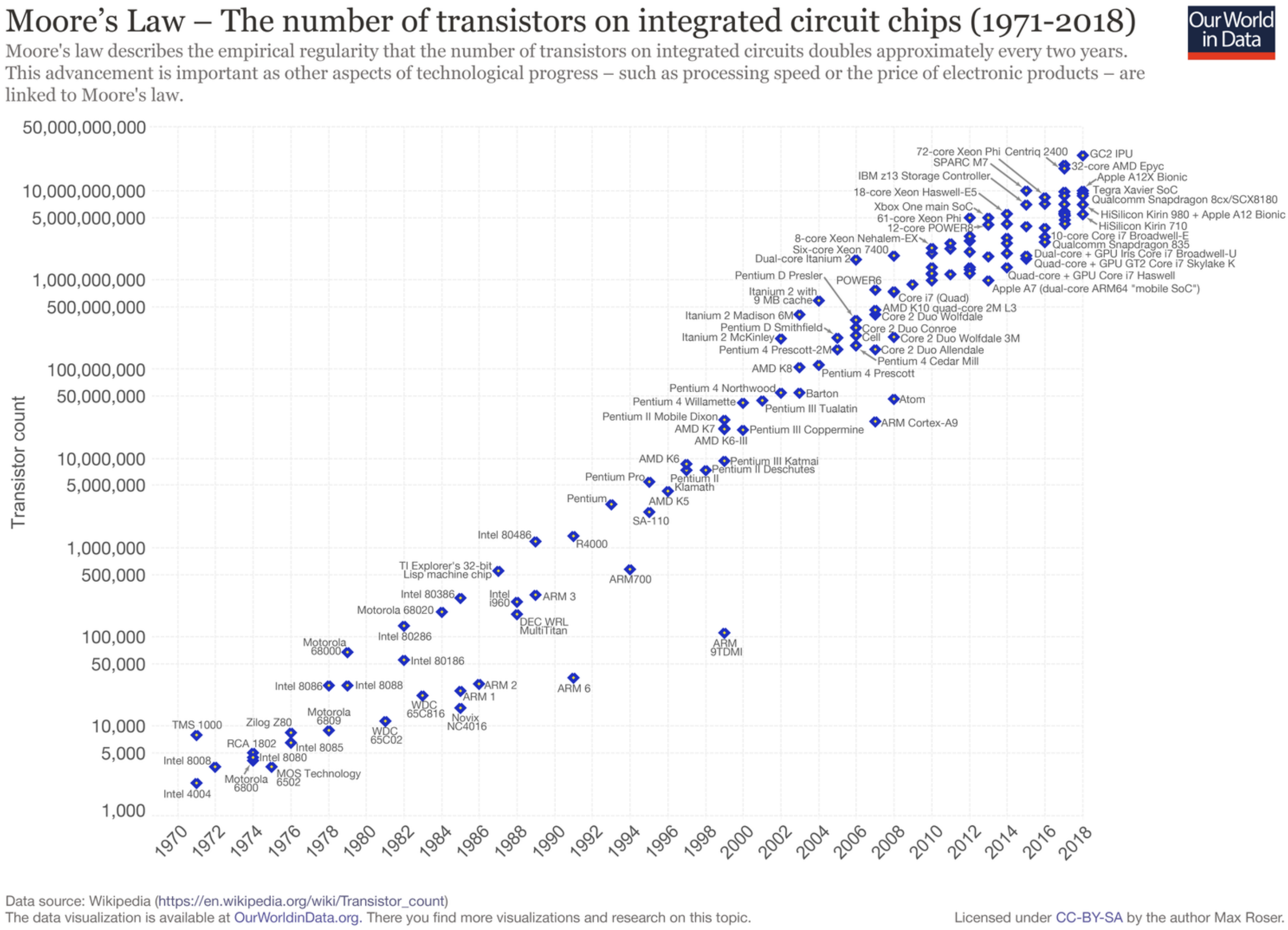


Figure 1- 5. Moore's Law

along with the rise of the internet as illustrated below,

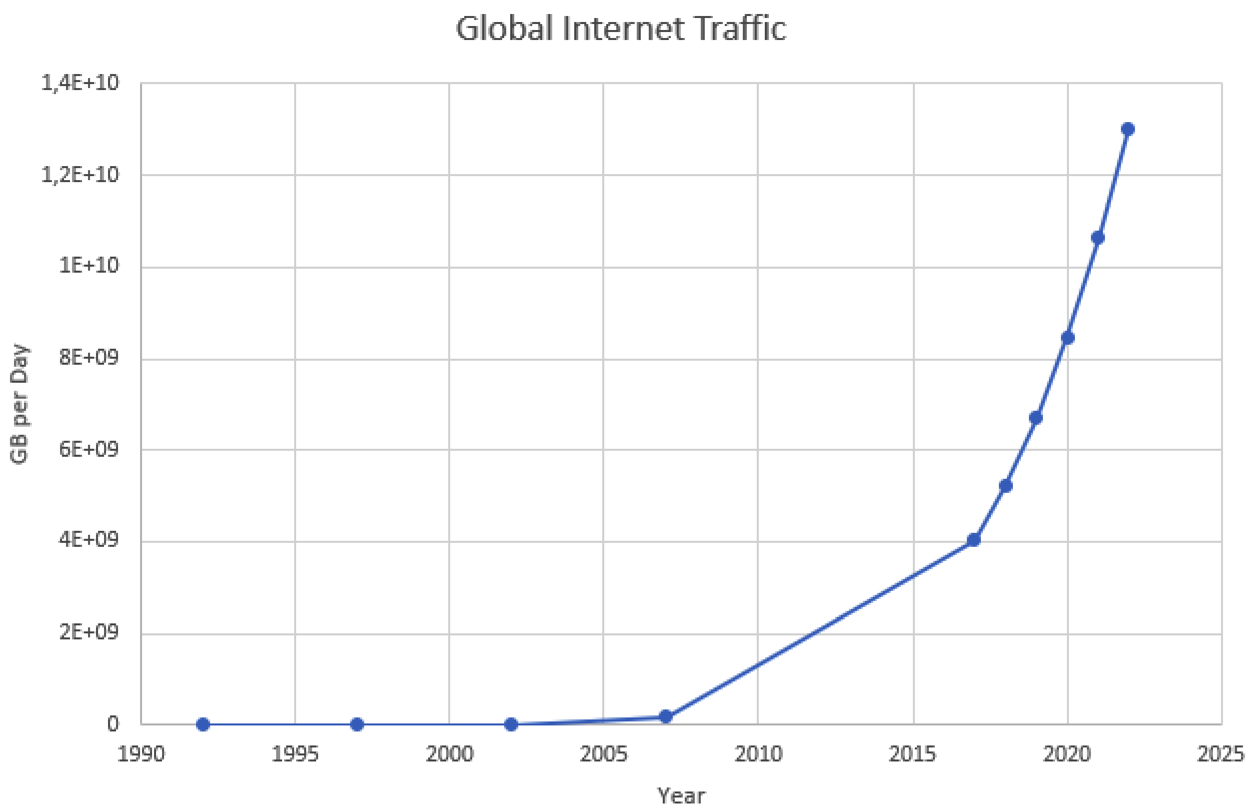


Figure 1- 6. Global Internet Traffic

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.

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.

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 was indeed seminal, ground-breaking, and way ahead of its time, over the years, practical application has continued to remain a challenge. In this section, we will look at some of the foundational terms and concepts behind domain-driven design. Elaboration and practical application of these concepts will happen in upcoming chapters of this book.

To understand DDD, first and foremost, we need to understand what we mean by the first "D" — **domain**.

1.4.1. 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- 7. **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.

1.4.2. What is a Subdomain?

The domain of a business (at least the successful ones) almost always encompasses fairly complex and abstract concepts. With a view to better deal with this complexity, domain-driven design advises decomposing the domain of a business into multiple manageable parts called **subdomains**. This facilitates better understanding and makes it easier to arrive at a solution. For example, the online retail domain may be divided into subdomains such as product, inventory, rewards, shopping cart, order management, payments, shipping, etc. as shown below:

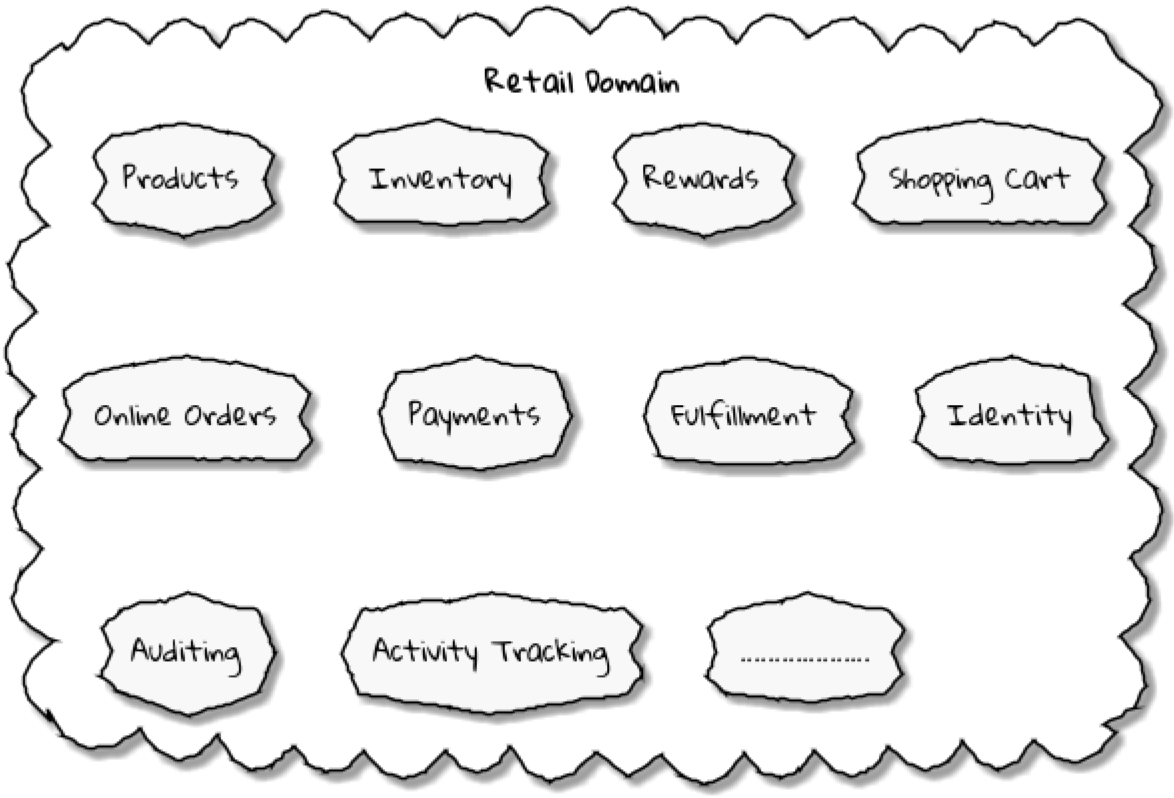


Figure 1- 8. Subdomains in the Retail domain

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:

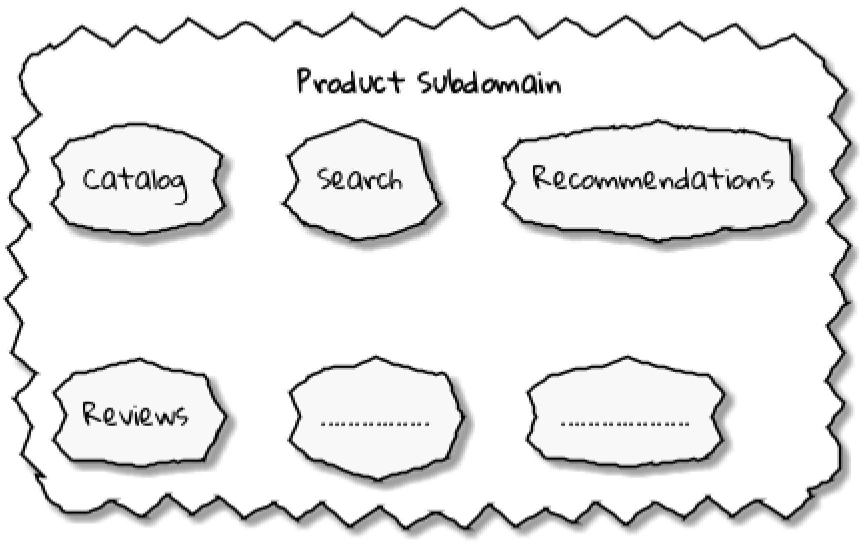


Figure 1- 9. Subdomains in the Products subdomain

Further breakdown of subdomains may be needed until we reach a level of manageable complexity.

1.4.3. Types of Subdomains

Breaking down a complex domain into more manageable subdomains is a great thing to do.

However, not all subdomains are created equal. With any business, the following three types of subdomains are going to be encountered:

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

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

1.4.4. Domain Experts

To run a successful digital business, you need specialists — 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.

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

1.4.5. Promoting a Shared Understanding

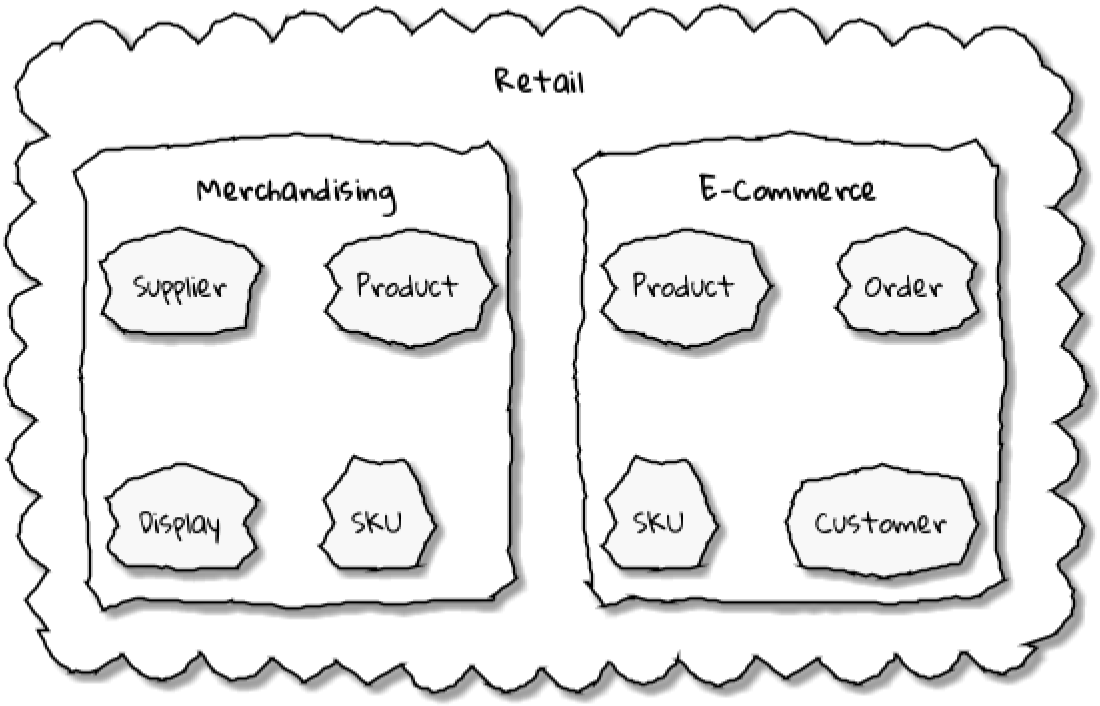
Previously, we saw how organizational silos can result in valuable information getting diluted. At a credit card company I used to work with, the words plastic, payment instrument, account, PAN (Primary Account Number), BIN (Bank Identification Number), card were all used by different team members to mean the exact same thing - the ***credit card*** when working in the same area of the application. To make matters worse, a lot of this 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.

1.4.6. 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. It is very important to note that this domain model is modeled to fall within the context of a single subdomain for which a solution is being explored, not the entire domain of the business. This boundary is termed as a ***bounded context*** i.e. the ubiquitous language and domain model are only valid within those bounds and context — not outside of it. This means that 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***.

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.



DDD has a catalog of strategic and tactical patterns which accelerate this process of continuous learning. In addition, modern techniques such as domain storytelling, event storming, and evolutionary architecture can greatly aid this process of evolving the ubiquitous language and domain model. We will examine all of these in much detail in upcoming chapters,

> The thrust of DDD is that **one single model** form the bedrock of team communication, design, and implementation. While teams may and will indeed require a variety of means to express the model, it is very important to keep the executable code and the various representations up to date at all times.

1.4.7. The Essence of DDD

In this section we have taken a look at DDD at a very high level. Enclosed below is an attempt to capture the essence of what domain-driven design means.

