INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Supporting Migration to Microservices with Domain Driven Design

ANDRÉ MIGUEL SANTOS AZEVEDO

Outubro de 2019

POLITÉCNICO
DO PORTO

Supporting Migration to Microservices with Domain Driven Design

André Miguel Santos Azevedo

Dissertation to obtain the degree of Master in Informatics Engineering, Specialization Area of Software Engineering

Advisor: Isabel Azevedo

Jury:

President:

[President's Name, DEI, ISEP]

Members:

[Member Name, DEI, ISEP]

[Member Name, DEI, ISEP]

Porto, October 2019

Dedicatória

“Dedico o meu trabalho a todos os que me suportaram desde família, professores e colegas.”

Resumo

Uma empresa *e-commerce* com um *marketplace* disponível para todo o mundo tem crescido significativamente durante os anos, com mais clientes interessados e um aumento nas vendas. Com isto, o *software* cresceu com a tendência, numa arquitetura a nível monolítica.

A área de processamento de encomendas foi a que se mais sentiu este aumento, com um serviço monolítico que integra múltiplas equipas de desenvolvimento com âmbitos de negócio abrangentes. A equipa apelidada de “ST” é uma das participantes, tendo como responsabilidades, a implementação e manutenção do *software* para Ocorrências em Encomendas e Devoluções.

No entanto, para a equipa “ST” é cada vez mais complicado realizar alterações no seu *software*, visto que o domínio não é mais flexível, sendo uma tarefa difícil para os programadores, a sua compreensão e a sua evolução para novos requisitos de negócio.

Aproveitando a tendência de evolução existente na plataforma, a equipa “ST” realizará uma migração do *software* de Ocorrências para uma arquitetura orientada a microserviços, com auxílio de *Domain Driven Design*, definindo os contextos limitados, desenhando os modelos de domínio, e implementando os respetivos serviços com a utilização de boas práticas de engenharia, como padrões de *software.*

No final, os programadores da equipa “ST” com o auxílio dos seus conhecimentos, serão os responsáveis por fazer uma avaliação final da solução, de forma a perceber se esta mitigou os constrangimentos que causavam o problema enunciado.

Palavras-chave: *e-commerce*, microserviços, *Domain Driven Design*, padrões de *software*

Abstract

An e-commerce company with a marketplace available for all over the world has grown significantly over the years, having more interested customers increasing its sales. With this, software tended to expand in a monolithic way.

The order processing is one of the areas that grew the most, with a monolithic service that integrates multiple development teams with business broad scopes. The team called “ST” is one of the participants, having the responsibility of implementing and maintain the software related to Order and Return Occurrences.

However, for the “ST” team developers is increasingly difficult to change its software, since the domain is not flexible anymore, being a tough task for developers, its understanding and its evolution for new business requirements.

Taking advantage of the evolution trend, existent on the platform, the “ST” will migrate the Occurrences software to a microservices oriented architecture, with the Domain Driven Design support, delineating its bounded contexts, drawing its domain models, and implementing the corresponding services with usage of good engineering practices, such as software patterns.

Finally, the “ST” team developers, supported by their knowledge, are the responsible ones to evaluate the final implemented solution, in order to understand if it mitigates the constraints that caused the announced problem.

Keywords: e-commerce, microservices, Domain Driven Design, software patterns

Contents

Figures List

Tables List

Acronyms

Acronyms List

API	Application Programming Interface
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DDD	Domain Driven Design
DI	Dependency Injection
IDE	Integrated Development Environment
OOP	Object Oriented Programming
REST	Representational State Transfer
SQL	Structured Query Language
UI	User Interface

1 Introduction

This chapter introduces the project context and the problem to solve. In addition, the objectives for this project and the preconized approach to find a solution are presented. Finally, the chapter ends with a description of how this document is structured.

1.1 Context

This project happens in an e-commerce company, which has a marketplace available all over the world, selling products for many customers from its merchants. These merchants are also known as partners of the company.

The team in which the student integrates, hereinafter referred to as "ST", consists of five software engineers, one quality assurance element and one senior team leader. ST is responsible for producing and maintaining software for some functionalities associated with Order Processing, the ODS service, such as Stock Management, Item Packaging, and Occurrences. Occurrences (Section 2.1.4) are anomalies in Orders and Returns that merchants report, so these can be solved to unlock the Order Processing or Return Processing. After the reporting, the support teams are responsible to help and solve those occurrences.

ODS is monolithic (Section 2.2.3.1) service responsible for the entire business-related with orders, being constituted by multiple business blocks, such as Orders, Returns, Order Processing, and Return Processing, Order Occurrences and Return Occurrences, and many other businesses related both with Orders and Returns.

This service began to be developed five years ago, having one team responsible for its own but also having developments and maintenances from multiple teams.

ODS follows a monolithic architecture and it is divided by multiple layers with different responsibilities. Communications with this service are made via REST.

Since this service is the main responsible for orders in the company, it has a size considerably large, having many presentation controllers, services, domain objects, and repositories.

Nowadays, the Occurrences software is located inside the ODS service and is responsible for:

- Configure each Occurrence reason and its form fields;
- Save Order and Return Occurrences;
- Apply related business rules and validations;
- Publish those constraints, by creating tickets into a support platform, so the responsible support team related to partner service can solve the cases.

There are many reasons to create an occurrence, and each one has a different form that must be filled by merchants.

1.2 Problem

With more customers and sales, the company platform has been growing and the software has been continually evolving with new developments and applications, and for Occurrences, there was no exception. In fact, the company has been implementing a microservices oriented architecture (Section 2.2.3.2) in the overall of its applications in order to evolve its platform. In the context of Occurrences, its software will be migrated from ODS service into a microservices architecture.

Having new business requirements over time, developing them on ODS, created difficulties for developers in the evolution and understanding of its domain model.

Business concepts and relationships were the same for Order and Return Occurrences, and with that, a shared domain model was implemented with the purpose of support both businesses in the same way, without having the need to replicate concepts or relationships. This domain model is constituted by multiple domain objects related to Occurrences, but that is shared simultaneously for different business contexts (for orders and for returns). Besides that, having little business rules, domain model objects were treated as data objects that carried information along with the system.

Furthermore, with software evolution, specific requirements for orders and returns emerged. Thus, application services are responsible to apply business logic over the domain instead of domain objects. With this strategy, domain logics were spread over the application services inside the monolithic system. Besides that, the model is being shared by two businesses (orders and returns) that are taking different paths related to business.

Having this, for developers it is difficult to understand the business concepts and rules in the code, requiring extra efforts in the moment of evolving the domain model for new business logic or changes. Furthermore, it is required to take a double attention when applying new business requirements to Order and Return Occurrences, since one change in Order

Occurrences can affect directly Return Occurrences, and vice-versa, creating the need to test and do all the regression, carefully, to both businesses to make sure the system requirements are being met.

More details about the problem related to Domain Model will be presented in Section 2.3.

1.3 Objectives

The objective of this project is to create a solution that eliminates the constraints of the domain for the Occurrences software, during its migration for microservices, in a way that developers can have fewer difficulties during the implementation of new business requirements.

Therefore, the following steps must be accomplished to reach the project's objective:

- Design a solution to migrate Occurrences software for a microservices oriented architecture, using Domain Driven Design, and applying design patterns whenever they are adequate for the problem;
- Implement technical aspects of Domain Driven Design related to the provided solution, that must be comprehended by the remaining elements of the team, and which helps the construction of a domain with quality, in the various microservices.

1.4 Structure

In the first chapter, the detailed context of the project is presented, as well as the existent project problem, which must be solved, and the objective with its steps to reach one solution to the corresponding problem.

In the second chapter, it is detailed the Background of this project, with the State of the Art, which presents the fundamental concepts and a technical foundation. This chapter ends with a detailed explanation of the project problem.

In the third chapter, the Value Analysis of this project is presented, with details of its concepts and the approached methodology to evaluate the opportunities, with the presentation of the ideas that were thought to define a solution strategy to eliminate the problem, and the selection of the best idea. It ends with a value proposition, and with the business representation provided by Business Model Canvas.

In the fourth chapter, it is presented the analysis and design of the solution to be implemented, designing and explaining the approaches to building a solution. This chapter formulates a guide that must be followed during the next chapter, related to Implementation.

In the fifth, the technical implementation of the designed solution is detailed. The technical aspects are detailed and explained, following the design that was presented in the previous chapter. In this chapter, the final solution is built up, describing the implemented technical steps necessary to achieve it.

In the sixth chapter, the evaluation process for the solution to be implemented is defined, presenting the measurements to use, the hypotheses intended to be tested in order to support the work results, detailing the evaluation methodology and hypotheses testing. It ends with the solution evaluation, announcing the results that were obtained with the new solution, and concluding if it was a success.

Finally, in the seventh chapter, achieved objectives are presented, and also limitations and necessary future work for this project, ending with a personal appraisal.

2 Background

This chapter presents fundamental and technical concepts related to the State of the Art of the project. Furthermore, this chapter ends with a detailed explanation about the project problem, approaching the concepts that were previously introduced.

2.1 Fundamental Concepts

This section explores the E-commerce, the concepts about an Order and a Return, and also about Occurrences.

2.1.1 E-commerce

E-commerce, which stands for electronic commerce, is the definition of buying and selling of goods and services over an electronic network. Many types of transactions can occur such as: business-to-business (B2B), business-to-consumer (B2C), consumer-to-consumer (C2C) or consumer-to-business (C2B) (Rouse, 2018).

Nowadays, monolithic architecture is the base B2C and B2B e-commerce platforms, which had grown continuously over time until having several difficulties to update their systems. For many organizations, the solution to overcome those difficulties is the implementation of microservices oriented architecture, with the break of large applications into smaller services with defined responsibilities (Kwiecien, 2018).

"Almost all the successful microservice stories have started with a monolith that got too big and was broken up." – Martin Fowler (Fowler, 2015)

Implementing microservices step by step might require a long-term process to analyze and design property the complexity of the ecosystem. It must be implemented carefully in order to synchronize all the inner processes and to align the communications between the services. Companies such as Amazon and Netflix have already structured their systems using microservices oriented architecture (Kwiecien, 2018):

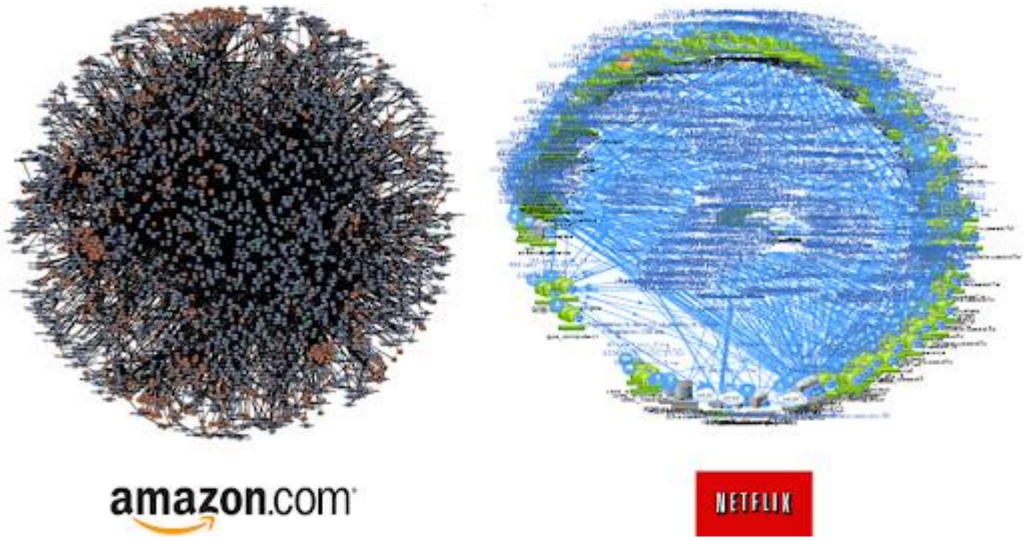


Figure 1 - Microservices ecosystem in Amazon and NetFlix
(Kwiecien, 2018)

2.1.2 Order

Order is a concept that has different definitions, depending on its context. One Order can be classified as Sales Order or as Merchant Order, which is explained next.

The Sales order is a prepaid request, triggered by a customer, for a product or service (or multiple ones), written in a form that, ultimately, will become a receipt (Lazzari, 2018). The customer is responsible to select the items intended to a cart, selecting sizes and quantities wished, while it is shopping. After that, he starts the checkout process.

Checkout is the process where the customer fills the necessary fields about billing address, shipping address and method, payment method and the payment. After the payment is released, it is created a Sales Order that will be processed by the platform.

A document is generated from the company to its customers, containing all the details about the Sales Order, such as information about products, sizes, prices, and quantities. Also, the customer has access to a document already filled by the company, so he can return the items using the provided shipping address.

Inside the Sales Order, are located the items that the customer selected in the checkout process, and which he has visibility about details of order processing, as for example, if some items were already shipped to his shipping address location.

The items inside a Sales Order might be provided by different merchants. This means that in the platform, it is necessary to split this Sales Order into multiple Merchant Orders since merchants can be different and those orders must be processed separately by the system.

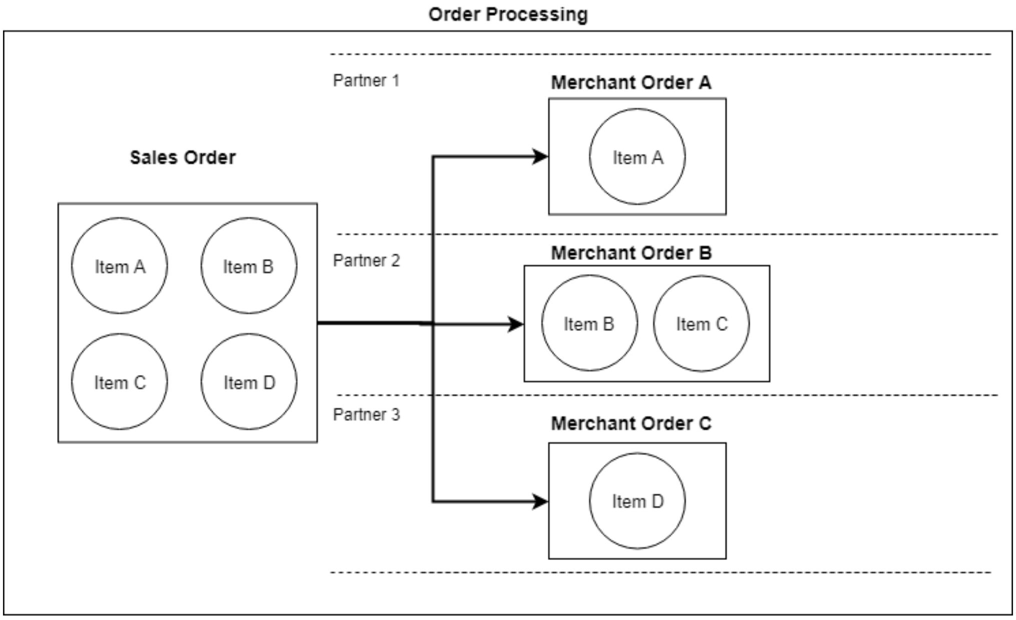


Figure 2 - Sales Order split into Merchant Orders

In the previous figure, it is possible to verify that during the order processing the Sales Order is split into many Merchant Orders as necessary, where each one contains the items corresponding to its merchant.

During this document, the concept Order has always the meaning of Merchant Order, since the Occurrences are only related to Merchant Orders.

Order Processing

Order processing is the process of order fulfillment, where its operations or facilities are normally known as distribution centers. Order processing is generally the term used to describe the workflow process that is associated with the main steps as picking, packaging, and delivery of the customer wished items to a shipping carrier that is responsible to deliver them to the intended location (Reviews and Kotler, 2017).

Figure 3 - Order Processing
(AllJewelrybrands, 2018)

In the company, order processing starts when a customer finishes its checkout process and the Sales Order is available for the system and split into Merchant Orders.

Merchants will be the main responsible for order processing associated with its stock products. They will pick each article and verify if it is available to send, preparing it for the packaging. After that, the recommended box will be selected, so it can have the best fit for the chosen items and the packaging is prepared with all the documentation necessary for the delivery.

Finally, when the packaging is ready, the order is sent to the corresponding carrier, which will be responsible to deliver the package to the shipping address defined by the customer.

2.1.3 Return

Return is when a customer sends a product back to the seller, having some reasons such as the product does not fit the intended size or because a wrong item was sent (AccountingTools, 2017).

In the platform, the Returns are always about Merchant Orders. The customer has a limited time to do a Return about the items after the order is shipped to the destination, not being authorized to do it after the limit is exceeded.

After sending the merchandise back to its seller, the items are evaluated by the merchants in order to verify if any defect exists, which disables the article to be sold again. If an issue is found, the customer will not be refunded, otherwise, it will receive back the corresponding value that he paid initially in the sales order.

Return Processing

In the company, Return Processing is the act of process one return about one or multiple items that customer sent back to merchants. Carriers are responsible to deliver the products to the stores again, and those are responsible to verify if the sent articles are in good conditions to be resold.

Finally, merchants confirm that the items were delivery with success and if its state is ok to sell again to customers.

2.1.4 Occurrences

In the company, are called Occurrences to anomalies that can happen associated with orders or returns, which merchants have the freedom to report to the responsible supporting teams to help them to solve the reported situations. Next will be provided the context about occurrences related to Orders and to Returns.

Order Occurrences

Occurrences can exist in any activity of order processing, such as picking, packaging or delivery. As it was referred before, during the document the concept Order has the meaning of Merchant Order. With that, Order Occurrences are always about Merchant Orders and its processing.

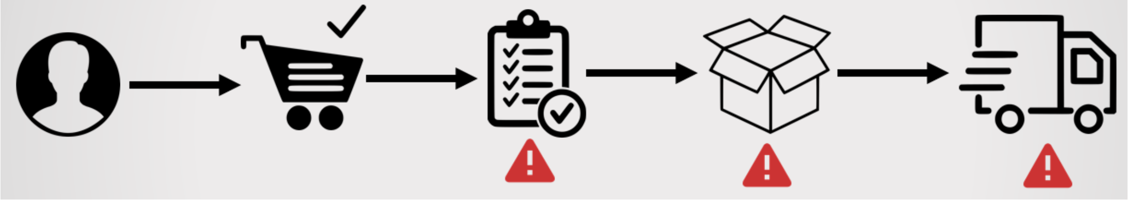


Figure 4 - Occurrences during Order Processing

Each activity has a set of causes that can be chosen by the merchant, so he can explain why he is reporting the situation, and what is the main cause to do it. Each cause might have specific forms to be filled, and each field might be or not be required to have a value. Besides that, the merchant can attach any additional documents that are required, or that he wants to send in a way to help to solve the constraint.

When an occurrence is created, the order pauses its processing in a way to solve any associated problem. With that, time delivering can be affected and the customer may have to wait for its products more than expected. While that is happening, support teams and merchants will be working together to solve the occurrence as much fast as it can, so the order can be unlocked back to its processing and delivered for the customer.

During the processing, one order can have multiples occurrences created by the merchant at any moment.

Return Occurrences

These occurrences can exist during return processing, in any activity related to a merchant, such as delivery or item verification.

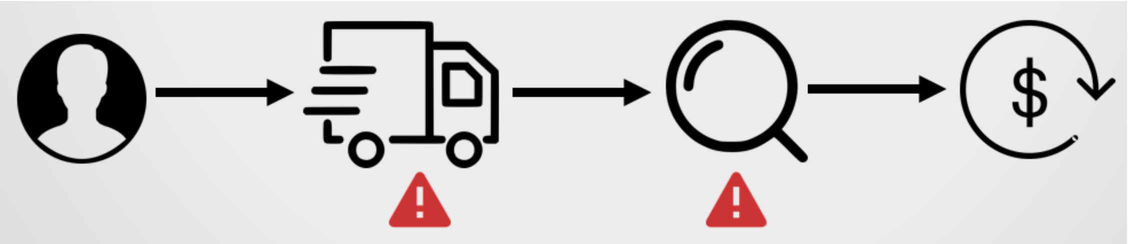


Figure 5 - Occurrences during Return Processing

Identically to order processing, each activity has a set of causes that can be chosen by a merchant, so he can explain why he is reporting the situation and what is the main cause to do it.

Similar to Orders, when an occurrence is created, the return will pause its processing in a way to have solved any associated problem. With that, support teams and merchants will work together in order to solve the occurrence, so the items associated can be available again to sales.

2.2 Technical Foundation

In this section, technical concepts and approaches are explained, and compared if necessary, in order to learn and understand their objects and usage, creating a technical comprehension. With these concepts in mind, it is possible to design and build a solution that solves the constraints of the project problem.

2.2.1 Software Evolution with Microservices

Microservices can be defined as small, independently deployable modules that interact with one another, using loosely coupled communication pathways. Being smaller then monolithic systems, and having many advantages (Section 2.2.3.2), microservices are a new way to implement and deliver resilient software systems that embrace change and evolution (Oliphant, 2018).

Microservices are considered as a logical extension of long-term trends related to the evolution of software development, having the expectation to have more organizations implementing microservices as an architecture solution to update its monolithic systems, satisfying their needs (Oliphant, 2018).

2.2.2 Technical Debt

Technical debt in software development was introduced by Ward Cunningham (Cunningham, 1992) as a metaphor to explain to nontechnical product stakeholders the need for what it is called as "refactoring". This metaphor is used to describe many kinds of debts related with software development, encompassing broadly anything that stands in the way of deploying, selling, or evolving a software system, or anything that adds to the friction from which development endeavors suffer: test debt, people debt, architectural debt, requirement debt, documentation debt, or just an amorphous, all-encompassing software debt (Sterling, 2010).