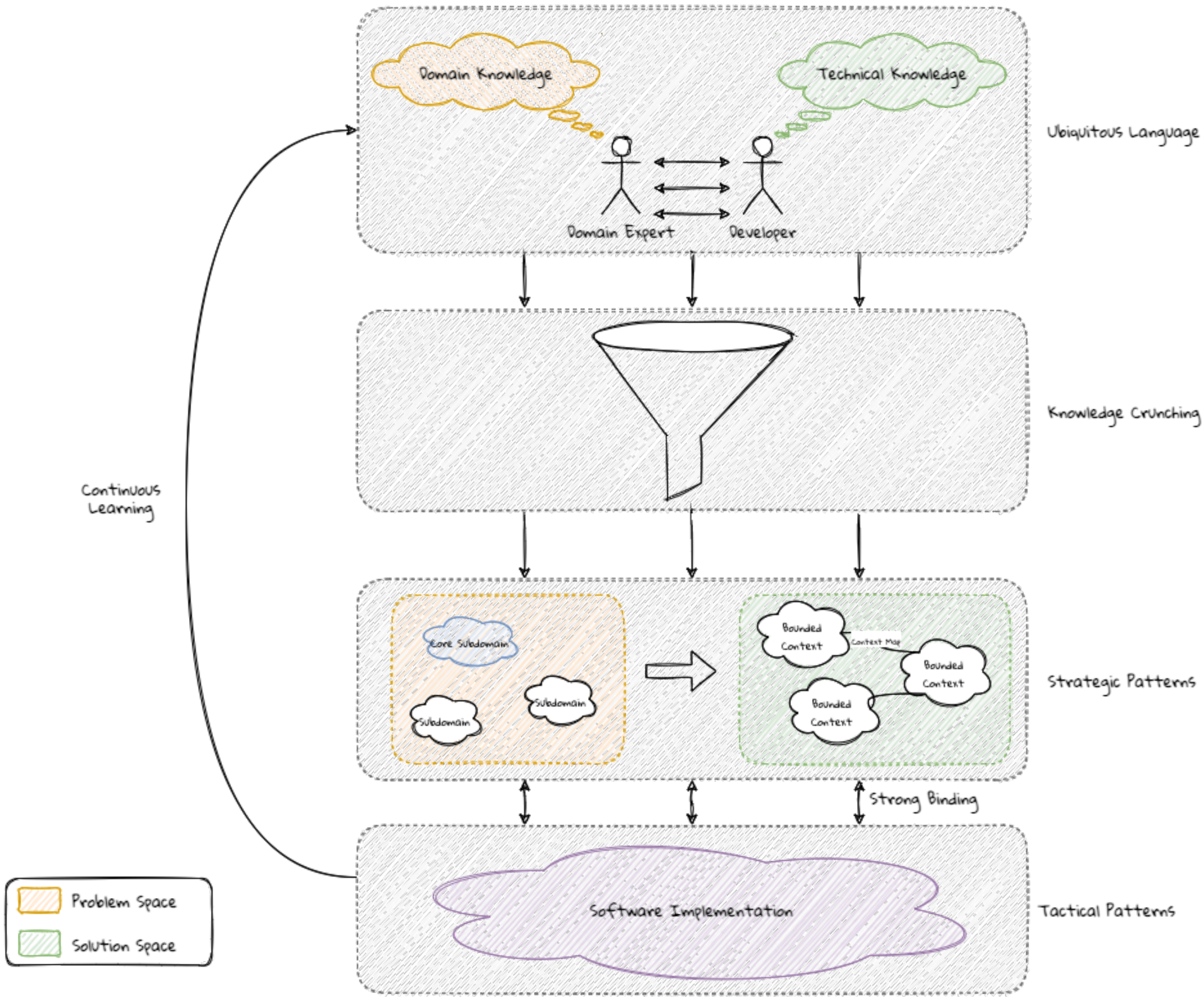


Figure 1- 10. Essence of DDD

In subsequent chapters we will reinforce all of 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.

1.5. Why is DDD Relevant? Why Now?

> He who has a why to live for can bear almost any how.
>
> — 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:

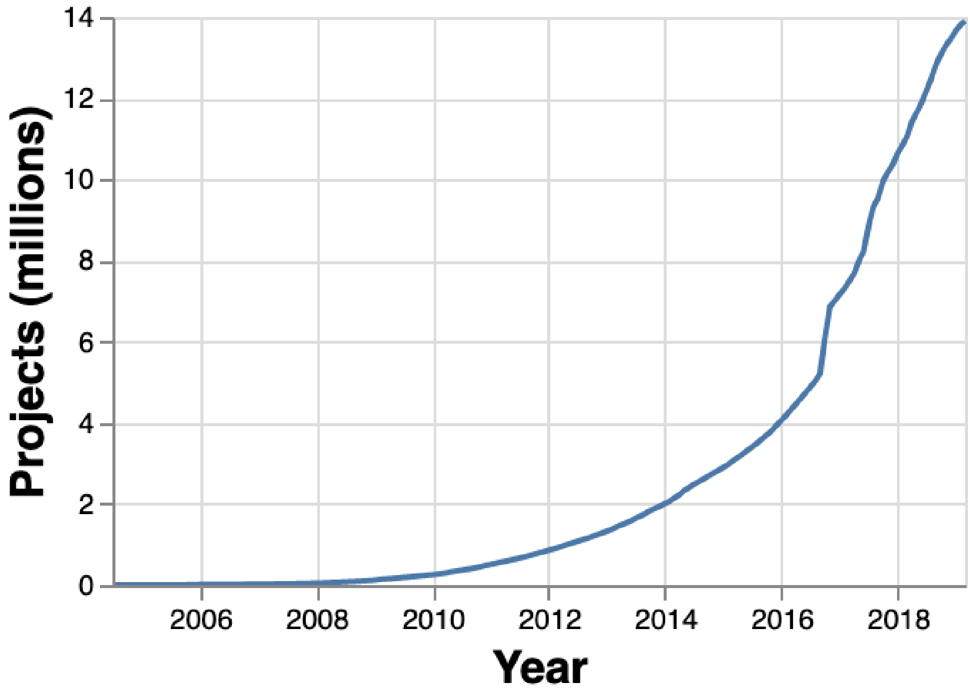


Figure 1- 11. Open source Java over the years. Source: https://mvnrepository.com/

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 buiding 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- 12. 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. What do the terms domain and sub-domain mean?
3. What are the different types of sub-domains?
4. What is the difference between sub-domains and bounded contexts?
5. 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

Title	Author	Location
Project success: definitions and measurement techniques	JK Pinto, DP Slevin	https://www.pmi.org/learning/library/project-success-definitions-measurement-techniques-5460
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

```java
class Account {

    void openCardAccount(Applicant applicant,
                         MonetaryAmount initialAmount) {
        Status status = checkCreditWorthiness(applicant);
        if (status.isWorthy()) {
            Account account = Account.createFor(applicant);
            account.deposit(initialAmount);
            account.activate();
        }

    }
}
```

1.9. Answers

1. Refer to section 1.2
2. Refer to sections 1.4.1 and 1.4.2
3. Refer to section 1.4.3
4. Refer to section 1.4.7
5. Refer to section 1.5

Chapter 2. The Mechanics of Domain-Driven Design (30 Pages)

> When eating an elephant, take one bite at a time.
>
> — Creighton Abrams

As mentioned in the previous chapter, many things can put a project off course.In this chapter, we look at how DDD gives us a set of tenets and techniques to arrive at a collective understanding of the problem at hand in the face of ambiguity, break it down into manageable chunks and translate it into reliably working software.

2.1. Understanding the Problem

2.1.1. Problem Space vs. Solution Space

2.1.2. Dealing with Ambiguity

2.2. Arriving at a Shared Understanding

2.3. Breaking Down the Problem

2.3.1. What is a Domain?

2.3.2. What is a Sub-Domain?

2.3.3. The Core Sub-Domain

2.4. Modeling a Solution

2.4.1. What is a Model?

> ⚠ Anemic domain models

2.4.2. Context Maps

2.4.3. Bounded Contexts

2.5. Implementing the Solution

2.5.1. Entities

2.5.2. Value objects

2.5.3. Aggregates

Chapter 3. Where and How Does DDD Fit? (15 pages)

> 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 to help us deal with system complexity. In this chapter we will examine how DDD compares with several of these architecture styles and how/where it fits in the overall scheme of things when crafting a software solution.

3.1. Architecture Styles

3.2. Layered Architecture

3.3. Onion Architecture

3.4. Hexagonal Architecture

3.5. Service Oriented Architecture

3.6. Microservice Architecture

3.7. Event-Driven Architecture (EDA)

3.8. 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:

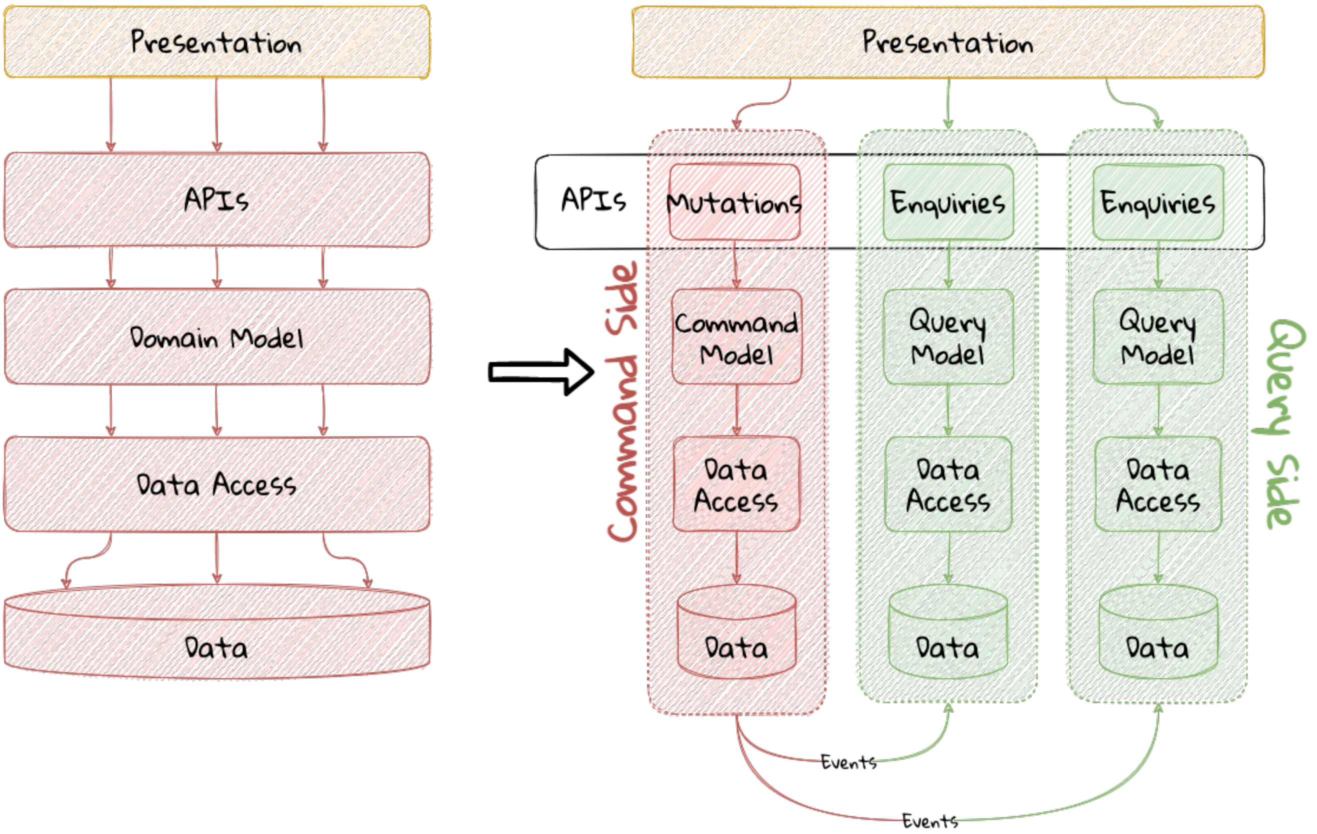


Figure 1- 13. Traditional versus CQRS Architecture

> 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.

3.8.1. When to use CQRS?

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.

3.9. Serverless Architecture

Part 2: Implementing DDD in the real world

In Part 2, we will implement a real-world application using JVM-based technologies such as Vaadin, Spring Boot, Axon Framework, Cadence, among others.

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/)

4.3. Introducing the LC application

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.

4.3.1. What is a 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).

4.3.2. The Letter of Credit business

TODO

4.3.3. The LC issuance application

XYZ Bank has reached out to us in order to automate the process of LC application and issuance. In this chapter, and indeed the rest of this book, we will strive to understand, evolve, design and build a software solution to automate this process.

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

Above 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.

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 audio-visual 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:

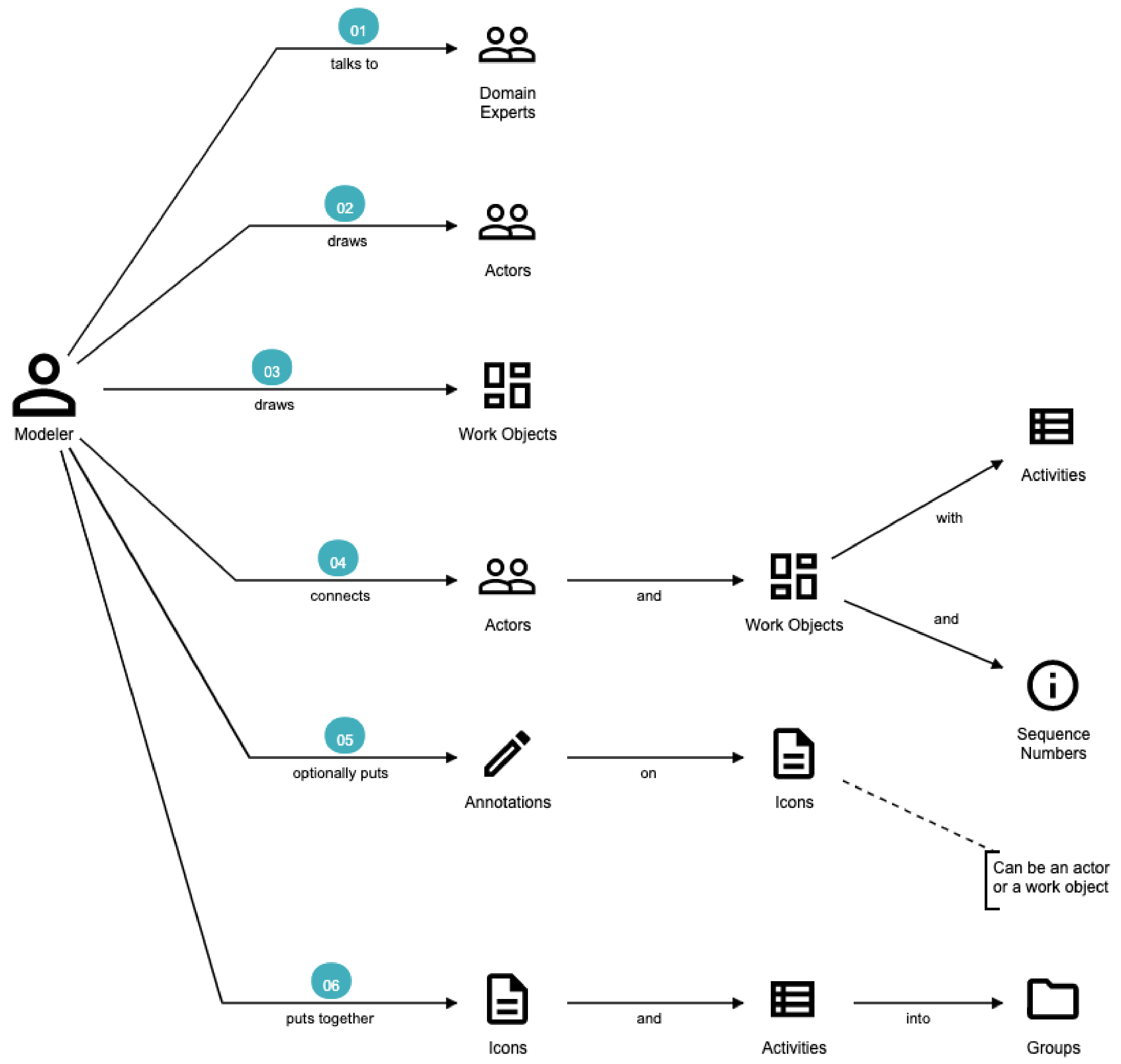


Figure 1- 14. Domain storytelling summarized

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 sub-domains/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- 15. 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**."*