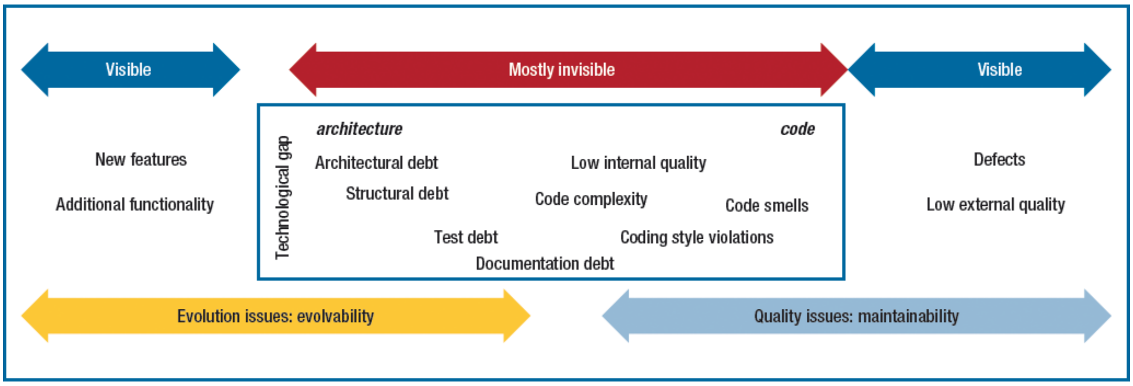
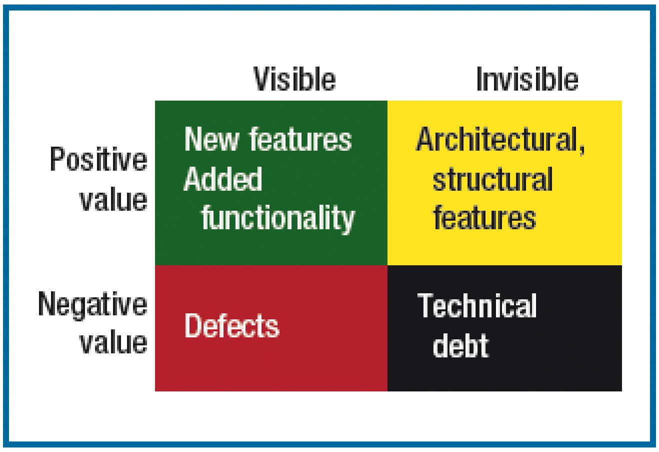


Figure 6 - Technical Debt Landscape
(Ozkaya, Nord and Kruchten, 2012)

The previous figure represents a possible organization for the technical debt landscape. It is possible to distinguish visible elements such as new functionality to add and defects to fix, and the invisible elements, which are commonly visible to software developers. The left side is dealing with software evolution or its challenges, while the right side is dealing with quality issues, both internal and external. It is proposed that technical debt is limited to invisible elements, that is, to the elements in the rectangular box, including invisible aspects of evolution and quality (Ozkaya, Nord and Kruchten, 2012).

Addressing Technical Debt

When the time to market is essential, the debt might be a good solution to improve the software. To pay this debt it is necessary to identify it and its causes. After that, it is necessary to manage the debt explicitly, involving listing debt-related tasks in a common backlog during release and iteration planning (Ozkaya, Nord and Kruchten, 2012).



	Visible	Invisible
Positive value	New features Added functionality	Architectural, structural features
Negative value	Defects	Technical debt

Figure 7 - Backlog Elements
(Ozkaya, Nord and Kruchten, 2012)

The previous figure illustrates how the elements might be organized in a backlog, with four different areas for improvements, with tasks to attend to increase value for the future: adding new features (green), investing in architecture (yellow), reduce negative effects on value of defects (red) or technical debt (black). Currently, backlogs are filled with features to be implemented, a few architectural tasks, some defects when problems happen and no elements for technical debt (Ozkaya, Nord and Kruchten, 2012).

Technical debt is not only about code quality, but also about structural or architectural choices or to technological gaps. With this, the main issue to face is how the organization can decide future changes. The decision-making process about which sequence of changes to apply could be the main reconciling point across the whole landscape shown in figure 6, from adding new features, adapting to new technologies, fixing defects and improving quality.

Technical debt should not be treated in isolation from adding new functionality or fixing defects, and at a given point of time, debt could be defined as deferred investment opportunities or poorly managed risks (Ozkaya, Nord and Kruchten, 2012).

2.2.3 Software Principles and Patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander

Alexander is an architect about buildings, but this definition works well for software. The focus of the pattern is a particular solution, one that's both common and effective in dealing with recurring problems (Fowler *et al.*, 2002).

Software Patterns offer well-established solutions to architectural and architecture problems, help to document design decisions, facilitate communication between stakeholders through a common vocabulary, and describe the quality attributes of a software system as forces (Avgeriou, 2005).

In this section, software patterns related to this project are presented, in order to understand how structured are the patterns that are referred several times on this document, and what benefits they provide.

2.2.3.1 Monolithic Architecture

Monolithic architecture is an architectural style that develops and deploys an application as a single unit containing all the necessary components. Normally, one monolithic application consists of a three-tier layer: UI, business logic and data access as it is possible to see in the next figure (Mikkonen, 2012):

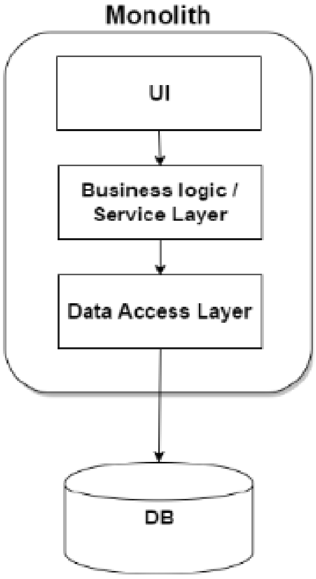


Figure 8 - Monolithic three-tier layer
(Mikkonen, 2012)

The referred layers have the following responsibilities (Microsoft, 2018c):

1. **UI**: is responsible to present the data information for the user, and might permit data manipulation and data entry;
2. **Business Layer**: is responsible to enforce business rules of the system;
3. **Data Access Layer**: is responsible to manage the access to database data and to abstract its technology communications for superior layers.

When this architecture is implemented in applications, which size is relatively small, it can result in several benefits (Richardson, 2018b):

1. **Simple to develop**: the objective of development tools and IDEs nowadays is to support the development of monolithic applications;
2. **Simple to deploy**: it is only necessary to deploy the application on the appropriate runtime;
3. **Simple to scale**: an application can be scaled by running multiple instances of the service.

Besides having its advantages, also it has characteristics that can be classified as disadvantages, mainly when the application becomes large (Richardson, 2018b):

1. **Continuous deployment is difficult**:
    a. It is an obstacle for frequent deployments since to update one component is necessary to redeploy the entire application;

    b. When one component is deployed, the others might fail its starting. This may produce a problem spread over the application, even if the components were untouched.

2. **Scaling can be difficult**:
    a. The application can scale by running multiple instances, but it can't scale with an increasing data volume;
    b. Each copy of the application instance will access to the data, which makes increases memory consumption;
    c. Different components can have different resource requirements. This means that one task must have an intensive CPU while another might need an intensive memory. With monolithic architecture, it is not possible to scale each component independently.
3. **Difficulties in scaling development**:
    a. Once it gets a certain size it might be useful to have different teams acting and being responsible in different components. However, a monolithic application prevents the teams from working independently which requires coordination between team developments.
    b. It creates an obstacle for the teams due to deployments. When having multiple teams working on the same application, it is much more difficult to create or update a change in the production environment.
4. **Requires a long-term commitment to a technology stack**:
    a. This architecture forces teams to use the same technology stack since the start of the development, which prevents to implement some components of the application in different languages or frameworks;
    b. If the application uses a framework that becomes obsolete, then it can be a challenge to migrate the whole application to a newer and updated framework.

Most of the applications have monolithic architecture because of these reasons. This approach is enough to develop features in the beginning and having a possibility that the size of the codebase will never grow that much and the need for fine-grained scaling will be never necessary (Mikkonen, 2012).

2.2.3.2 Microservices Oriented Architecture

Microservices Oriented Architecture is a distributed application which modules are microservices, where each one, is an independent and cohesive process that can be interacted via messages (Dragoni *et al.*, 2017).

These services have recently emerged with a promising architectural style to give solutions for enterprises with monolithic applications (Knoche, 2016). Microservices are inspired by Service Oriented Architecture (SOA) and focus on aspects, such as componentization of small services, application of agile practices for development, deployment, and testing, usage of automated infrastructure with continuous delivery and decentralized data management (Di Francesco, 2017).

In the next figure, it is possible to verify how an example of how microservices architecture is structured:

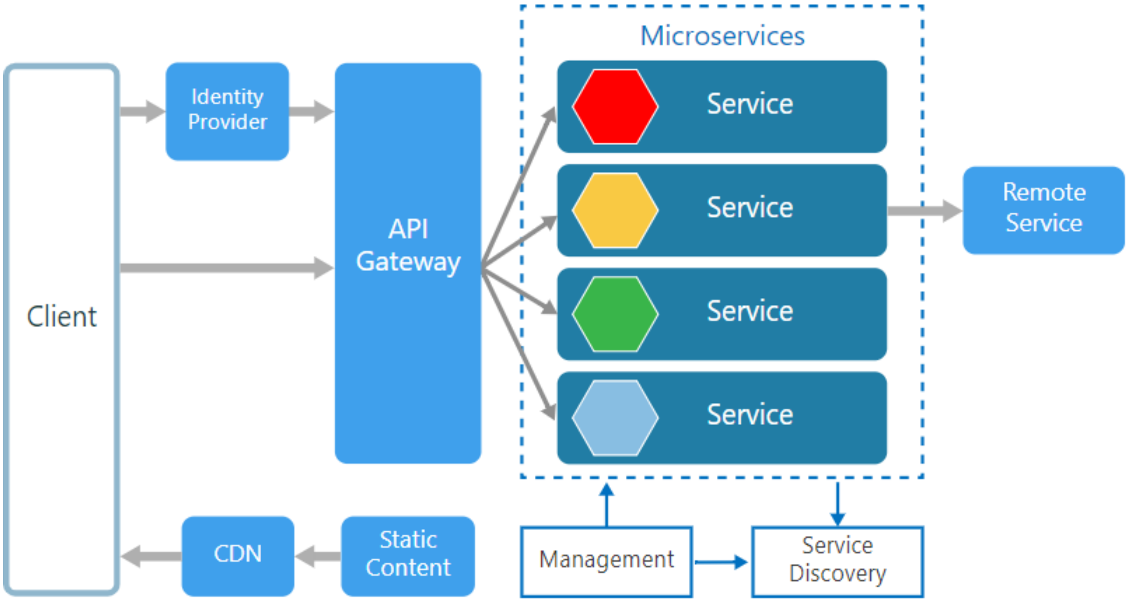


Figure 9 - Microservices architecture
(Wasson, Buck and Celarier, 2018)

Besides the services themselves, some other components might appear in a typical architecture (Wasson, Buck and Celarier, 2018):

1. **Management**: Component responsible for placing services on nodes, identify failures and rebalancing services across the defined nodes;
2. **Services Discovery**: Component responsible for maintaining a list of services where specified nodes are located on. Also enables a service lookup to find an endpoint for a service;
3. **API Gateway**: Component responsible for being an entry point for clients. Services are not called directly by those, so they must call the API Gateway, which redirects the call to the responsible services. It might aggregate responses from multiple services in order to return a single response with the necessary information.

This architecture is considered as an alternative to monolithic architecture, having its characteristics, the next ones are classified as advantages (Richardson, 2018a):

1. **Enables continuous deployment**:
    a. Improves testability since services are smaller and faster to test;
    b. Improves the ability to deploy since services can be deployed in an independent way;
    c. It helps to structure teams, so each one can be responsible for one or multiple services. Each team can develop, deploy and scale their services independently from the other teams.

2. **Each service is small**:
    a. It improves the understanding of a developer;
    b. It doesn't overload IDE's, so it is more productive;
    c. The application starts faster, which accelerates developer's productivity and deployments;
    d. Each service has its own database. This means databases schemas will be significantly smaller than in monolithic.
3. **Improves fault isolation**: When a service is down, having errors or performance issues, only it is affected. This means the other services will still be available to handle requests and will not be directly affected by that service;
4. **Eliminates long-term commitments to a technology stack**: Microservices can be implemented in different technologies. This makes possible to choose proper technologies for the context of the service and provides knowledge to the teams that want to learn more about software.

Besides having several advantages, also it has characteristics that can be classified as disadvantages (Wasson, Buck and Celarier, 2018):

1. **Systems are more complex**:
    a. Microservices application has more moving parts than the same application as monolithic;
    b. Each service is simple, but the entire system is more complex.
2. **Lack of governance**: Implementing multiple different languages or frameworks can difficult the maintenance of the application.
3. **Network latency**:
    a. Using several small services can imply more communications that can result in more latency over the network, slowing down the requests;
    b. If communications are not well designed, the application will slow down.
4. **Data integrity**: Having each service responsible for its data creates difficulties to maintain data consistency in the system.

2.2.3.3 SOLID

SOLID is an acronym used to represent five design principles when doing Object Oriented Programming (OOP). These principles were introduced by Robert C. Martin, but the SOLID acronym was, however, identified later by Michael Feathers (LH, 2019). In this section, SOLID principles are addressed at a high level so their values can be understood, and its benefits for microservices implementation can be detailed. During this section, SOLID principles (Martin, 2000) are briefly explained, as also the SOLID applied to microservices.

1. Single Responsibility Principle
Single Responsibility Principle says: "*Find one reason to change and take everything else out of the class*". It claims to separate the things that change for different reasons, grouping together the things to change for the same reasons. Early, Object Oriented Design principles had as

grouping together functions that operated on the same data structures, so that the methods of a class would all manipulate the same variables of that class, but if those methods change for different reasons then they really belong in separate classes.

2. Open Closed Principle

This is one of the most important principles for object-oriented design, having the meaning that modules should be written with availability to be extended, without requiring them to be modified, being open for extension but closed for modification. With that, it is intended to have modules with the capability to be modified without changing the source code of the corresponding ones.

3. Liskov Substitution Principle

This principle was approached by Barbar Liskov during her work, regarding data abstraction and type theory. Derived classes should be substitutable for their base classes. That means that a user of a base class should continue to work properly if a derivate of that base class is passed to it.

In order to be substitutable, the contract of the base class must be honored by the derived class. To state the contract of a method, it is necessary to declare what must be true before the method is called. This is called the precondition. If precondition fails, then the results of the method are classified as undefined, and the method ought to be not called. It is possible to declare what the method guarantees that will be true once it has completed. This is called the postcondition. A method that fails its postcondition should not return. With that, a derived class is substitutable for its base class if (Martin, 2000):

- Its preconditions are no stronger than the base class method;
- Its postconditions are no weaker than the base class method.

With this, derived methods should expect "*no more and provide no less*".

4. Interface Segregation Principle.

The essence of this principle is simple. When having a class that has several clients, rather than loading the class with all the methods that the client needs, should be created specific interfaces for each client and multiply inherit them into the class. When software is maintained, interfaces are tended to change and there are times that these changes have a huge impact, forcing the recompilation and redeployment of a very large part of the design. This impact can be mitigated by adding new interfaces to existing objects, rather than changing the existing interface.

5. Dependency Inversion Principle

Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes. The implication of this principle is quite simple. Every dependency in the software design should target to an interface or an abstract class. No dependency should target a concrete class. The reason is that concrete things change currently, while abstract thins change much less frequently (Martin, 2000).

6. SOLID applied to Microservices

When developing microservices, it is possible to apply SOLID principles during its implementation. Table 1 represents how these principles can be applied to microservices, provided by each principle (Terreno, 2013).

Table 1 - SOLID in Microservices
(Terreno, 2013)

Principle	Definition
Single Responsibility Principle	Each responsibility should be a separate microservice because each responsibility is an axis of change.
	A microservice should have only one reason to change.
	If changes in business rules require changes to the microservice, then a change to the database schema, report format, or any other segment of the system should not for that microservice to change.
Open Closed Principle	The existing code of microservices should never have the need to change. This prevents new bugs in the existent code. If it never changes, then it cannot break.
Liskov Substitution Principle	If for each microservice instance m1 of type S there is a microservice instance m2 of type T such that for all other microservices P defined in terms of T, the behavior of P is unchanged when m1 is substituted for m2 then S is a rewrite of T.
Interface Segregation Principle	The dependency of one microservice to another one should depend on the smallest possible interface.
Dependency Inversion Principle	With this principle it is possible to avoid designs on microservices which are: • Rigid: hard to change due to dependencies; • Fragile: changes cause unexpected bugs; • Immobile: difficulties in the reuse due to implicit dependence on current application code.

2.2.3.4 CQRS

Command Query Responsibility Segregation (CQRS) is a pattern that allows using a different model to update information than the model used to read the information (Fowler, 2011).

The mainstream approach people use for interacting with an information system is to treat it as a CRUD data store. With this, exists a model of a record structure where it is possible to create, read, update and delete records. But when the needs become more sophisticated, it may be necessary to gather information in a different way to the record store, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places (Fowler, 2011).

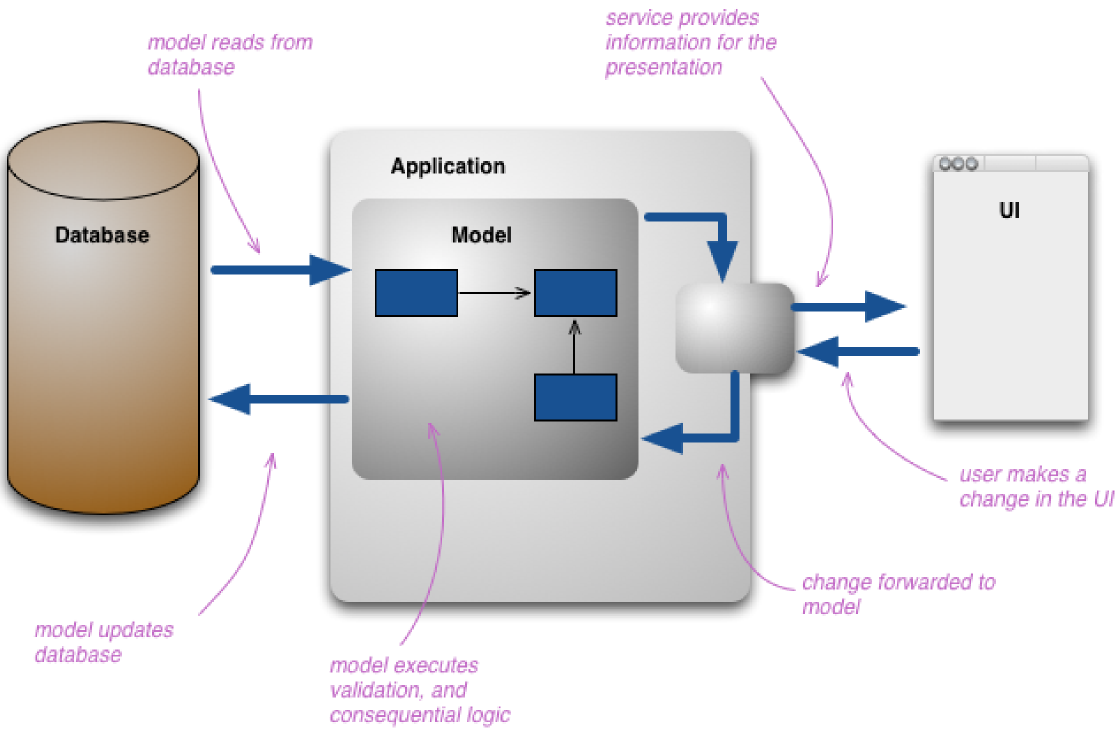

model reads from database
service provides information for the presentation
Application
Model
Database
UI
user makes a change in the UI
change forwarded to model
model updates database
model executes validation, and consequential logic

Figure 10 - Common Design Model
(Fowler, 2011)

With that, the software starts to have multiple representations of information. Developers typically build their own conceptual model which they use to manipulate the core elements of the model. Domain Model is usually the conceptual representation of the domain and typically is used to make persistent storage as close to it (Fowler, 2011).

The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively following the vocabulary of Command Query Separation. The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well (Fowler, 2011).

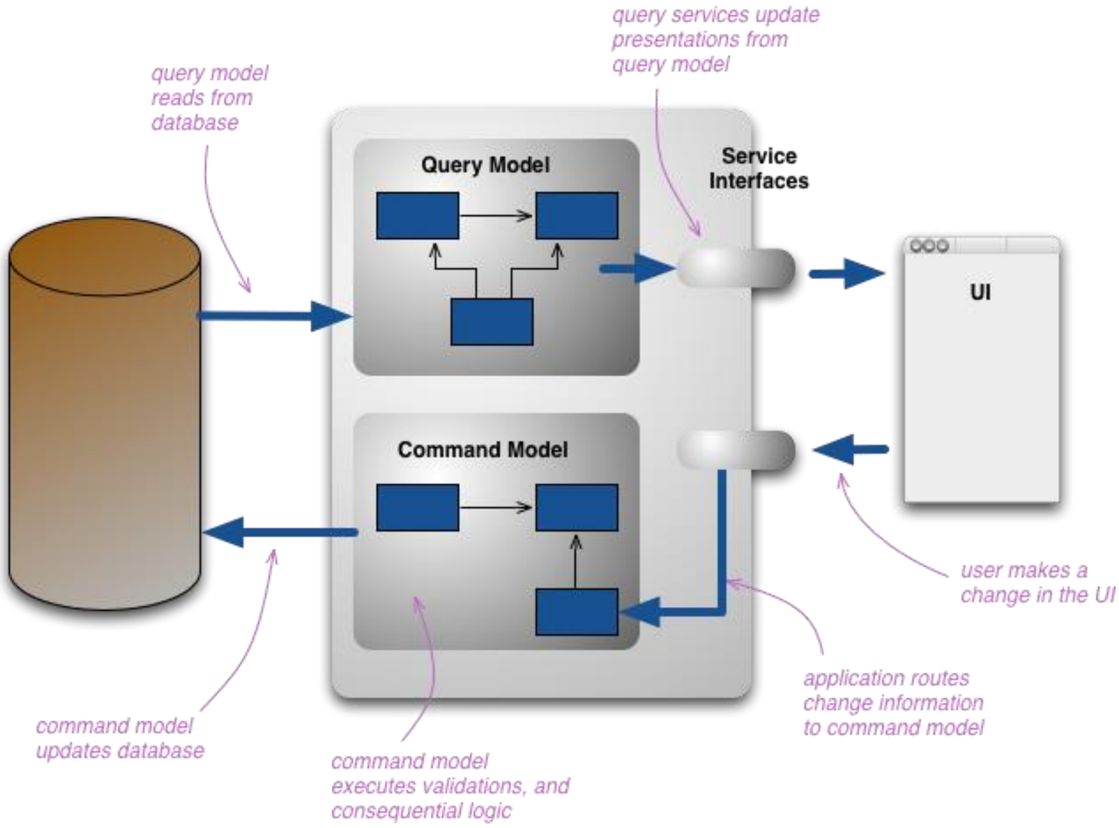

query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
command model updates database
application routes change information to command model
command model executes validations, and consequential logic

Figure 11 - CQRS Design Model
(Fowler, 2011)

The two models might not be separate object models, it could be that the same objects have different interfaces for their command side and their query side, rather like views in relational databases (Fowler, 2011).

When to use it

CQRS should only be used on specific portions of a system, such as a Bounded Context in Domain Driven Design, and not the system as a whole. In this way of thinking, each Bounded Context needs its own decisions on how it should be modeled (Fowler, 2011).

Few complex domains may be easier to tackle by using CQRS. Usually, there is enough overlap between the command and query sides that sharing a model is easier. Using CQRS on a domain that does not match it will add complexity, thus reducing productivity and increasing risk (Fowler, 2011).

The other main benefit is in handling high-performance applications. CQRS allows to separate the load from reads and writes allowing the application to scale independently. If an application has a significant disparity between reads and writes this is very handy. Even without that, it is possible to apply different optimization strategies to the two sides (Fowler, 2011).

Many information systems fit well with the notion of an information base that is updated in the same way that it is read, adding CQRS to such a system can add significant complexity. With that, this is a pattern that it is difficult to use well and that is easy to be mishandled (Fowler, 2011).