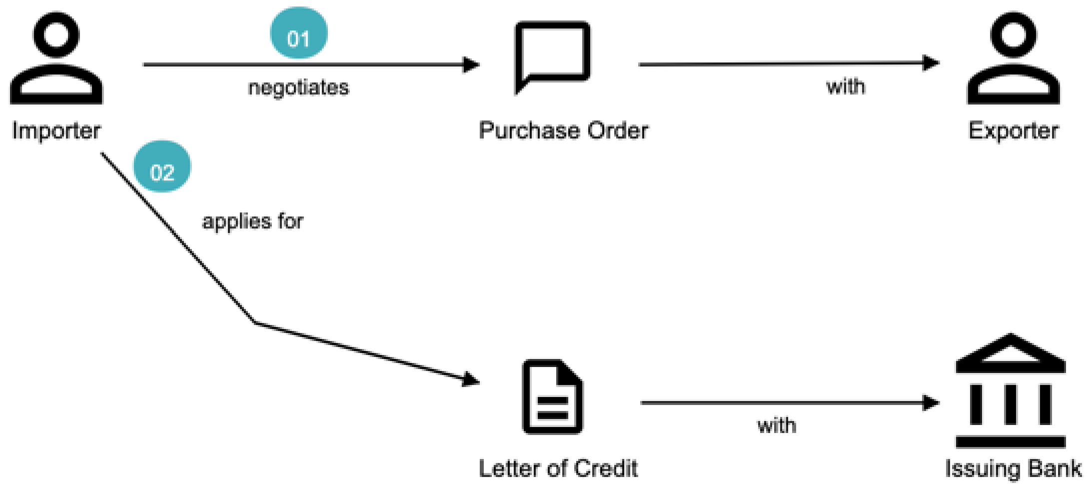


Figure 1- 16. Introducing the LC and the issuing bank

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."*

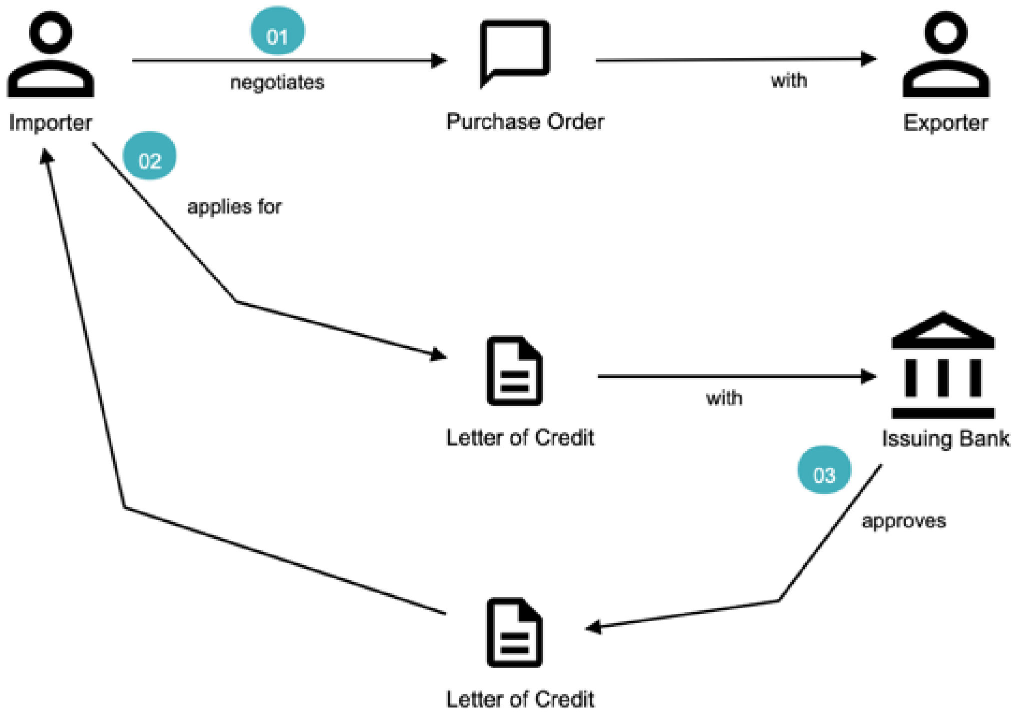


Figure 1- 17. Notifying LC approval to the importer

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**."*

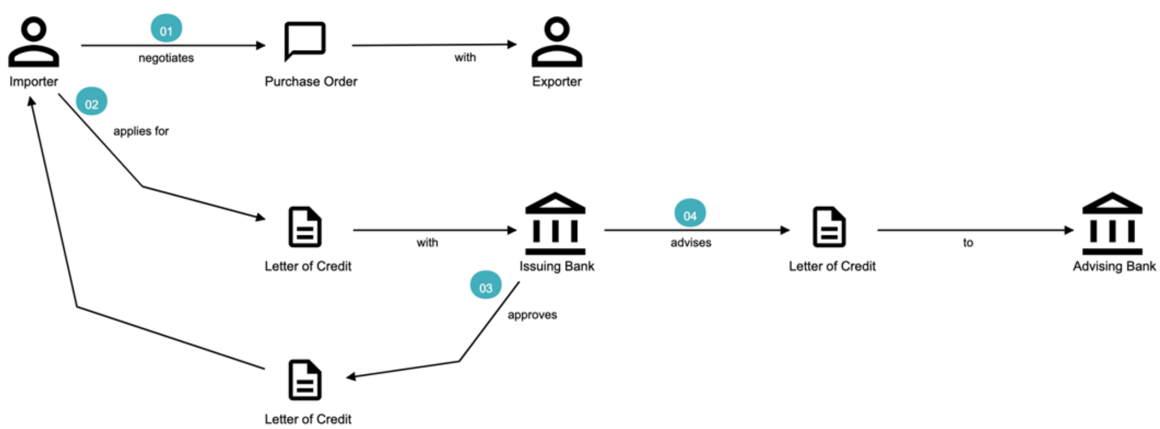


Figure 1- 18. Introducing the advising bank

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."*

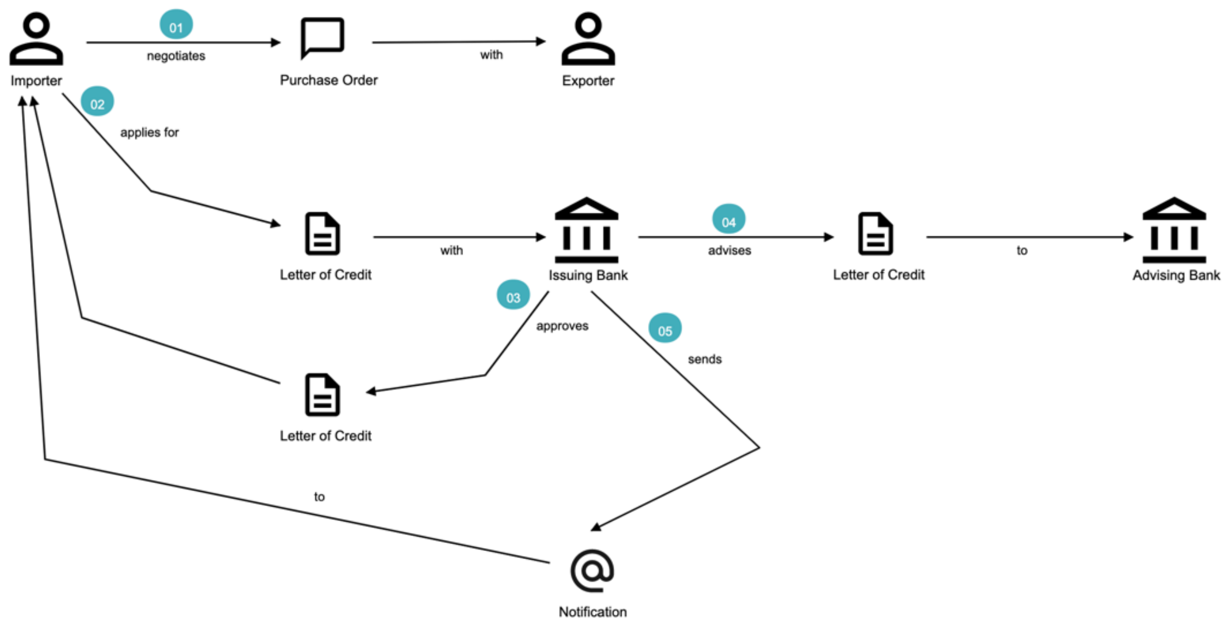


Figure 1- 19. Advice notification to the importer

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

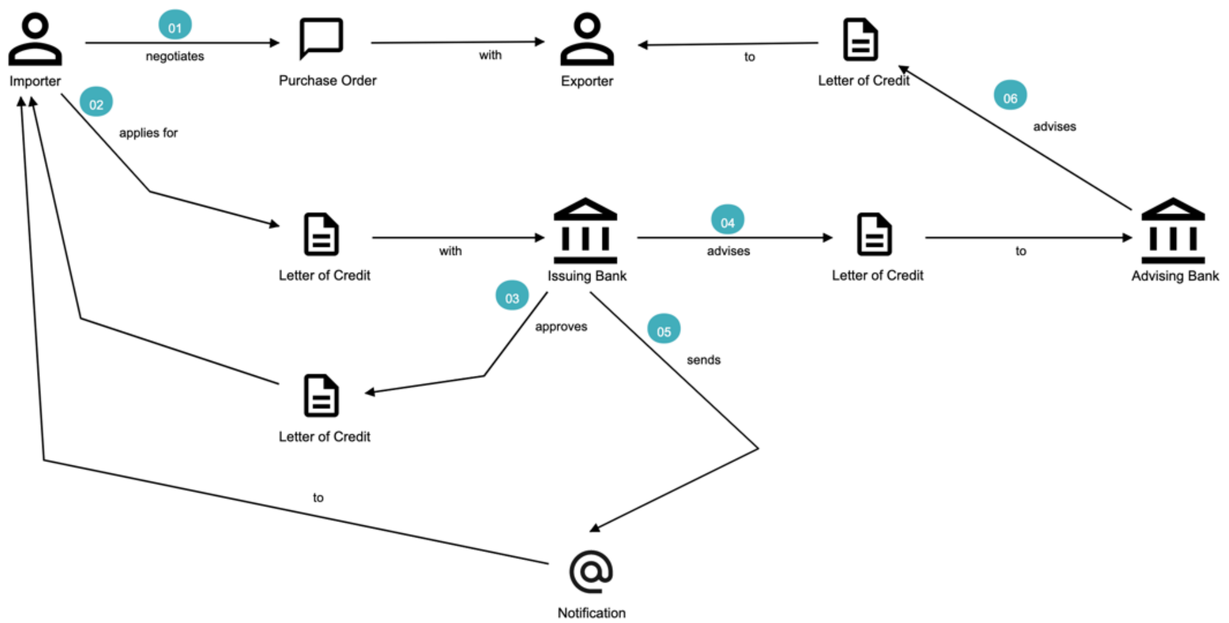


Figure 1- 20. Dispatching the advice to the exporter

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.

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

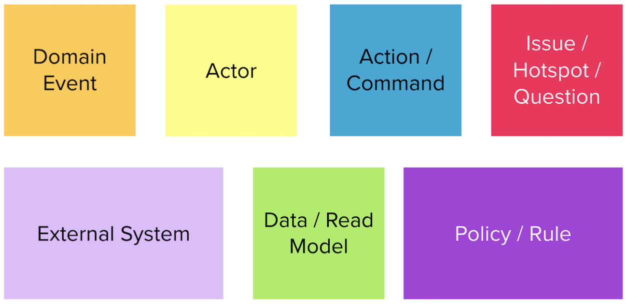


Figure 1- 21. EventStorming legend

> **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.

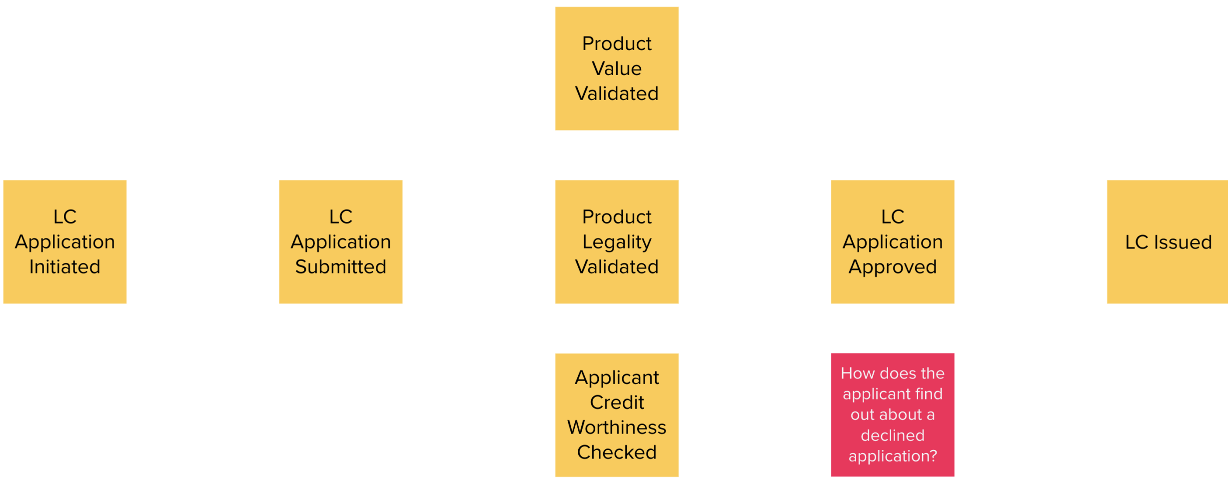


Figure 1- 22. Event chronology

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:

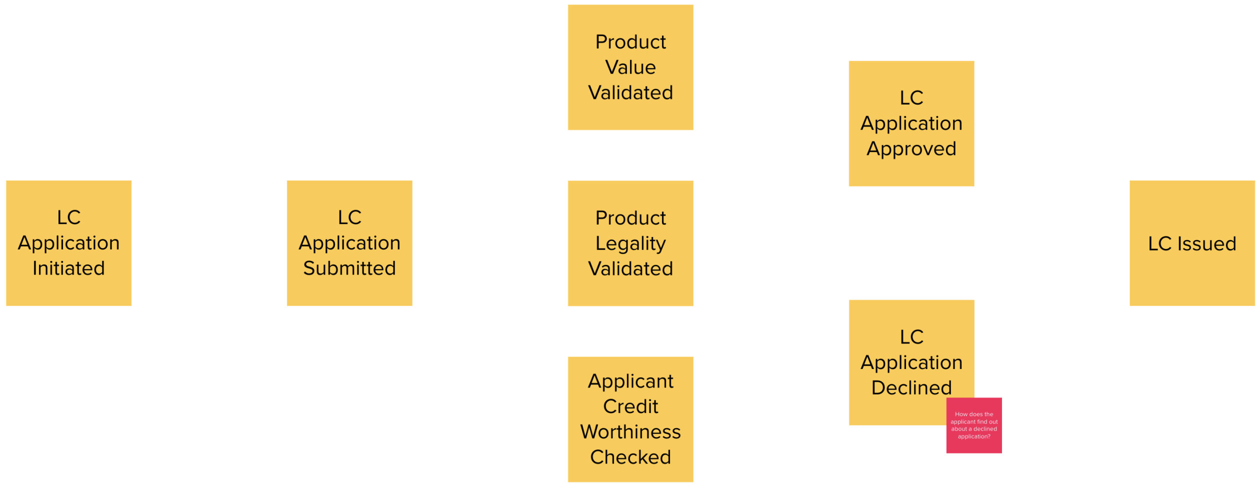


Figure 1- 23. New event to handle declined applications

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).

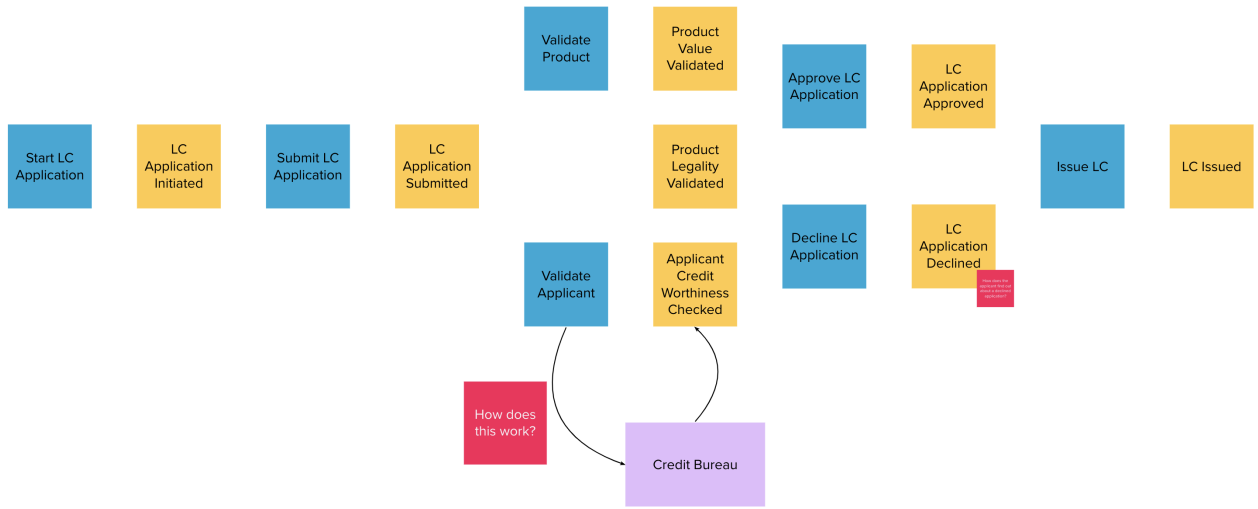


Figure 1- 24. Activities and external systems

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).