CQRS with Domain Driven Design

As previously referred, CQRS means having two objects for a read/write operation wherein other contexts there is one. With CQRS is possible to have more flexibility in the queries instead of limiting the queries with constraints from DDD patterns such as aggregates (Microsoft, 2019a).

Imagining a microservice-based on a simplified CQRS approach that uses a single database, but two logical models plus DDD patterns for the transactional domain, as shown in the next figure (Microsoft, 2019a):

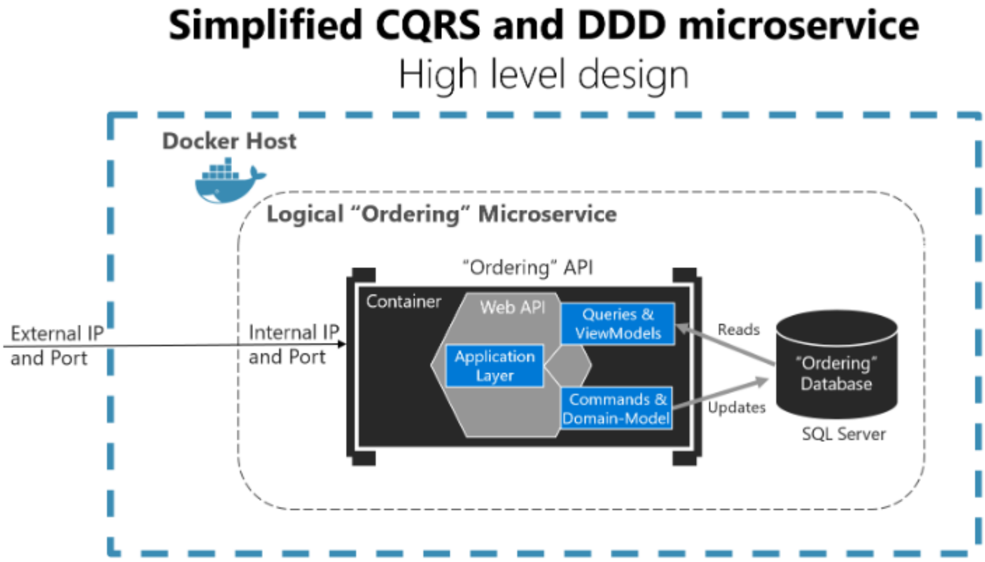


Figure 12 - CQRS and DDD Microservice
(Microsoft, 2019a)

Here, microservice has split the queries and ViewModels (data models created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates (Microsoft, 2019a).

Queries are idempotent, which means that every query that is executed in the system, the state of that system will not change. On the other hand, commands, which trigger transactions and data updates, change state in the system. With commands, it is necessary to be careful when dealing with complexity and ever-changing business rules. This is where DDD strategy is applied in order to have a better-modeled system (Microsoft, 2019a).

DDD patterns might introduce constraints in this design, since those add complexity with fewer benefits for reading and querying, although those provide benefits such as higher quality over time, especially in commands and other code that modifies system state (Microsoft, 2019a).

Because of that, is suggested to use DDD patterns only in transactional/updates area of microservice, which are triggered by commands. Queries can follow a simpler approach and should be separated from commands, following a CQRS approach (Microsoft, 2019a).

2.2.3.5 Clean Architecture

Over the last years, many ideas came up regarding the architecture of systems, such as Hexagonal Architecture, Onion Architecture, as many others. Though these architectures all vary somewhat in their details, they are very similar. These architectures have the same objective, which is the separation of concerns, all achieving this separating by splitting the software into layers, and each one has at least one layer for business rules, and another for interfaces (Martin, 2017).

The Clean Architecture (Martin, 2017) is an architecture that attempts to integrate all the previous architectures into a single actionable idea, as the next diagram represents.

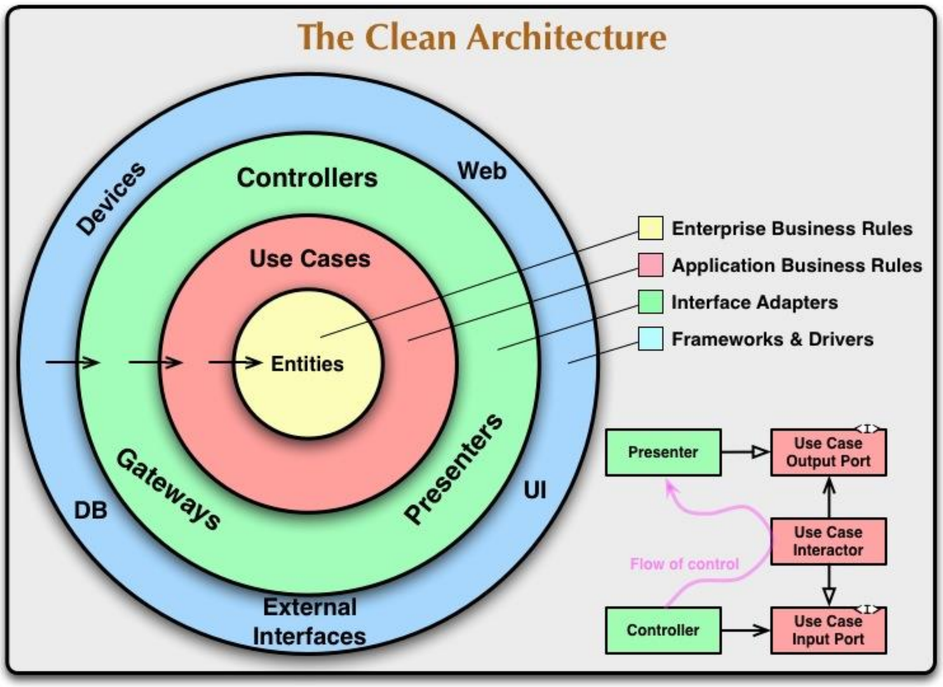


Figure 13 - Clean Architecture
(Martin, 2017)

1. The Dependency Rule

The circles of the previous diagram represent different layers of software. In general, the further it goes, the higher level the software becomes. The outer circles are mechanisms and inner circles are policies.

The overriding rule that makes this architecture work is the Dependency Rule. This rule says that source code dependencies can only point inwards and that nothing in an inner circle can know anything at all about something in an outer circle. In particular, the name of something declared in an outer circle should not be mentioned by the code in an inner circle. That includes functions, classes, variables, or any other named software entity.

2. Entities

Entities are responsible to encapsulate business rules. It can be an object with methods, or it can be a set of data structures and functions. It does not matter so long as the entities can be used by many different applications in the enterprise. Entities encapsulate the most general and high-level rules and they are the least likely to change when something external changes.

3. Use Cases

This layer contains the application-specific business rules. It encapsulates and implements all the use cases of the system. These use cases orchestrate the flow of data to and from the entities and direct those entities to use their business rules to achieve the goals of the use case.

It is not expected that changes in this layer affect the entities, neither it is expected that this layer can be affected by external changes such as database or user interface, being isolated from such concerns.

It is expected that changes to the operation of the application will affect the use-cases and therefore the corresponding software in this layer. Also, if details of a use-case change, then code in this layer will certainly be affected.

4. Interface Adapters

The software in this layer is a set of adapters that are responsible to convert data from the format most convenient for the use cases and entities, to the format most convenient for some external software such as database or Web. In this layer, if an MVC architecture is used, then its presenters, views, and controllers will belong here. These models are just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.

In this layer, data is converted from the most convenient for entities and use cases, into the form most convenient for the persistence framework that is being used. No code inward of this circle should know at all about the database. If the database is an SQL database, then all the SQL should be restricted to this layer, and in a particular to the parts of this layer that are responsible for the database.

Also, this layer is responsible to adapt data from external services to be used in the internal form by use cases and entities.

5. Crossing Boundaries

In the previous diagram, it is possible to verify how circle boundaries are crossed. It shows the controllers and presenters communicating with the use cases in the next layer, where the flow starts in the controller and moves through the use case, and then winds up executing in the presenter. Relatively to source code dependencies, each one of them points inwards towards the use cases.

This pattern uses the Dependency Inversion Principle, such that the source code dependencies oppose the flow of control at just the right points across the boundary. This technique is used to cross all the boundaries in the architectures. It is possible to take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so that it is possible to achieve the Dependency Rule, no matter what direction the flow of control is going in.

6. Conclusion

Using Clean Architecture (Martin, 2017) separates the software into layers, confirming to the Dependency Rule, creating a system that is intrinsically testable, with all the benefits that imply. When any of the external parts of the system become obsolete, like the database or web framework, it is possible to replace those elements with a minimum impact.

2.2.4 Domain Driven Design

Software development is mostly used to automate processes that exist in the real world or to provide solutions to real business problems. These automated processes or real-world problems are considered as the domain of the corresponding software. It is important to understand since the beginning that the software is originated from and deeply related to the domain (Evans, 2003).

At the beginning of a software project, it is important to focus on the domain, since the entire purpose of the software is to enhance a specific domain, otherwise, it will introduce strain into the domain, provoking malfunction, damage, or even wreak chaos (Evans, 2003).

1. The Ubiquitous Language

During software implementation, usually exists long discussions between software architects or developers and domain experts. Software specialists have the intention to extract knowledge from the domain experts, in order to transform it into a useful form. While doing that, they may find issues with the implemented model or approach, and with that, changes are necessary to apply in the model (Evans, 2003).

A core principle of Domain-Driven Design is to use a language based on the model since it is the common ground and the place where the software meets the domain. The Ubiquitous Language (Evans, 2003) connects all the parts of the design, creating the premise for the design team to work with success. It requires much time to shape the project's design since it is necessary to focus to make sure that the key elements of the language are brought to light. It is necessary to find those key concepts which define the domain and the design and find corresponding words for them, using these works during communication. It is also necessary to experiment with alternative expressions, which reflect alternative models, refactoring code, renaming classes, methods and modules to conform with the new model. Confusion over terms during conversation must be resolved, in a way to all members agree in the meaning of ordinary words (Evans, 2003).

When building a Ubiquitous Language that has a clear outcome: the model and the language are strongly interconnected with one another, and a change in the language should become a change to the model (Evans, 2003).

If domain experts cannot understand something in the model or the language, then it is most likely that there is something wrong with it. On the other hand, developers should watch for ambiguity or inconsistency that will tend to appear in design (Evans, 2003).

2. Model-Driven Design

Any domain can be expressed with many models, and any model can be expressed in various ways in code. For each specific problem, there is more than one solution. One of the recommended design techniques is the analysis model, which is seen as separate from code design and is usually created by different people. This model is the result of business domain analysis, resulting in a model that has no consideration for the software used for implementation. This model reaches the developers which are responsible to do the design, and during this process knowledge about the domain can be lost or bad interpreted (Evans, 2003).

During model-driven design, many important patterns can be used. The purpose of these patterns is to present some of the key elements of object modeling and software design from the viewpoint of domain-driven design. Figure 14 is a map of the patterns and their relationships.