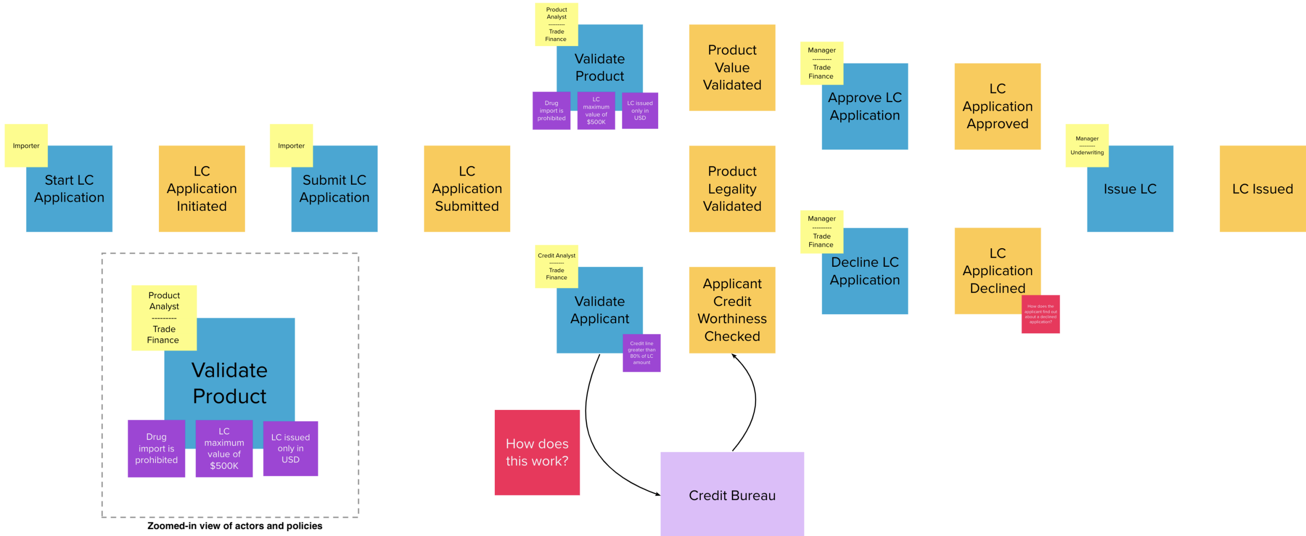


Figure 1- 25. Users and policies

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).

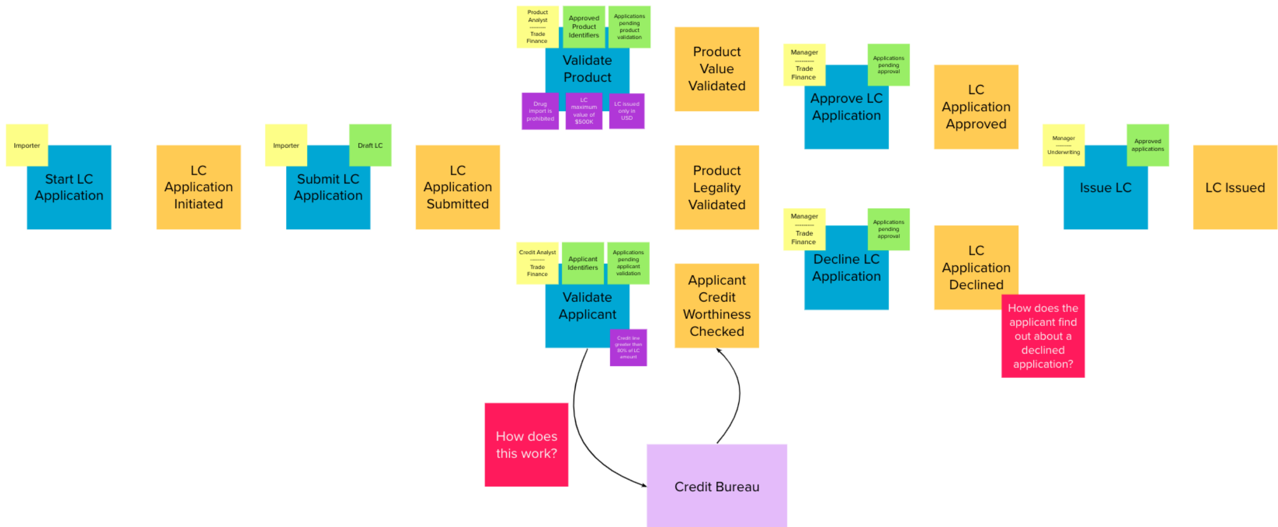


Figure 1- 26. Big picture eventstorming workshop board

> 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:

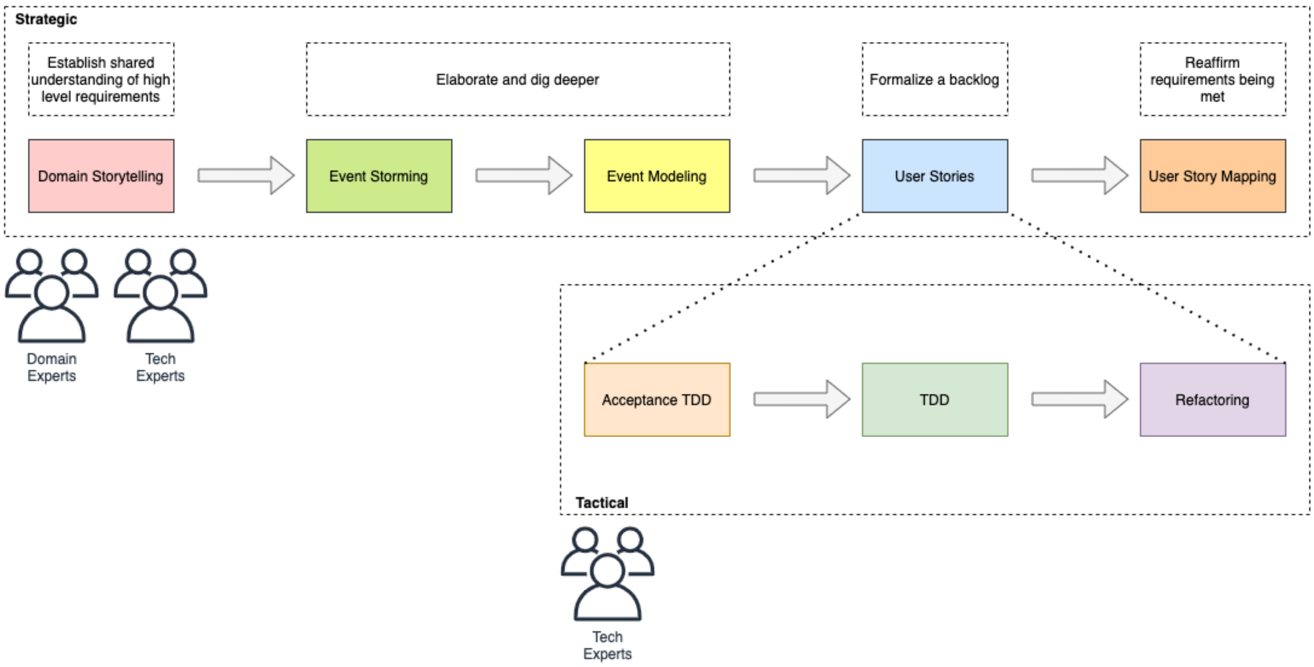


Figure 1- 27. Domain requirements elaboration continuum

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:

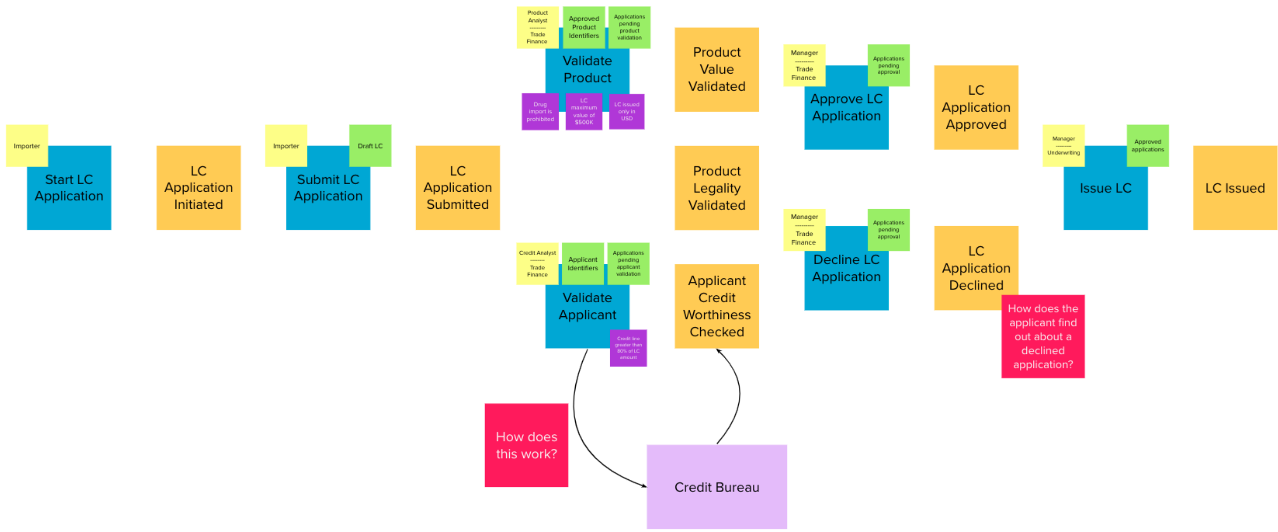


Figure 1- 28. Recap of eventstorming session

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:

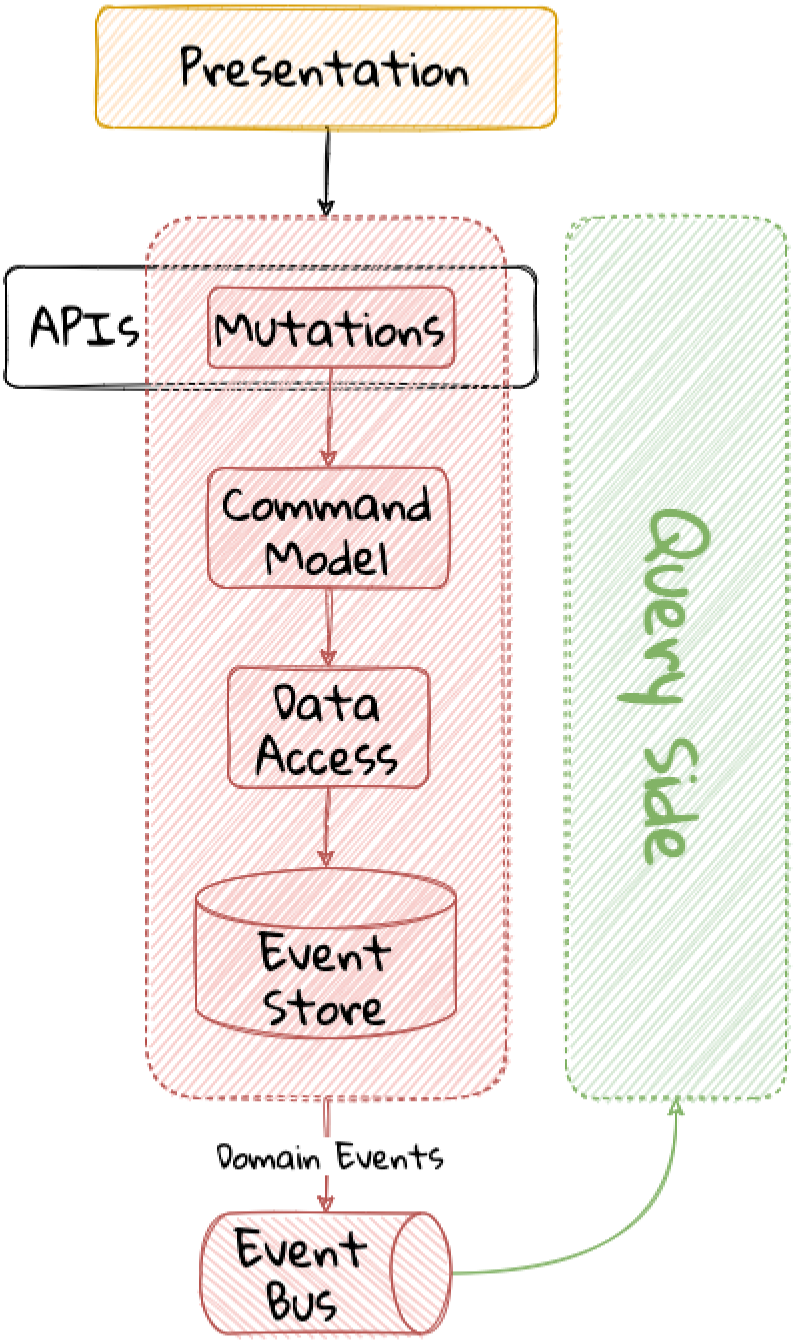


Figure 1- 29. CQRS application — command side

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[1], 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[2] 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[3] and Eventuate[4] 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:

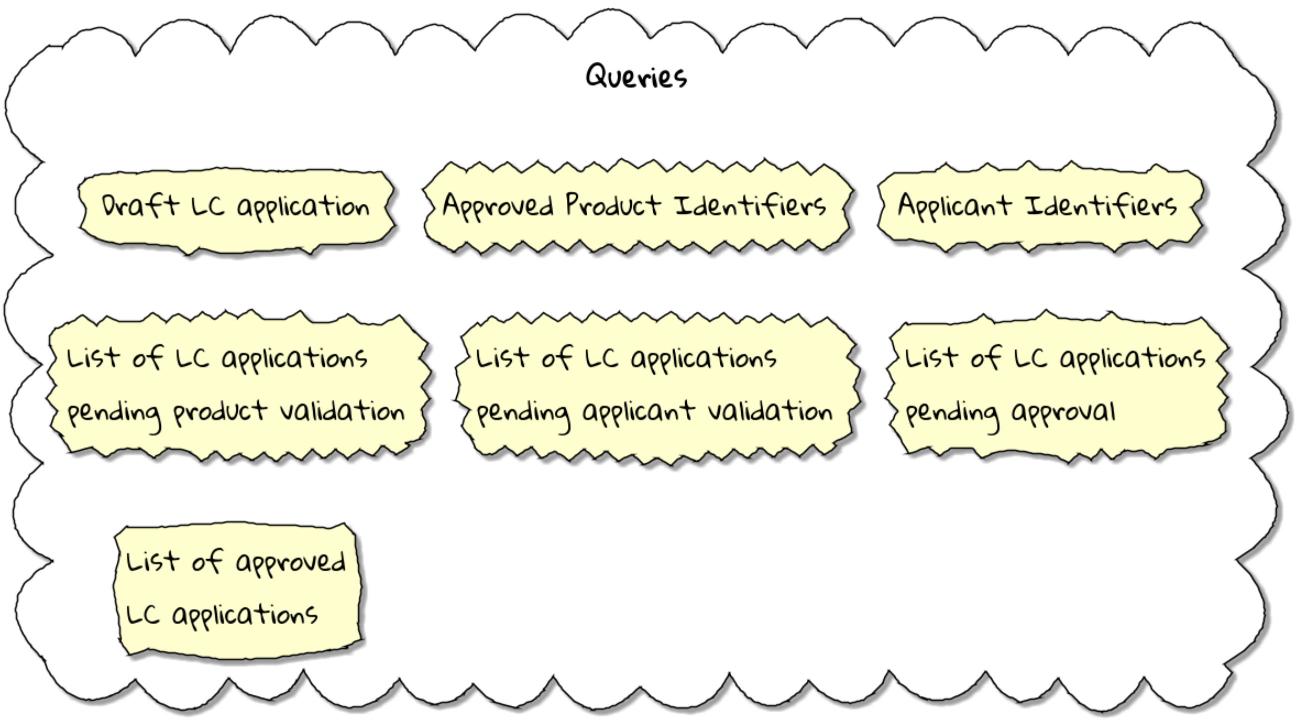


Figure 1- 30. Identified commands

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:

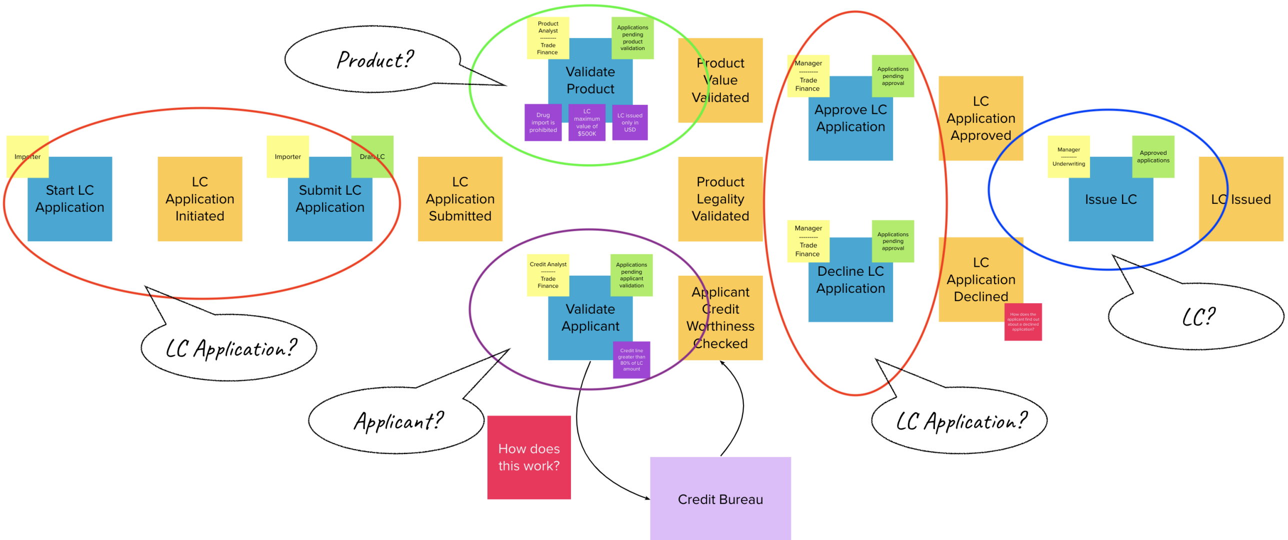


Figure 1- 31. First cut attempt at aggregate design

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:

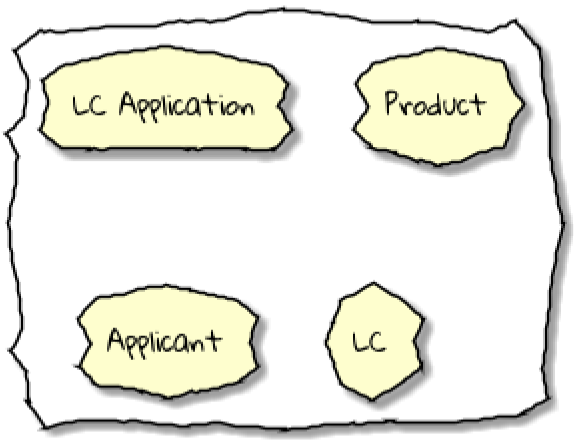
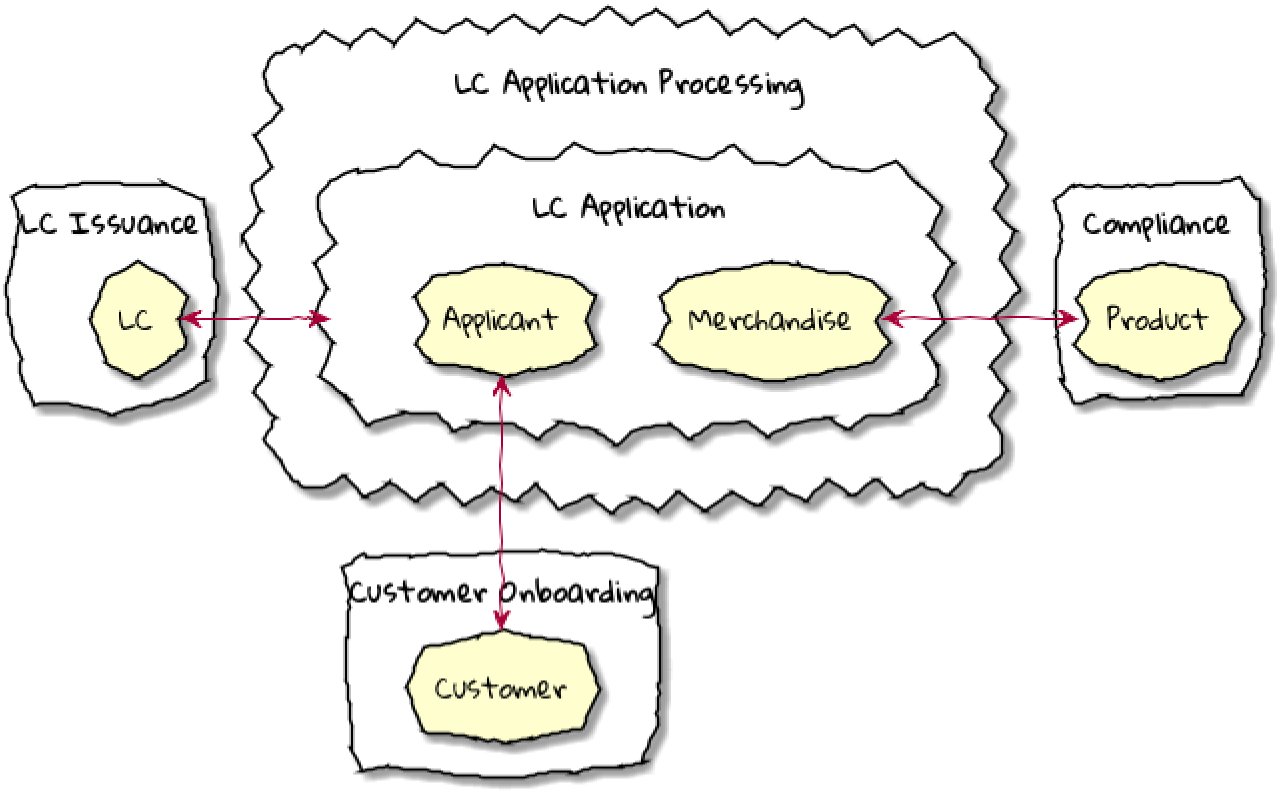


Figure 1- 32. Potential aggregates at first glance

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:

Unresolved directive in 05-implementing-domain-logic.adoc - include::images/context-map.adoc[]

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:

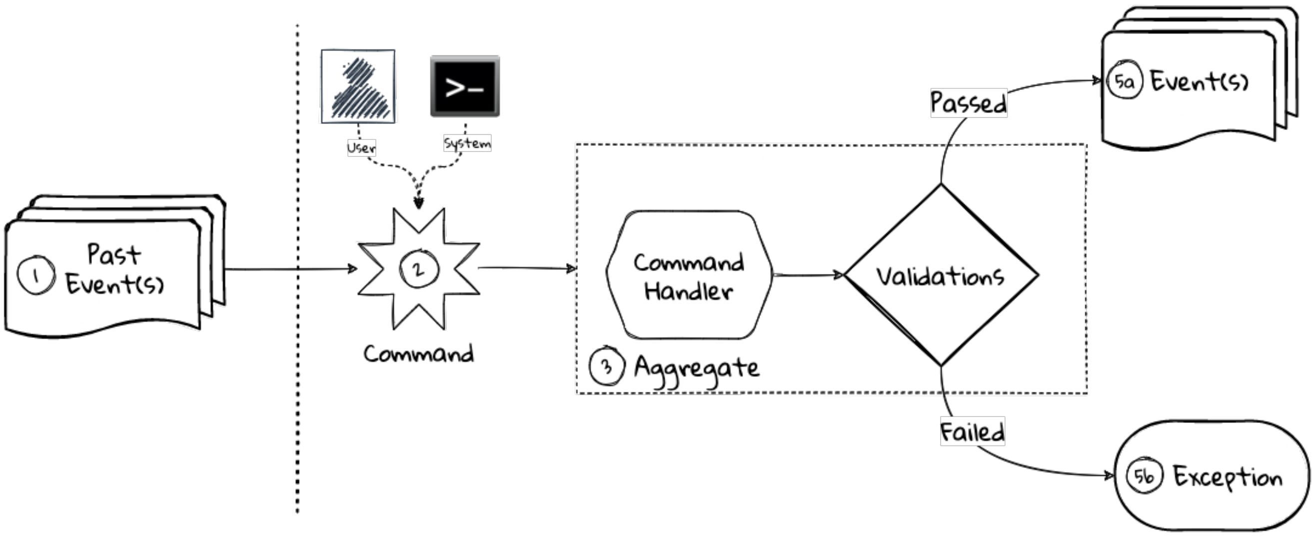


Figure 1- 33. An event-driven system

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.

```java
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:

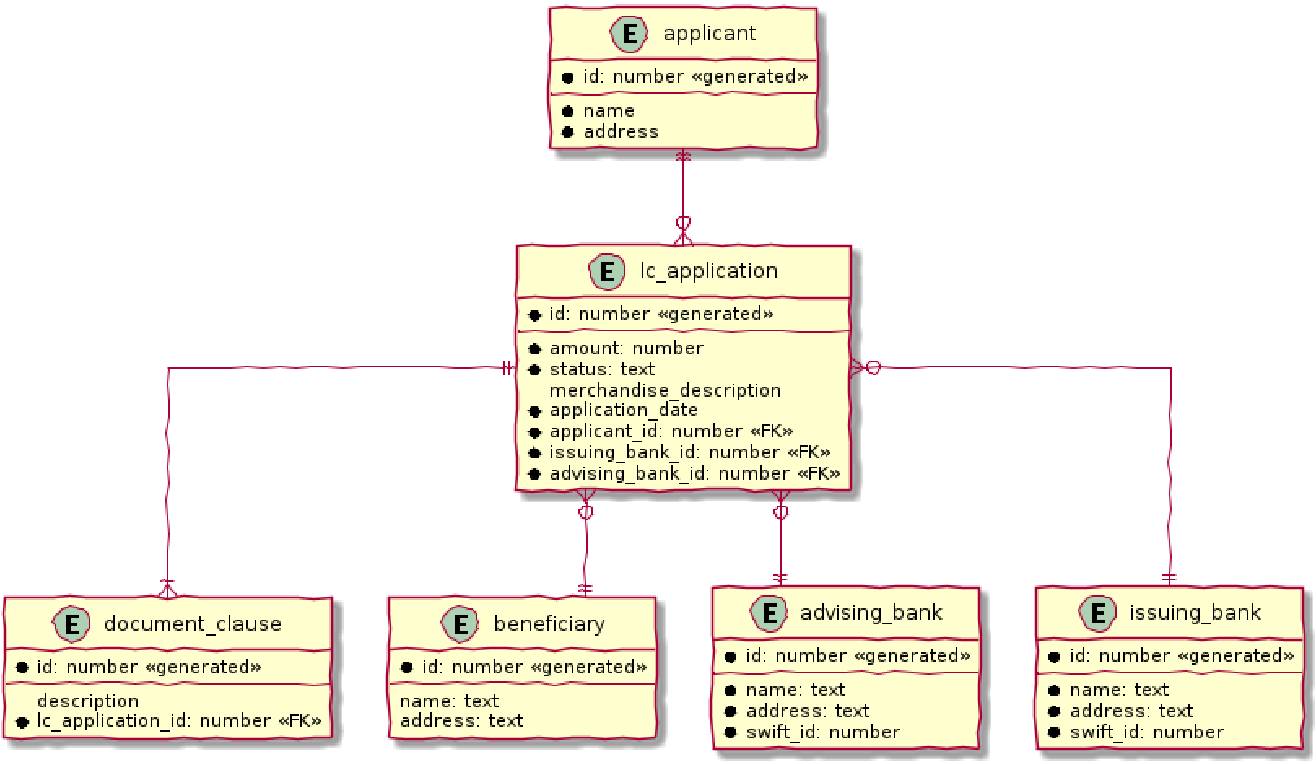


Figure 1- 34. Typical entity relationship model

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:.

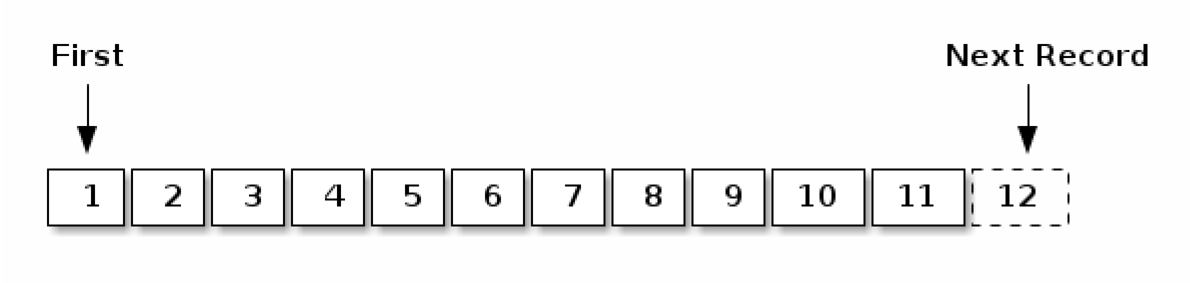


Figure 1- 35. The log data structure

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[5] and Axon Server[6] 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:

```java
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:

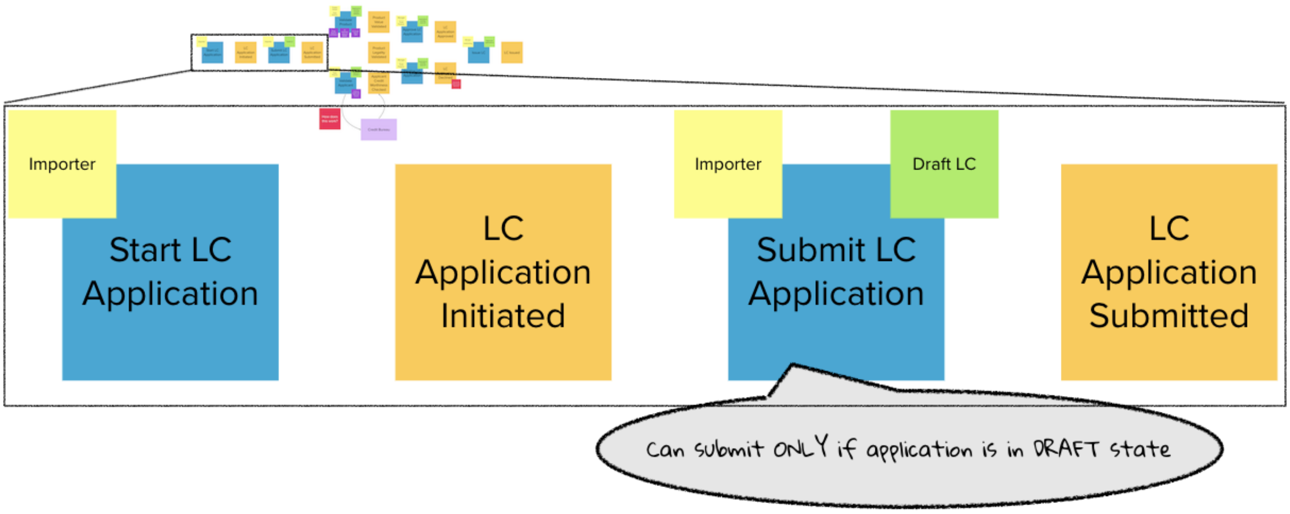


Figure 1- 36. Validations during the submit LC process

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:

```java
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:

```java
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:

```java
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[7].

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

```java
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 3.8.1 to list down the advantages of the approach versus the traditional approach. Share the reasoning with your teammates.

[1] https://ordina-jworks.github.io/domain-driven%20design/2016/02/02/A-Decade-Of-DDD-CQRS-And-Event-Sourcing.html

[2] http://axonframework.org/

[3] https://www.lagomframework.com/

[4] https://eventuate.io/

[5] https://www.eventstore.com/

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

[7] 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[8] 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:

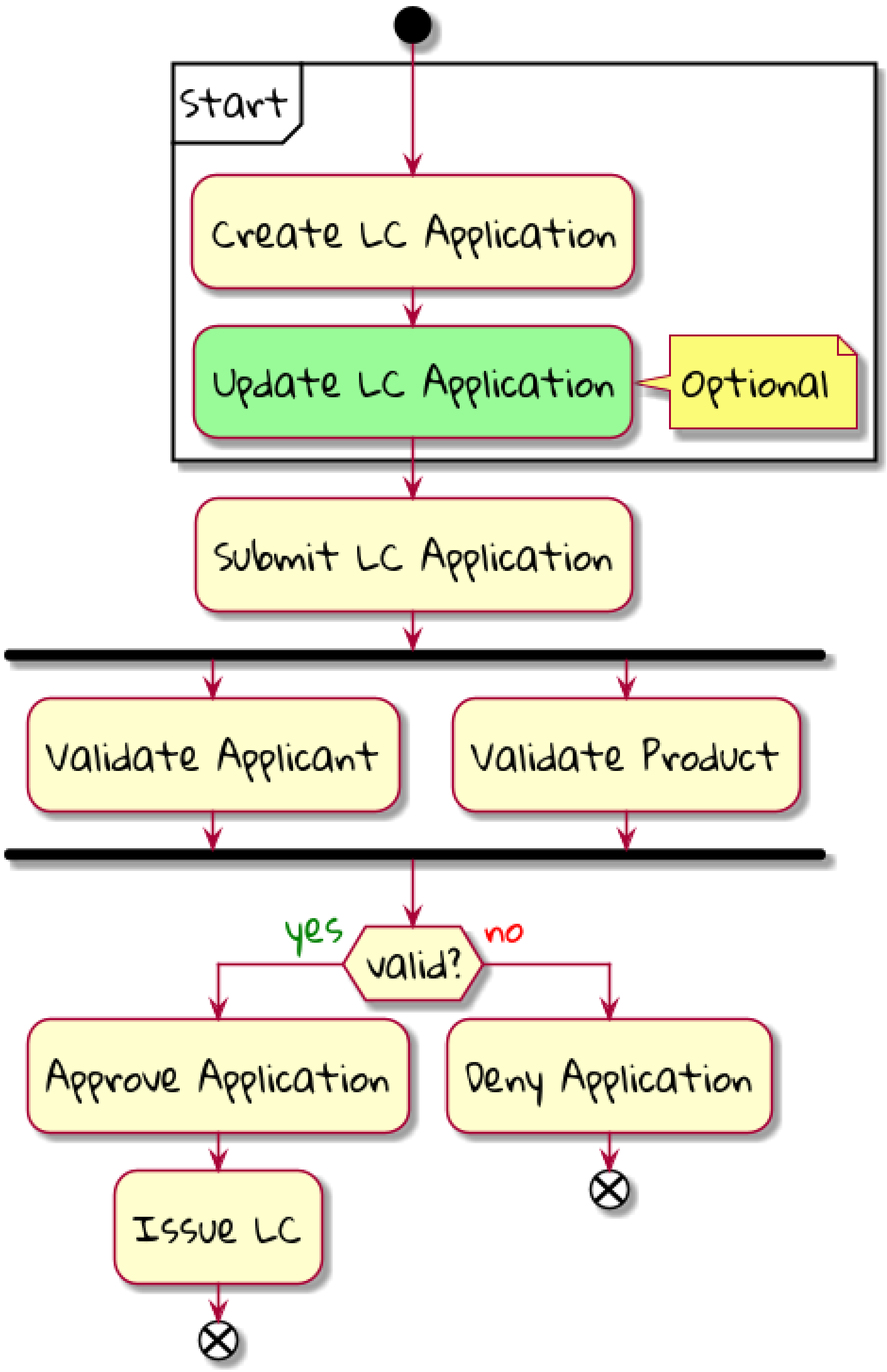


Figure 1- 37. Commands from the event storming session

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:

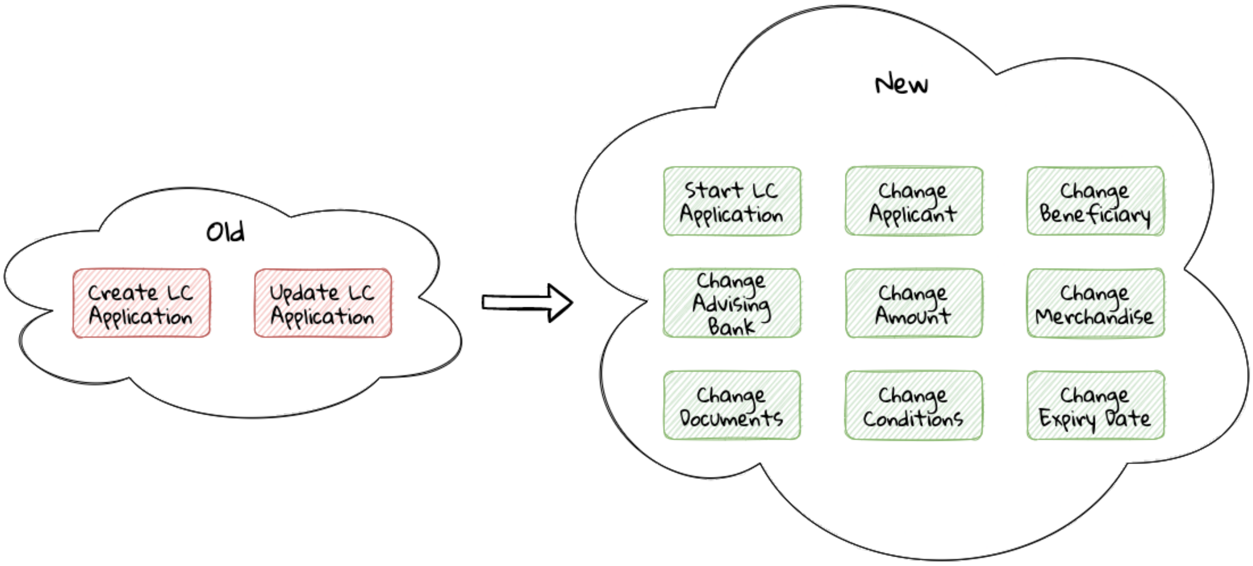


Figure 1- 38. Decomposed commands

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:

```java
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:

```java
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:

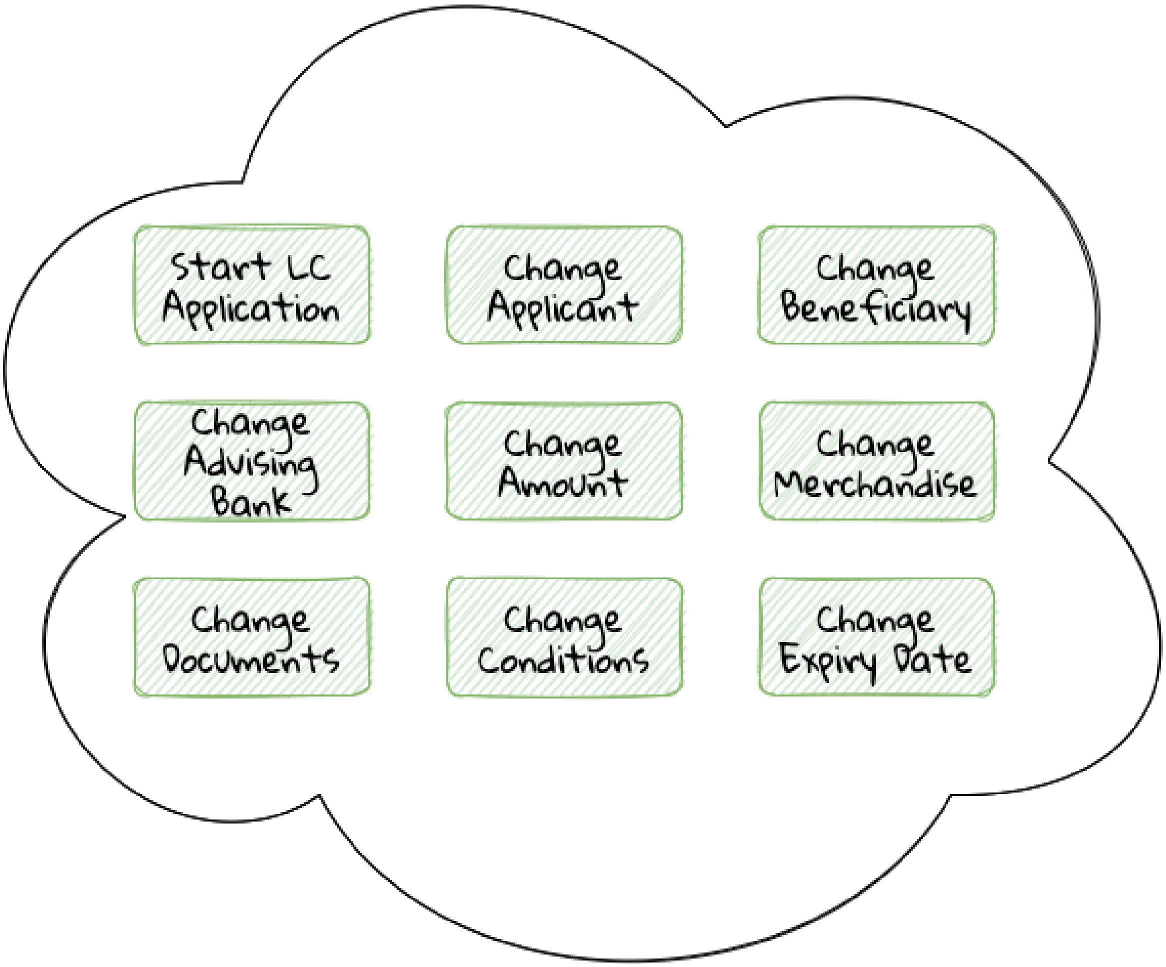


Figure 1- 39. Revised commands

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:

```java
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 {
    // ...
}
```

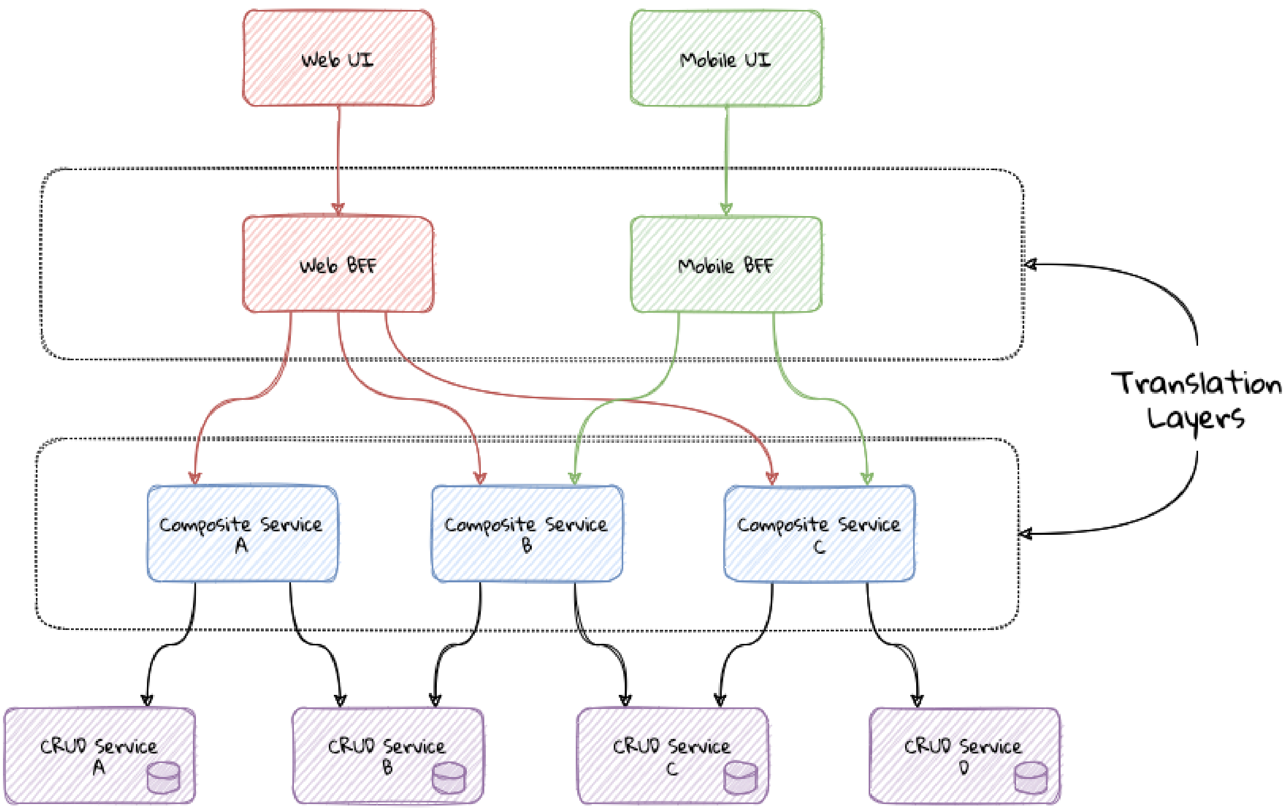
In our current design, the implications of this change are far reaching for both the provider and the consumer(s). Let's look at some of the consequences in more detail:

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*[9].
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[10]) 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:

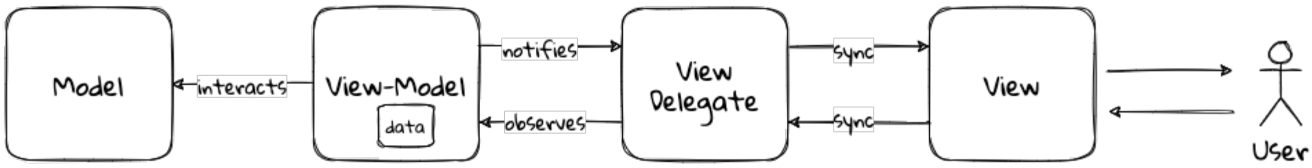


Figure 1- 40. MVVM design pattern

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:

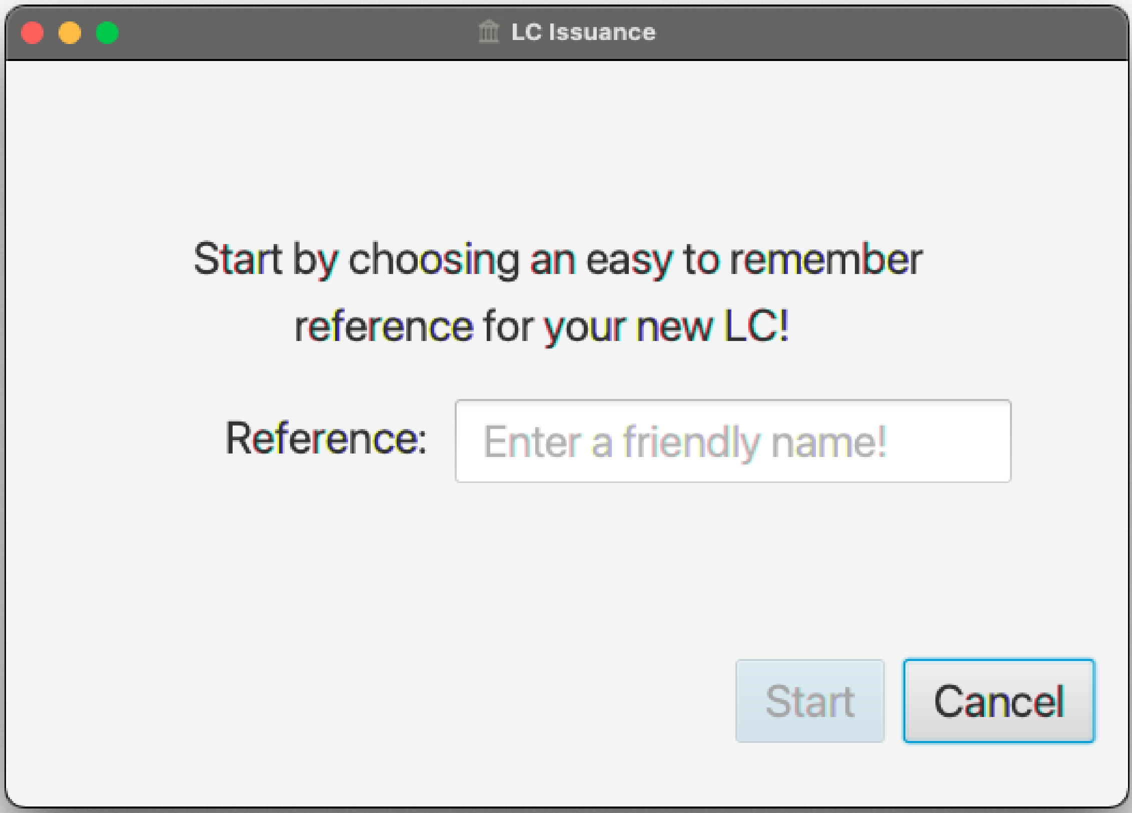


Figure 1- 41. Start LC creation screen

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[11]. Ted Neward also talks about this in his blog titled *The Fallacies of Enterprise Computing*[12].