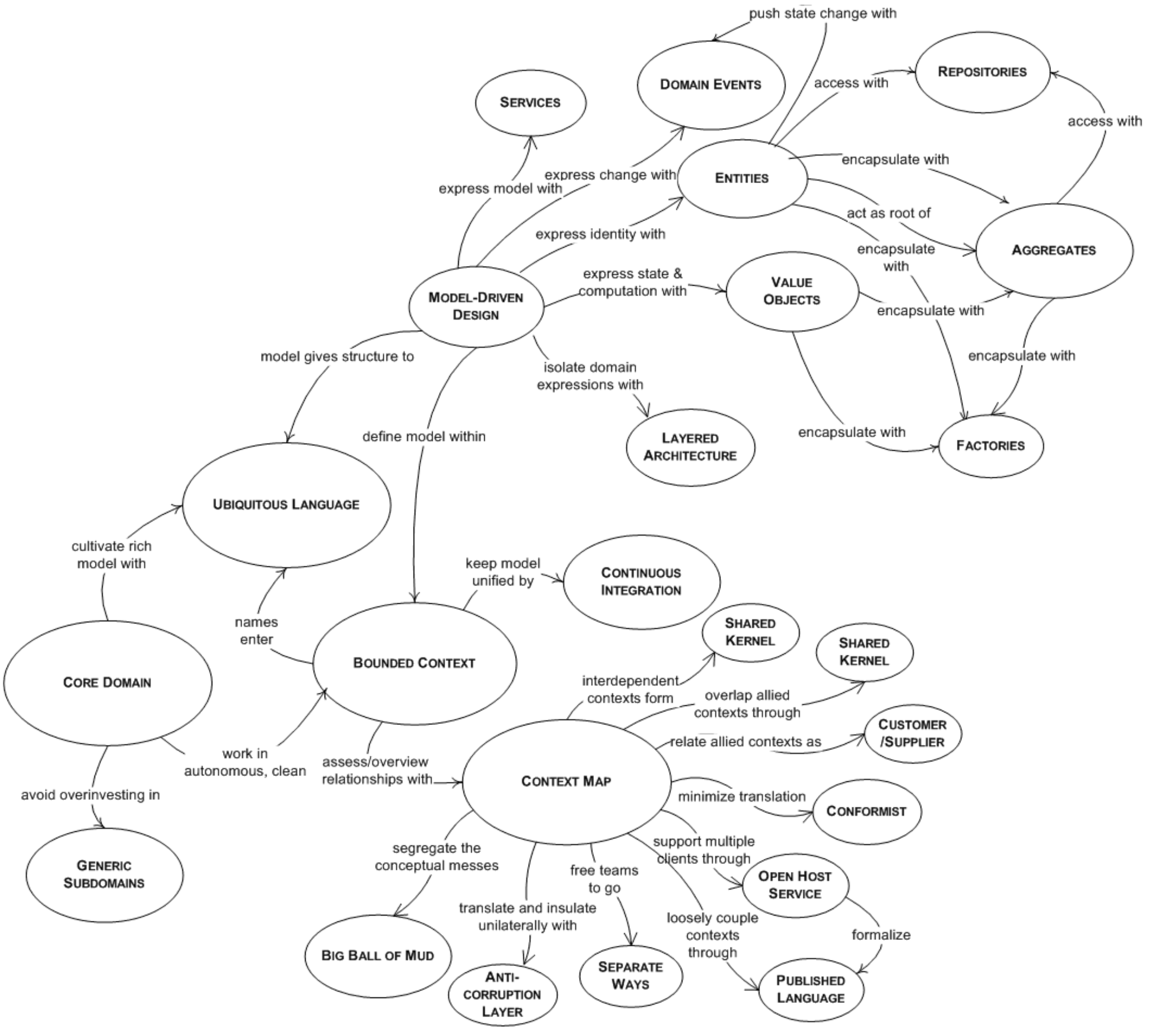


Figure 14 - DDD Patterns and its Relationships
(Evans, 2003)

Next, some patterns related to the model-driven design will be presented.

3. Layered Architecture

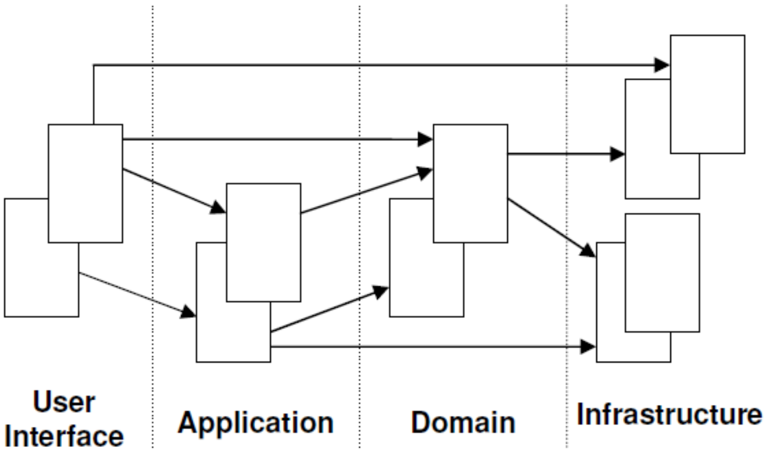


Figure 15 - Layered Architecture
(Evans, 2003)

When developing a software application, a large part of it is not directly related to the domain, but it is part of the infrastructure or serves the software itself. With Layered Architecture, it is possible to develop a design within each layer that is cohesive and that depends on the layers below. Architectural patterns should be followed to provide low coupling to the layers above. Also, all the code related to the domain model should be isolated in one layer from the user interface, application, and infrastructure code (Evans, 2003).

Domain objects must be free of the responsibility of displaying themselves, storing themselves, managing application tasks, and can be focused on expressing the domain model. With this, the domain model can be rich and clear enough to capture essential business knowledge and put it to work (Evans, 2003).

A common architectural solution for domain-driven designs contain four conceptual layers (Evans, 2003):

Table 2 - Layers Responsibilities on Layered Architecture
(Evans, 2003)

Layer	**Description**
User Interface (Presentation Layer)	Responsible for presenting information from the system to the user, interpreting user commands.
Application Layer	The thin layer which coordinates the application activity. It does not contain business logic, neither hold the state of business objects, but it can hold the state of an application task progress.
Domain Layer	This layer contains information about the domain and it is considered the heart of the business software. The state of business objects is held here, and the persistence of these objects and possibly their state is delegated to the infrastructure layer.
Infrastructure Layer	This layer acts as a supporting library for all the other layers. It provides communication between layers, implements persistence for business objects, contains supporting libraries for the user interface layer, etc.

It is important to split an application into separate layers, and establish rules of interactions between layers. If the code is not clearly separated into layers, it will soon become entangled that it becomes very difficult to manage changes (Evans, 2003).

4. Entities

The entity is considered as a category of objects that have an identity that remains the same throughout the states of the software. For these objects, it is not the attributes that matter, but a thread of continuity and identity, which spans the life of a system and can extend beyond it. Such objects are called Entities (Evans, 2003).

There are different ways to create a unique identity for each object. The ID could be automatically generated by a module, and used internally in the software without making it visible to the user, or generated by a database system (Evans, 2003).

When an object is distinguished by its identity, rather than its attributes, it is important to keep the class definition simple and focused on life cycle continuity and identity (Evans, 2003).

Entities are important objects of a domain model, and they should be considered from the beginning of the modeling process. It is also important to determine if an object needs to be an entity or not (Evans, 2003).

5. Value Objects

An object that is used to describe certain aspects of a domain, and which does not have an identity, is named Value Object. Selecting as entities the objects which conform to the entity definition, and making the rest of the objects Value Objects will simplify the design (Evans, 2003).

Having no identity, value objects can be easily created and discarded. It is highly recommended that value objects be immutable, being created by a constructor, and never modified during their lifetime. When it is needed a different value for the object, then another one is created. With this, value objects can be shared, improving software performances and manifesting software integrity (Evans, 2003).

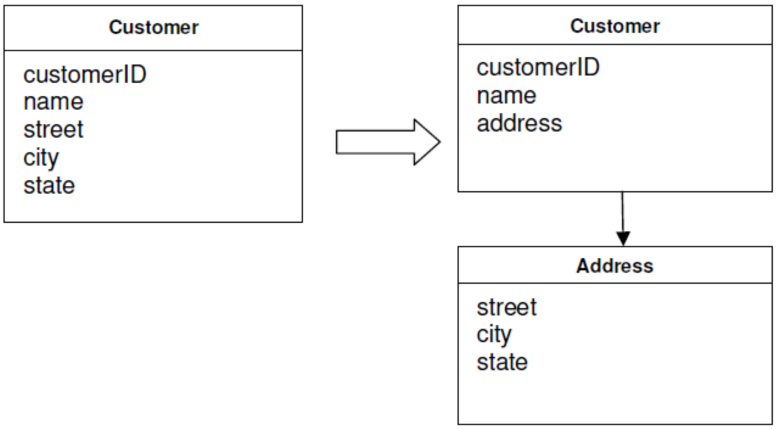


Figure 16 - Value Objects
(Evans, 2003)

Value objects can contain other value objects, and they can even contain references to Entities. Although value objects are used to simply contain attributes of a domain object, that does not mean that it should contain a long list with all the attributes. These attributes can be grouped into different objects (Evans, 2003).

6. Services

Some actions in a domain do not seem to belong to any object. They represent an important behavior of the domain, so they cannot be incorporated into some of the existent Entities or

Value Objects. Adding such behavior to an object would spoil the object, making it stand for functionality which does not belong to it. When such behavior is recognized in the domain, the best practice is to declare it as a Service (Evans, 2003).

A Service object does not contain an internal state, and its purpose is to provide functionality for the domain. The assistance provided by this kind of object can be significant, and a Service can group related functionality which serves the Entities and the Value Objects (Evans, 2003).

Services act as interfaces that provide operations, and it becomes a point of connection for many objects. That is one of the reasons why behavior, which naturally belongs to a Service, should not be included in domain objects (Evans, 2003).

A Service should not replace the operation which normally belongs to domain objects. This type of objects should not be created every time an operation is needed. But when such an operation stands out as an important concept in the domain, a Service object should be created for it. There are three characteristics of a Service (Evans, 2003):

1. A service performs an operation referred to a concept which does not naturally belong to an Entity or Value Object;
2. The performed operation refers to other objects in the domain;
3. The operation is stateless.

There are also services related to the application layer, which adds an additional level of complexity. These services are even more difficult to separate from their counterparts residing in the domain layer (Evans, 2003).

Both application and domain services are usually built on top of domain objects, providing required functionality related to those objects. Projecting the responsible layer for a Service might be difficult. If the performed operation belongs to the application layer, then the Service should be placed there. Otherwise, if an operation is about domain objects, and is strictly related to the domain, serving it, then it should be placed into the domain layer (Evans, 2003).

In conclusion, the application layer is a thin layer with service objects responsible to interact with database infrastructure operations and to interact with the domain layer when it needs to. The domain layer will contain the core of the domain, with objects related to concepts of domain and with services responsible for operations between domain objects directly (Evans, 2003).

7. Aggregates

Managing the life cycle of a domain object is a challenge, and if it is not correctly done, it may have a negative impact on the domain model. Aggregate is a domain pattern used to define object ownership and boundaries (Evans, 2003).

An aggregate is a group of associated objects which are considered as one unit about data changes. The Aggregate is demarcated by a boundary that separates the objects inside from those outside. Each Aggregate has one root, which is an Entity, and it is the only object accessible from outside. The root can hold references to any of the aggregate objects, and the other objects can hold references to each other, but an outside object can hold references only to the root object (Evans, 2003).

Aggregate ensures data integrity since other objects can hold references only to the root, which means that they cannot directly change the objects inside the aggregate. They only can request actions to the root, or even change it. With that, the root element will be able to change the other objects, but that is an operation contained inside the aggregate, and it is controllable. If the root is deleted and removed from memory, all the other objects from the aggregate will be deleted too, because there is no other object holding reference to any of them (Evans, 2003).

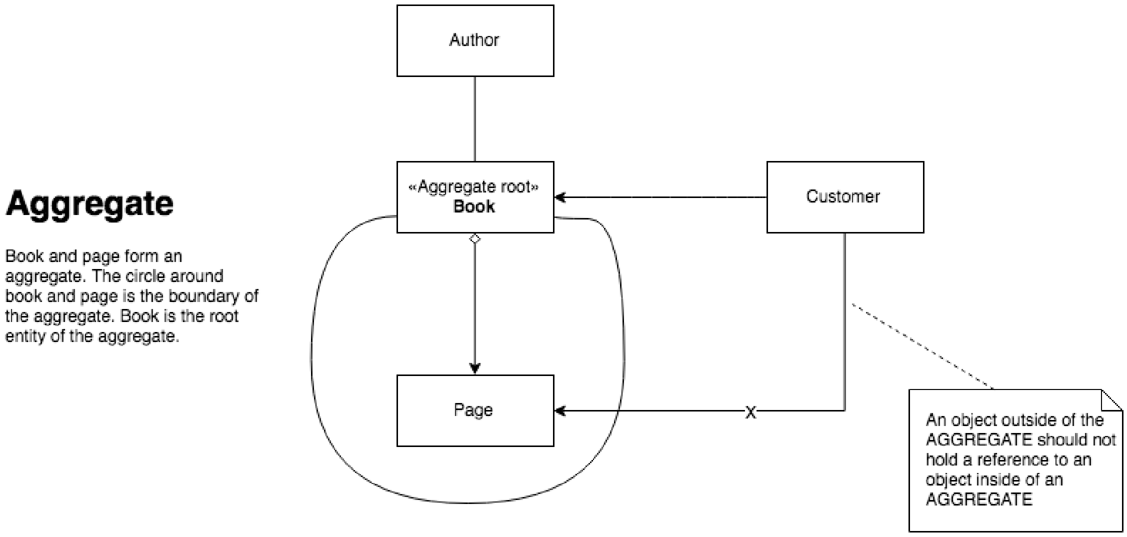

Aggregate
Book and page form an aggregate. The circle around book and page is the boundary of the aggregate. Book is the root entity of the aggregate.
Author
«Aggregate root» Book
Customer
Page
X
An object outside of the AGGREGATE should not hold a reference to an object inside of an AGGREGATE

Figure 17 - Aggregate Example
(Ruiz, 2018)

It is possible for the root to pass transient references of internal objects to external ones, but only with the condition that those external objects do not hold the reference after the operation is finished. One way to do that, is to pass copies of Value Objects to external objects since it does not matter what happens to those objects, because it will not affect the integrity of the aggregate in any way (Evans, 2003).

Furthermore, inner objects of an Aggregate should be allowed to hold references to the roots of other Aggregates (Evans, 2003).

8. Repositories

The Repository has the purpose of encapsulating all the necessary logic to obtain object references. Domain objects do not have to deal with the infrastructure to get the needed references to the other objects of the domain. These will be obtained from the Repository and the model is regaining its clarity and focus (Evans, 2003).

The Repository might store references to some of the objects, and when an object is created, it may be saved in the Repository, being retrieved from there to be used later. If a client requests an object from it, and the Repository does not have it, it may get it from the storage. Either way, the Repository acts as a storage place for globally accessible objects (Evans, 2003).

Its overall effect is that the domain model is decoupled from the need of storing objects or their references and accessing the underlying persistence infrastructure (Evans, 2003).

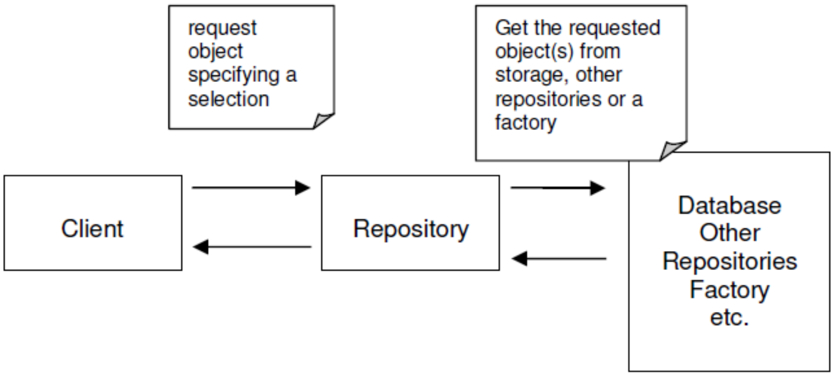

request object specifying a selection
Get the requested object(s) from storage, other repositories or a factory
Client
Repository
Database
Other
Repositories
Factory
etc.

Figure 18 - Repository Pattern
(Evans, 2003)

9. Bounded Context

Each model has a context, and when dealing with a single model, the context is implicit. When creating a new application that is supposed to interact with other software, for example, a legacy application, the new application has its own model and context, that is separated from the legacy model and its context, and they must not be combined (Evans, 2003).

Team cooperation and communication are more fluid and complete, which improves the developers' work for the same model. The context of a model is a set of conditions that need to be applied to make sure that the terms used in the model have a specific meaning (Evans, 2003).

The main idea is to define the scope of a model, drawing up the boundaries of its context, then do the most possible to keep the model unified (Evans, 2003).

A Bounded Context provides the logical frame inside of which the model evolves. When having multiple models, it is necessary to define the borders and the relationships between them. Each one having its own Bounded Context (Evans, 2003).

Related with Bounded Contexts, there exists a Context Map. It is a document that outlines the different Bounded Contexts and the relationships between them (Evans, 2003).

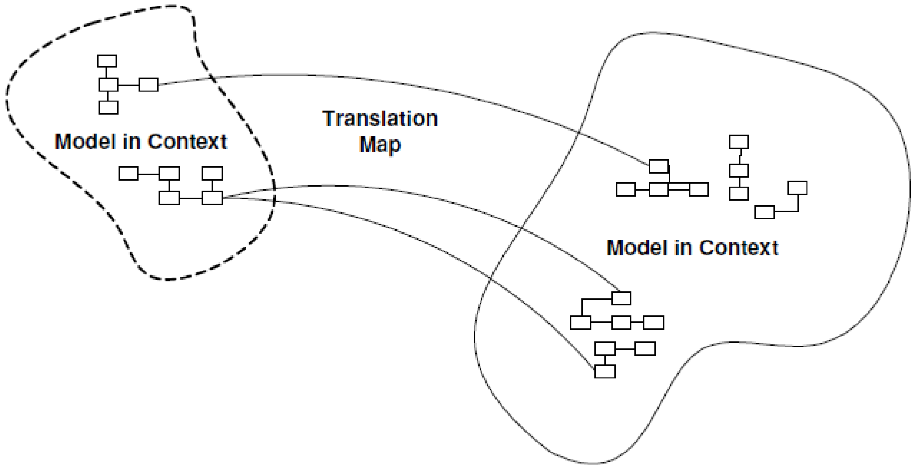


Figure 19 - Bounded Context Mapping
(Evans, 2003)

It is not enough to have separate unified models. They must be integrated because each model's functionality is just a part of the entire system. In the end, the pieces have to be assembled together, and the entire system must work properly (Evans, 2003).

Each Bounded Context should have a name that should be part of the Ubiquitous Language since it improves team communication when talking about the entire system. Everyone should know the boundaries of each context and the mapping between contexts and code. A common practice is to define the contexts, then create the modules for each context, and use a naming convention to indicate the context each module belongs to (Evans, 2003).

10. Domain Events

A domain event is something that has happened in the past in domain, that is wanted to be consumed in other parts of the same domain, for example, from other aggregates. The notified parts normally react to these events (Microsoft, 2018b).

Domain events help in expressing the domain rules, based in the Ubiquitous Language. These events also enable better separation of concerns among classes within the same domain.

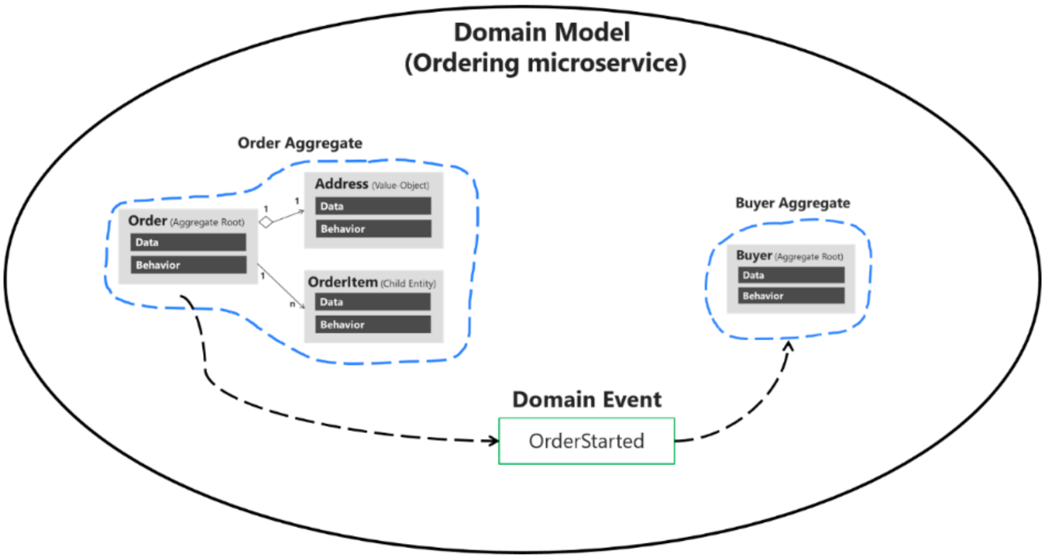


Figure 20 - Domain Event Example
(Microsoft, 2018b)

These events are similar to messaging-style events, with one important difference. With real messaging, message queuing, message brokers, or a service bus, a message is always sent asynchronously and communicated across processes and machines. This is helpful in multiple Bounded Contexts integration, microservices, or different applications. However, with domain events, an event is raised from a domain operation that is currently running, but that is intended to create any side effect in the same domain (Microsoft, 2018b).

The domain events and their side effects should occur as soon as possible, usually in-process, and within the same domain. Thus, these events can be synchronous or asynchronous. Integration events, however, should always be asynchronous (Microsoft, 2018b).

11. Domain Events vs Integration Events

Semantically, domain events and integration events are the same things: notifications about something that just happened. However, in terms of implementation, they are different. Domain events are messages pushed to a domain event dispatcher, which, therefore, can be implemented as an in-memory mediator (Microsoft, 2018b).

The purpose of integration events is to propagate transactions and updates to additional subsystems, whether they are other microservices, Bounded Contexts or even external applications. This is done by publishing integration events outside the microservice. They should occur only if the entity is successfully persisted, otherwise, it is as if the entire operation never happened. When an event is published to multiple receiver microservices, the appropriate event handler in each receiver microservice handles the event. These events must use asynchronous communication (Microsoft, 2018b).

When a domain event is thrown, it is possible to handle multiple actions related to other aggregates in the same domain, but it is also possible to run additional application actions, that are performed across microservices or bounded contexts, connecting these events with integration events and the event bus (Microsoft, 2018b).

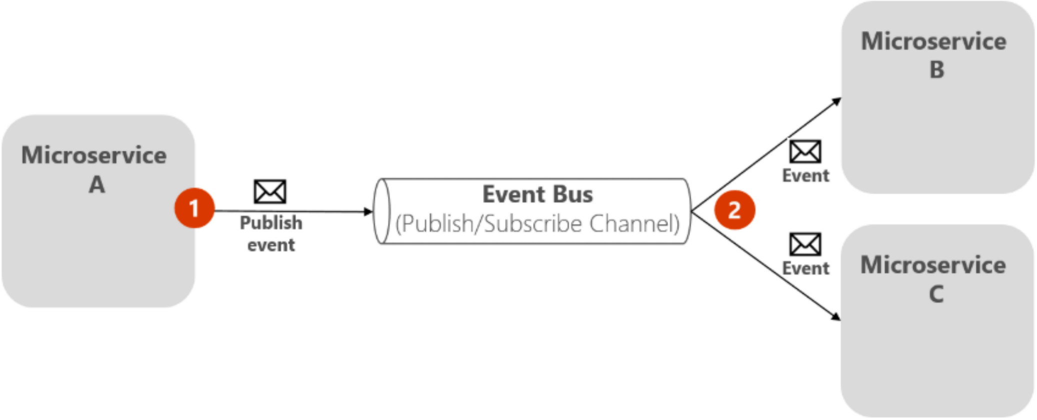


Figure 21 - Integration Event Example
(Microsoft, 2018b)

This event bus allows publish/subscribe communication between services without requiring the components to explicitly be aware of each other (Microsoft, 2018b).