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:

```java
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:

```java
@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:

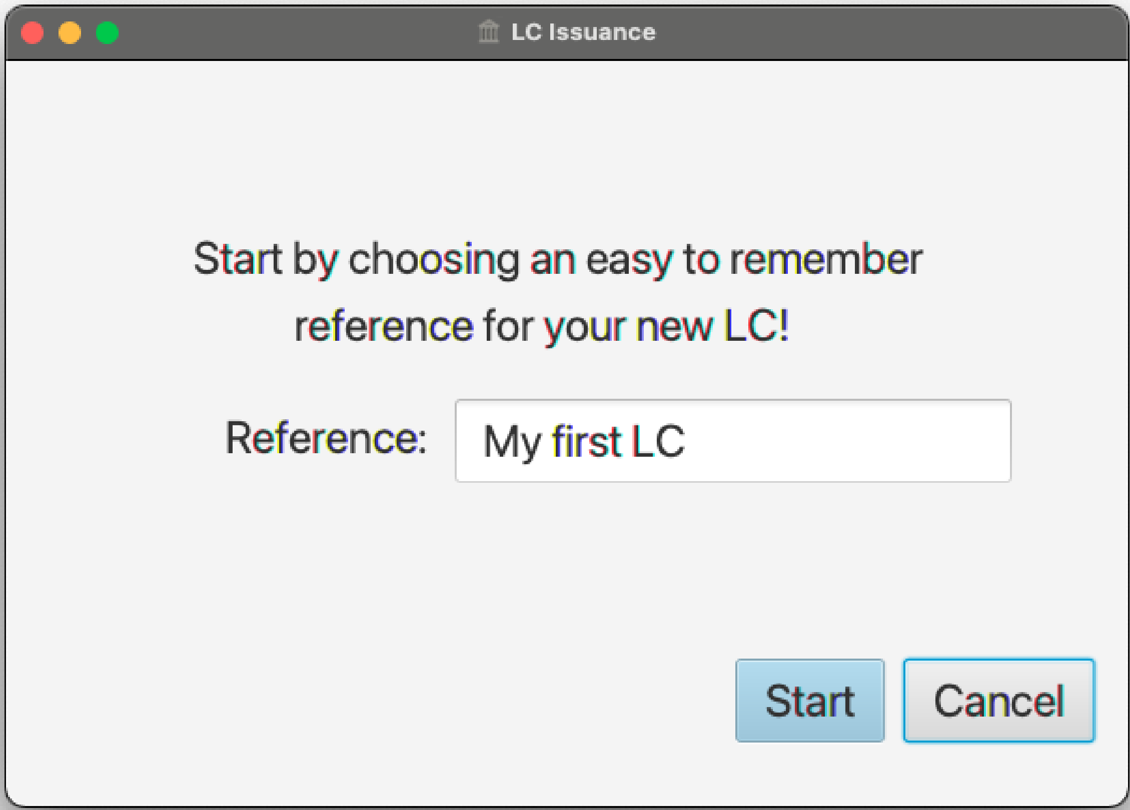


Figure 1- 42. The start button is enabled with a valid client reference

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- 39 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- 41. The revised `StartNewLCApplicationCommand` looks as shown here:

```java
@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:

```java
@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:

```java
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:

```java
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:

```java
@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:

```java
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:

```java
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

[8] https://openjfx.com/

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

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

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

[12] 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:

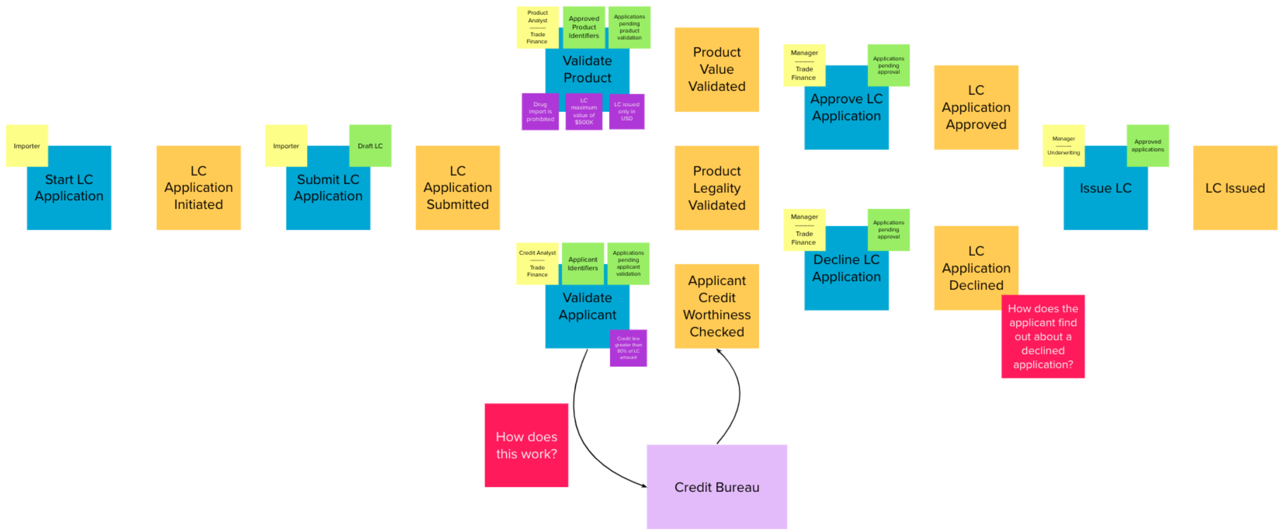


Figure 1- 43. Recap of eventstorming session

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:

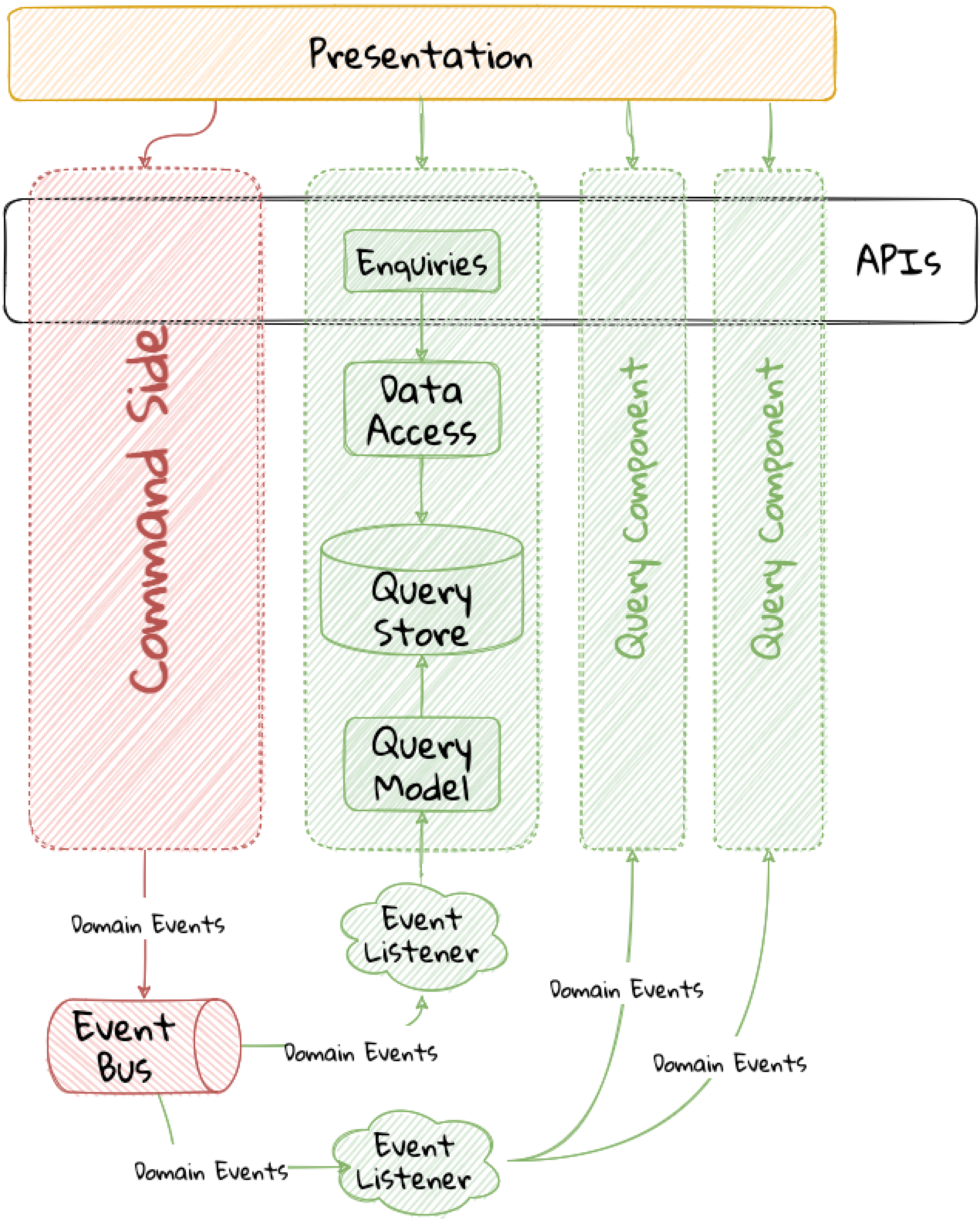


Figure 1- 44. CQRS application — query side

> 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 :

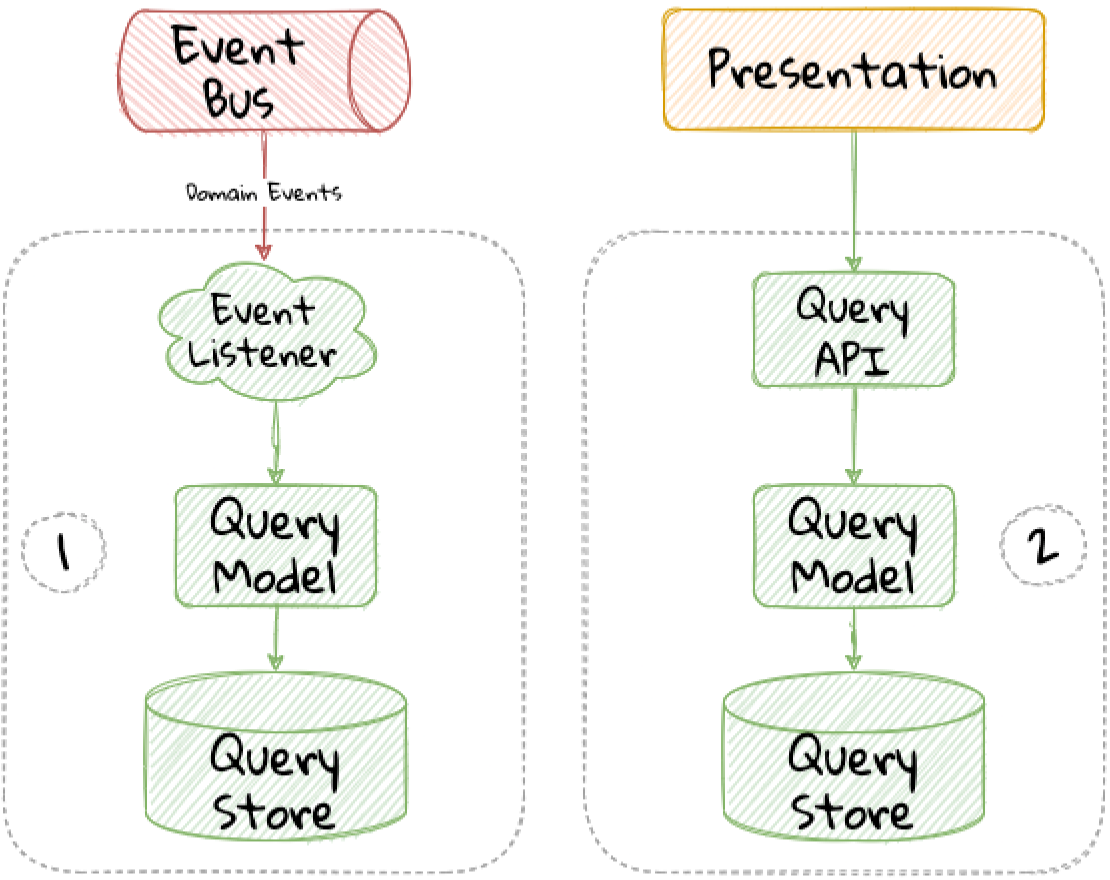


Figure 1- 45. Query side dissected

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:

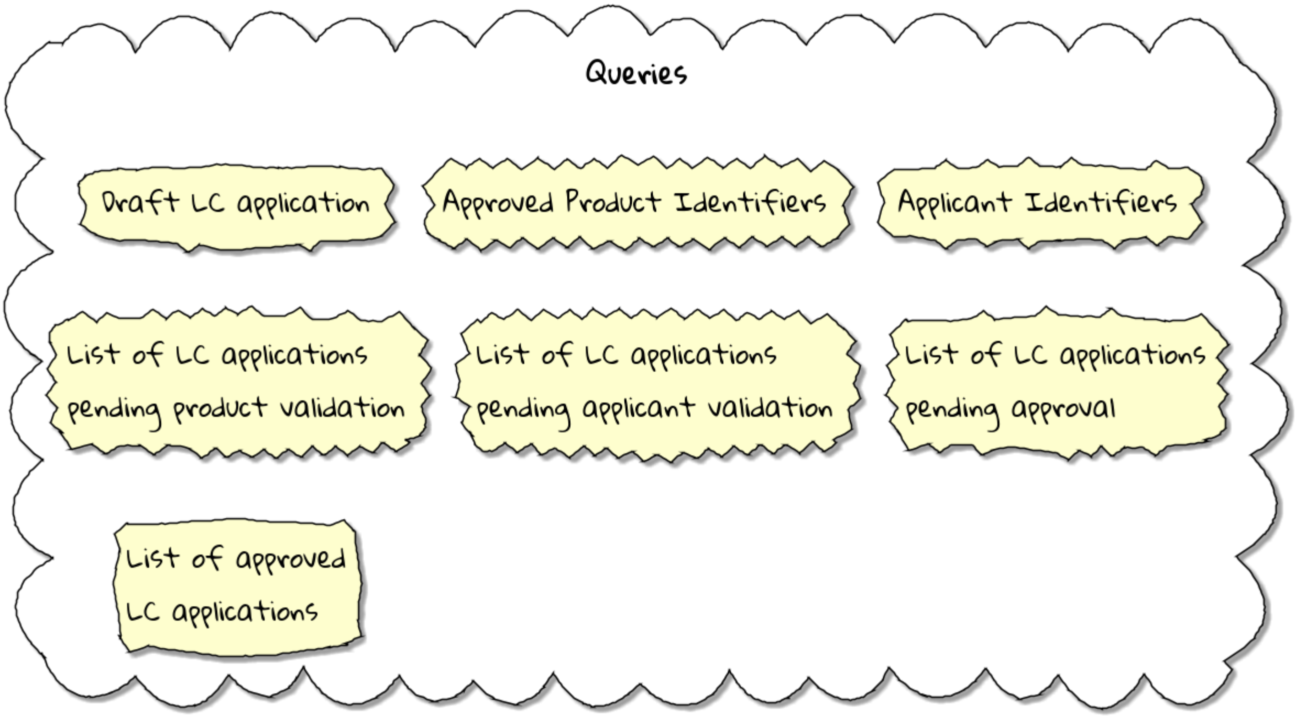


Figure 1- 46. Identified queries

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:

```java
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:

```java
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[13] 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[14] 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 sub-domain. 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[15] and EventStoreDB[16] 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[17], 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[18] (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:

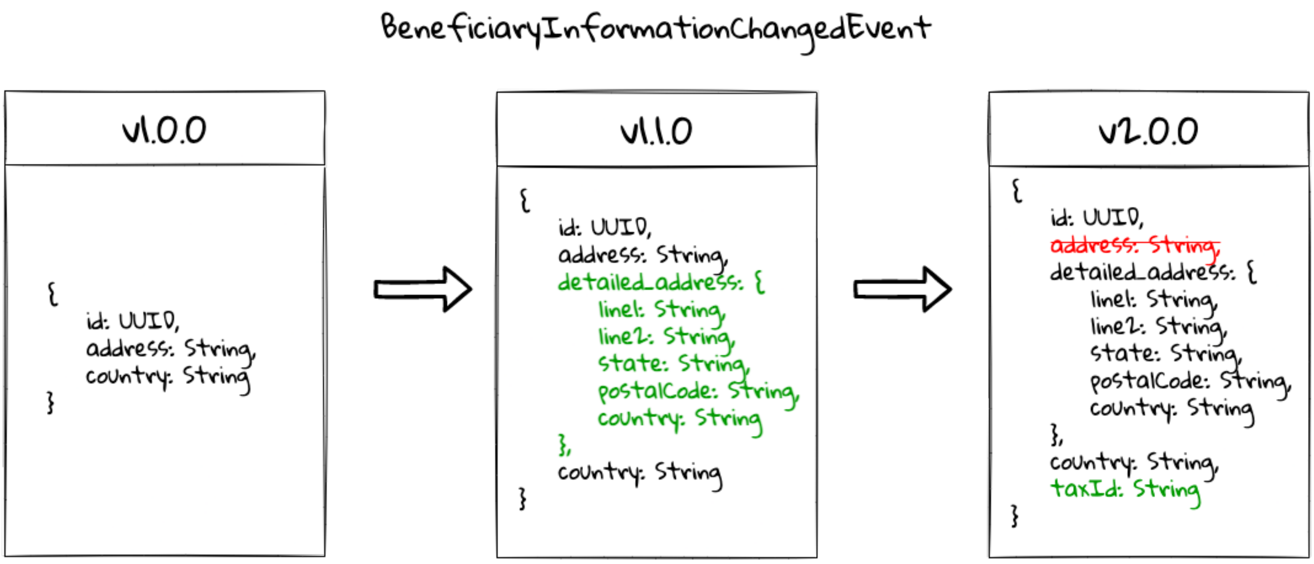


Figure 1- 47. Event evolution

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**[19] 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?

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

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

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

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

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

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

[19] 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:

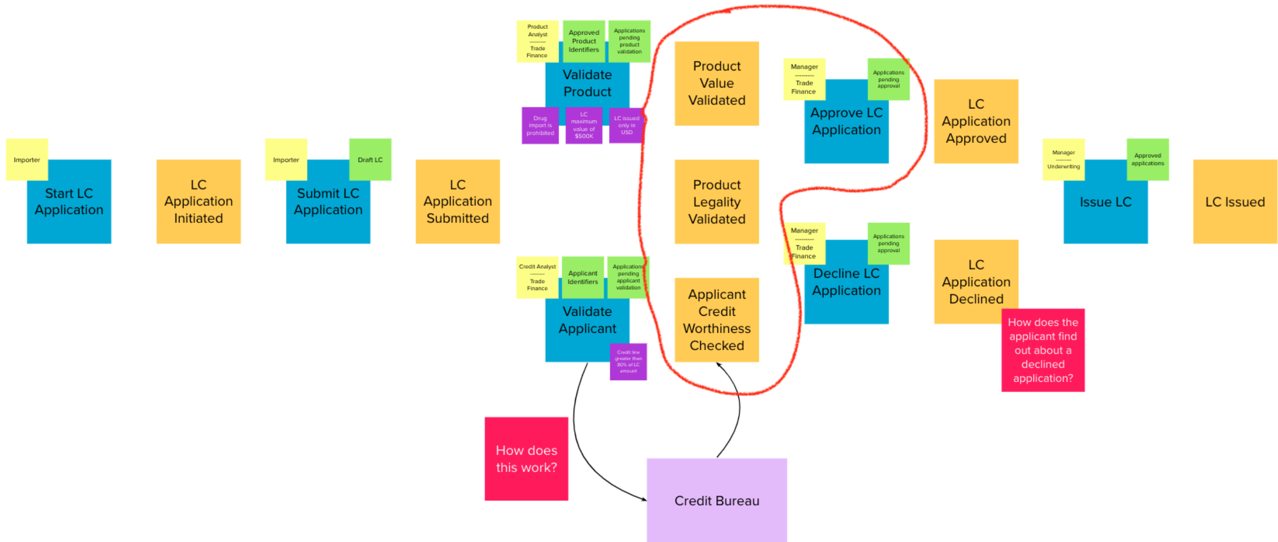


Figure 1- 48. Recap of eventstorming session

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**[20]. 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:

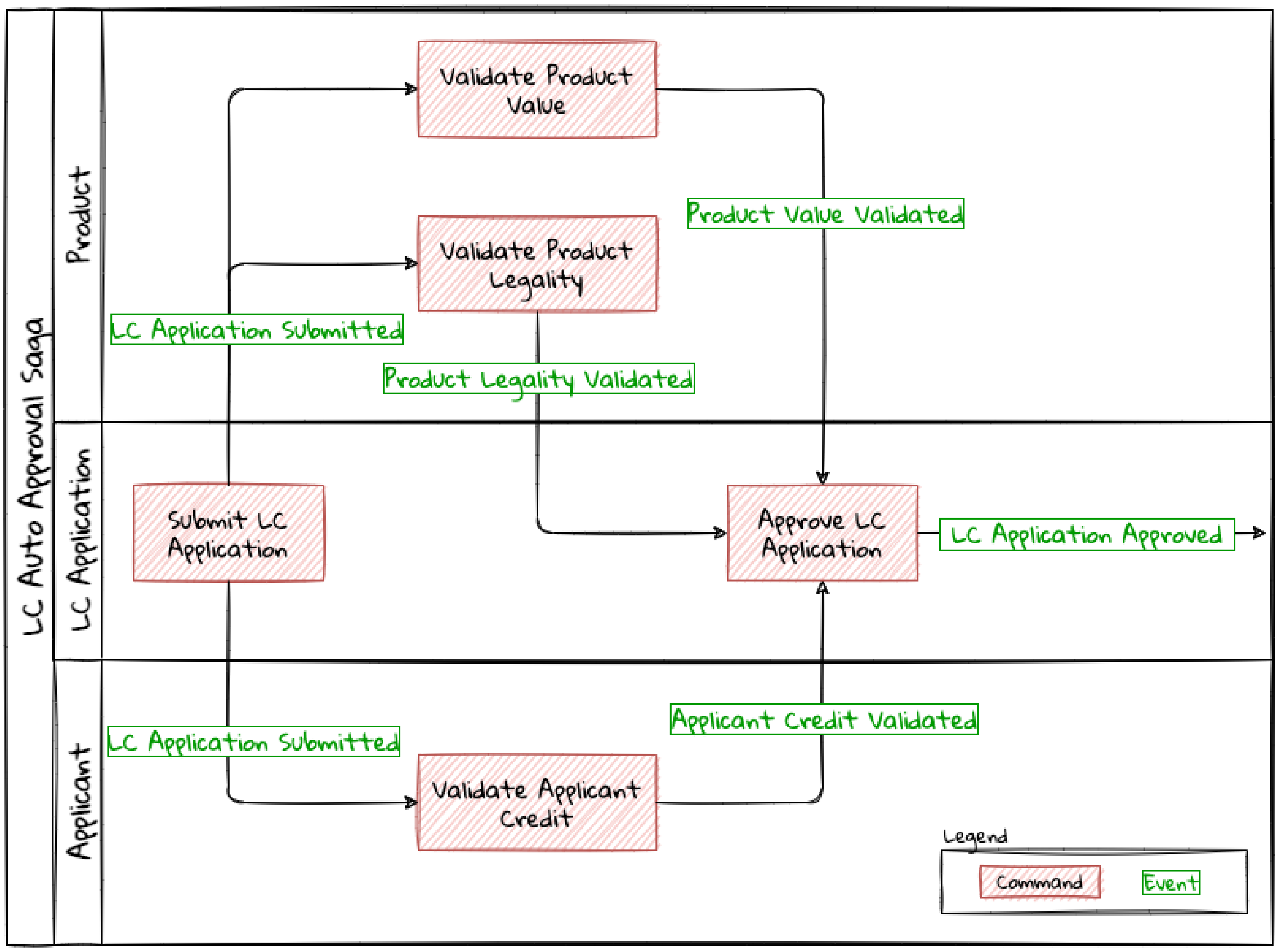


Figure 1- 49. Auto-approval process — logical view

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:

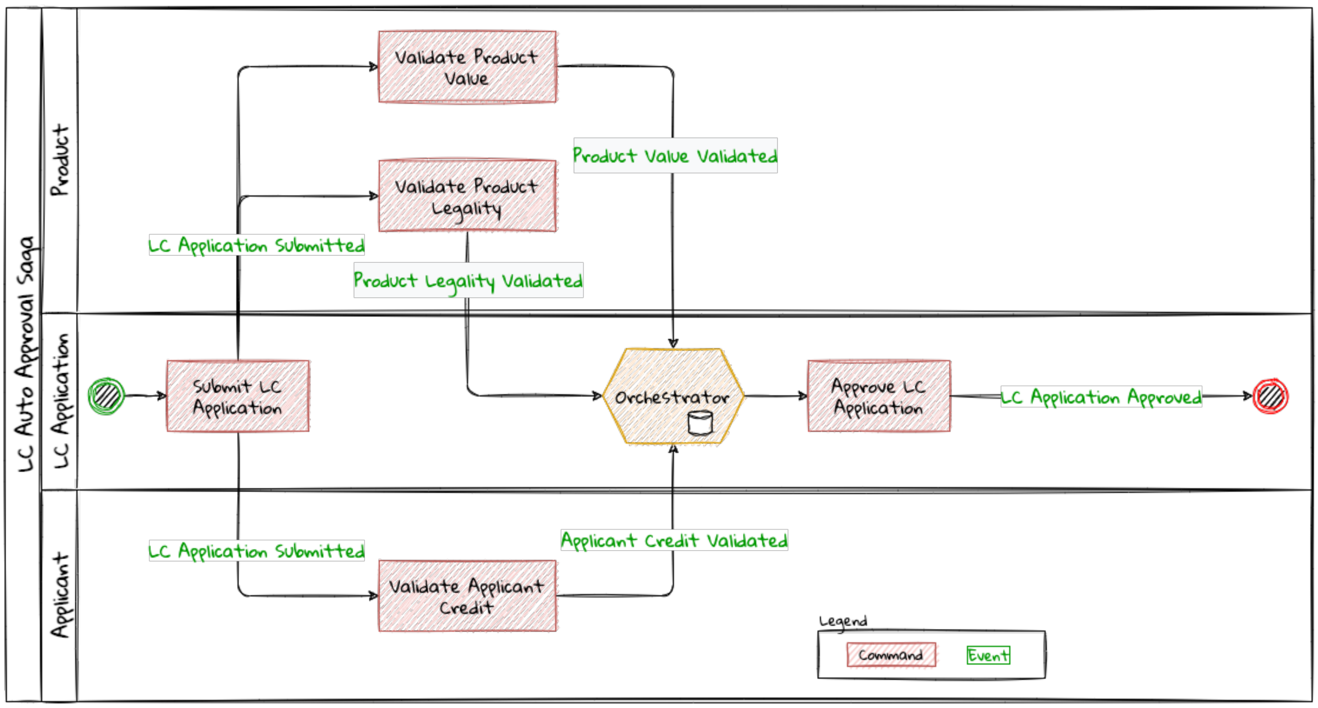


Figure 1- 50. Saga implementation using an orchestrator

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:

```java
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:

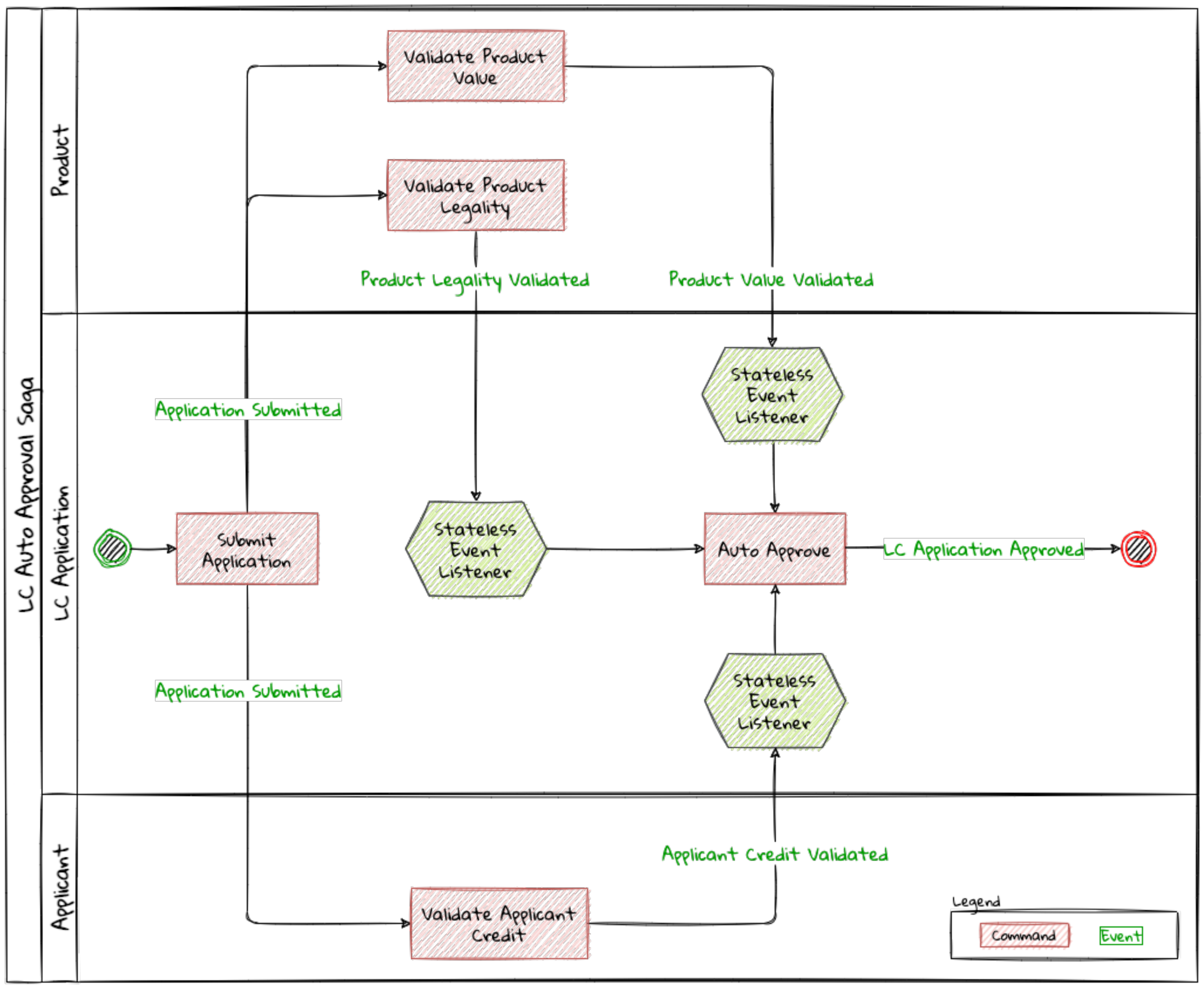


Figure 1- 51. Saga implementation using choreography

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*[21]. 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:

```java
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:

```java
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:

```java
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

[20] https://en.wikipedia.org/wiki/state_machine

[21] https://en.wikipedia.org/wiki/Distributed_transaction

Chapter 9. Integrating with External Systems (15 pages)

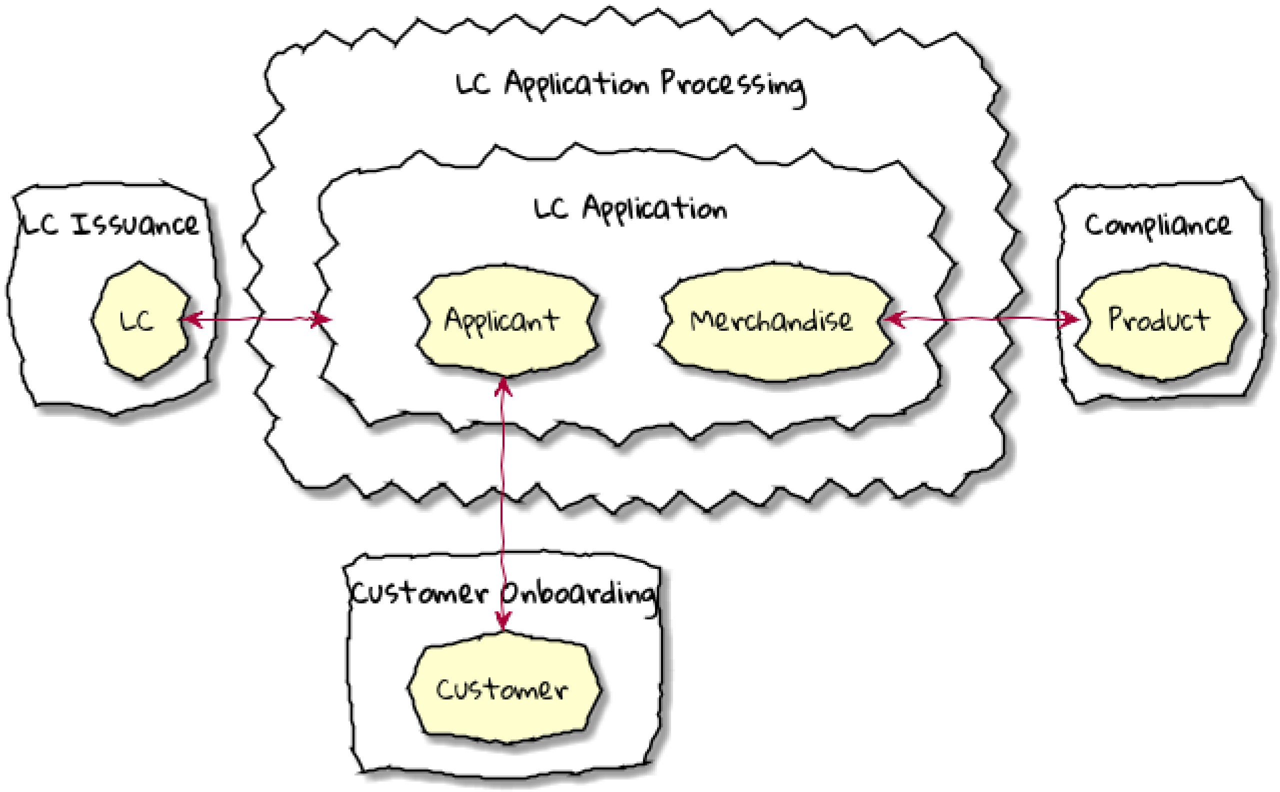
> Wholeness is not achieved by cutting off a portion of one's being, but by integration of the contraries.
>
> — Carl Jung

Thus far, we have used DDD to implement a robust core for our application. However, most bounded contexts usually have both upstream and downstream dependencies which usually change at a pace which is different from these core components. To maintain both agility and reliability and enable loose coupling, it is important to create what DDD calls the anti-corruption layer in order to shield the core from everything that surrounds it. In this chapter, we will look at integrating with a legacy Inventory Management system. We will round off by looking at common patterns when integrating with legacy applications.

9.1. Technical Requirements

9.2. Continuing our design journey

From our eventstorming session, we have arrived at four bounded contexts for our application as depicted here:



[lc application context map] | *lc-application-context-map.png*

Figure 1- 52. A simple context map for the LC application

9.3. Integration mechanisms

9.3.1. Symmetric relationship patterns

Partnership

Shared kernel

Separate ways

9.3.2. Asymmetric relationship patterns

Conformist

Anti-corruption layer

Open host service

9.4. Implementation patterns

9.4.1. Data-based

9.4.2. API-based

HTTP-based APIs

Message-based APIs

9.4.3. Shared code artifacts

9.4.4. Enforcing contracts

9.5. Legacy Application Migration Patterns

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 Microservices (15 pages)

> 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

We now have a working application which is bundled as a single package. In this chapter, we will distribute the UI, the command side, the query side and the saga components into distinct components. We will also look at how to test the system as a whole using the excellent testcontainers library.

10.1. Right Sizing Components

10.2. Maintaining Autonomy

10.3. Understanding the Costs of Distribution

10.4. Handling exceptions

> 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

10.4.1. Recovery

Automated recovery

Manual recovery

Compensating actions

Retries

10.5. Testing the Overall System

Chapter 11. Non-Functional Requirements (25 pages)

> Sometimes I feel like I am being forgotten.
>
> — Anonymous

While 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.

11.1. Dealing With Eventual Consistency

11.2. Scaling the Event Store with Snapshots

11.3. Event Versioning and Upcasting

11.4. Monitoring, Metrics and Tracing

11.5. Enhancing Performance

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