12. Anemic Domain Model

Anemic Domain Model is the representation of the domain model in software using entities that contains little or no business logic associated, such as validations, calculations, business rules, etc. (Fowler, 2003).

Martin Fowler considers this as an anti-pattern that is getting more and more popular in the community. When a domain model is anemic, it looks like it represents the real business, with objects named after the nouns, being connected with rich relationships and structure that true domain models have (Fowler, 2003).

The problem is when in the domain model exists little or no behavior on its objects, making them as classes with only getters and setters. Rather than designing business rules in the domain model, these behaviors are captured by service objects, carrying out all the computation and updating the model objects with results. These services are located on the top of the domain model, using objects for data (Fowler, 2003).

13. Rich Domain Model

Rich Domain Model is a strategy that involves the definition of blocks such as Entities, Value Objects, and Aggregate Root, having the goal to build a common language between developers and stakeholders, using a vocabulary that describes the business rules (Vich, 2018).

The main difference to the Anemic Domain Model is that domain logic is part of domain entities, gathering data and behavior in domain objects. That logic guides and controls how the entity is instantiated, validated and operated, preventing constraints of having entities with an inconsistent state (Mota, 2018).

14. Domain Model Strategy for Microservices

When building a microservice or bounded context, it is necessary to define a domain model strategy that must be followed during its implementation.

If microservice is simple like a CRUD service, then anemic domain model using entity objects with only data properties might be a good approach, not being worth to implement DDD patterns (Microsoft, 2018a), since it would require more time-consuming. In that specific case, it will be a persistence model, because it was created only to carry data for CRUD purposes (Microsoft, 2018a).

Because of that, microservices architectures are perfect for a multi-architectural approach, depending on each bounded context. Although the anemic domain model is considered an anti-pattern for some people, it really depends on the way the implementation is done. If microservice is simple enough, such as CRUD, using anemic domain model is not an anti-pattern (Microsoft, 2018a). However, if business rules are existent and if it is necessary to implement complex logic about the domain, then it might be considered as an anti-pattern for that bounded context. In this case, designing a rich domain model, with entities containing data that is directly connected with behavior, and using DDD patterns, such as aggregates and value objects, might have a significant benefit for the long-term success of microservice (Microsoft, 2018a).

2.3 Problem Detail

As referred before, the software is continually increasing and evolving, and Occurrences is an example of that. With new business requirements over time, several constraints started to appear. These constraints are detailed in this section in order to illustrate the problem.

In the domain, business concepts and relationships are shared between Order and Return Occurrences. This model is located inside ODS service, in a monolithic system, and the next figure represents how it is built.

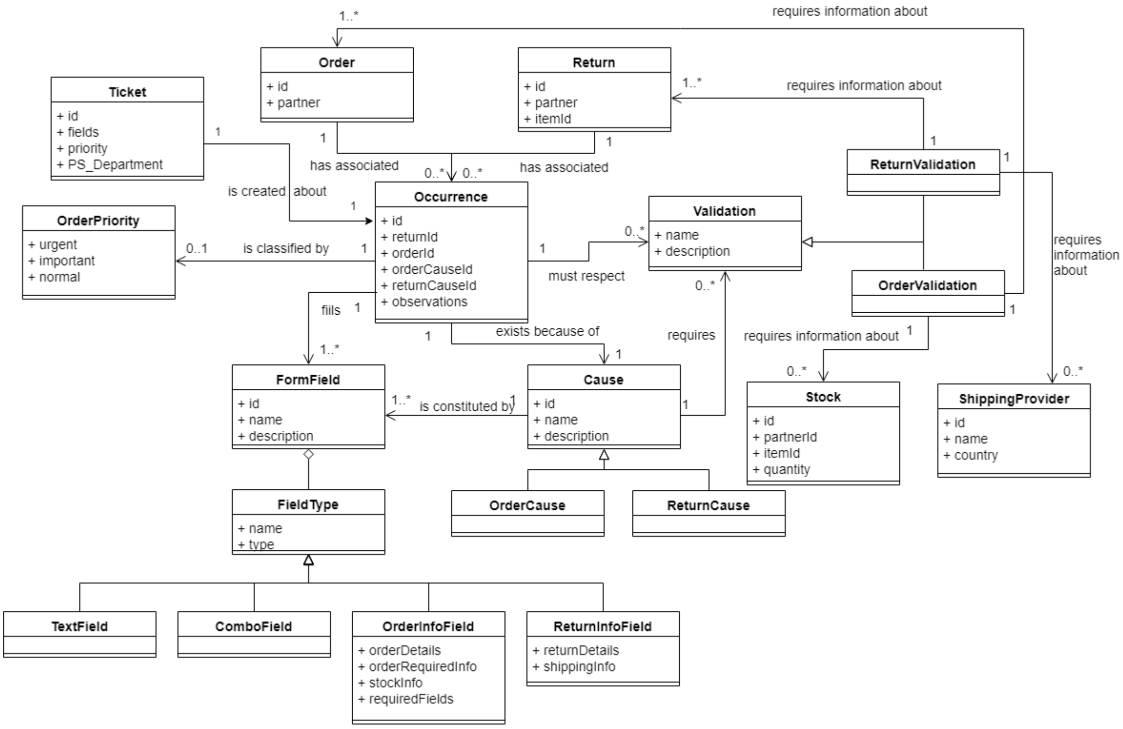


Figure 22 - Occurrences Domain Model

In the next table, it is possible to detail the domain model concepts:

Table 3 - Occurrences Domain Model concepts

Concepts	Details
Order	The order that is performed by the customer.
Return	Return that is performed by the customer.
Occurrence	An occurrence that is created for an associated order or return.
Order Validation	Required validation for an order occurrence related to a specific order. It gathers information in order to validate the inserted data about the occurrence.
Return Validation	Required validation for a return occurrence related to a specific return. It gathers information in order to validate the inserted data about the occurrence.
Cause	Cause that defines why order or return occurrence was created.
FormField	The field that gathers information related to an order or return occurrence.
FieldType	Type of the form field. Specific types are defined for order occurrences and return occurrences.
OrderPriority	Defined priority only for a related order occurrence: Urgent, Important and Normal.
Stock	Information about stock for an order.
Shipping Provider	Information about shipping provider for a return.

Ticket	Ticket related to the occurrence of being analyzed by the customer service department.

In the next table, it is possible to explain the domain model relationships:

Table 4 - Occurrences Domain Model relationships

Source	**Destination**	**Detail**
Order	Occurrence	One order might have multiple order occurrences associated with it.
Return	Occurrence	One return might have multiple return occurrences associated with it.
Occurrence	Cause	Occurrence is created because of a cause.
Occurrence	FormField	The occurrence must fill the form fields that were required by the cause.
Cause	FormField	The cause is constituted at least by a form field, so it can gather specific types of information from an occurrence.
Occurrence	Validation	When the occurrence is created it must respect the validations that were required by the cause.
Occurrence	OrderPriority	Occurrence related to orders is classified with a priority. This priority defines what is the urgency that occurrence must be resolved.
Cause	Validation	A cause might require order or return validations at the moment the occurrence is created.
ReturnValidation	Return	Return validation requires information about one or multiple returns in a way to validate information about the occurrence.
ReturnValidation	ShippingProvider	Return validation might have the need to read information about the shipping provider that is being used for the return.
OrderValidation	Order	Order validation requires information about one or multiple orders in a way to validate information about the occurrence.
OrderValidation	Stock	Order validation might have the need to read information about the stock information that is being used for the order.
Ticket	Occurrence	One ticket is created related to one occurrence. This ticket will be further analyzed and solved by the partner service-specific department.

This domain model has concepts and relationships that somehow represents the business, but it does not contain any behavior in its objects. Instead, application services, which are implemented over the domain, are the responsible ones for applying logic and rules related to business, coordinating the behaviors related to Order and Return Occurrences. In the next figure is presented a sequence diagram, with a high-level detailing, that demonstrates at the moment of saving an occurrence, how application services are responsible for applying business rules over the domain.

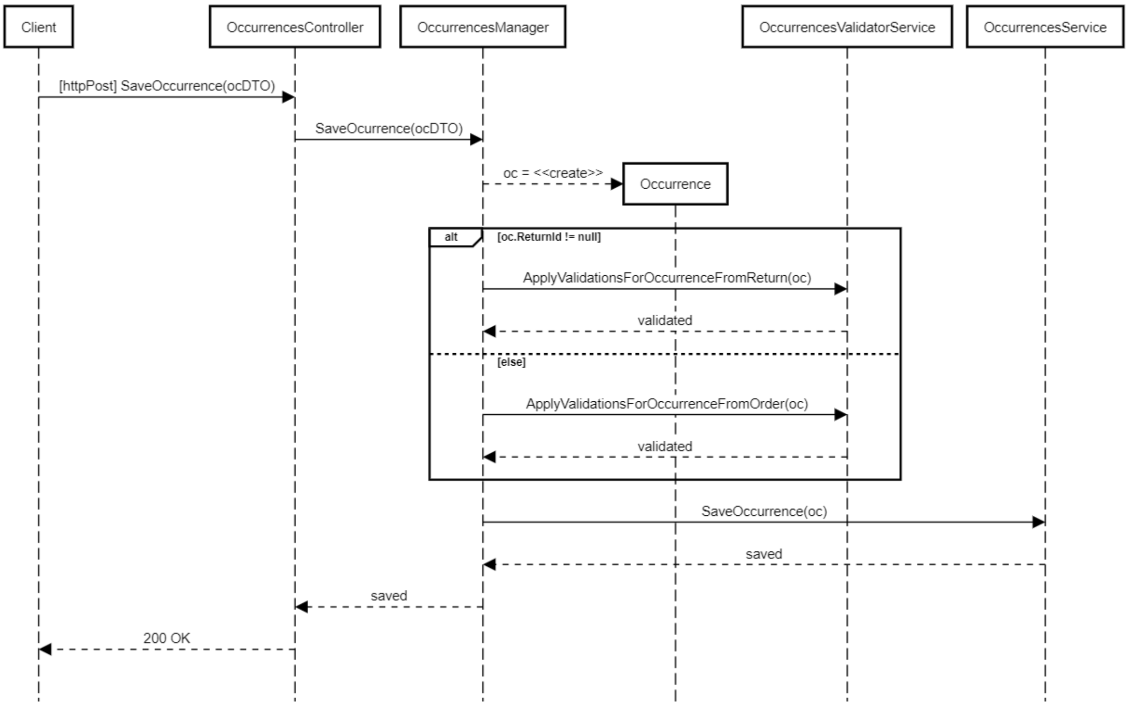


Figure 23 - Application Services with business logic

In the previous diagram, it is possible to verify that the domain Occurrence object is created from the DTO (Data Transfer Object), and after that, validations for occurrences are applied in application service. Besides that, it is verified if the Occurrence contains a ReturnId just to apply specific business logic related to Order or Return Occurrences. This means that sharing the domain model forces the execution of this kind of verifications, to make different business decisions depending on if an occurrence is from an order or from a return.

With that, this model is considered as an Anemic Domain Model, which exists no behavior on its objects, having only getters and setters in the classes, carrying the data information over the system, being controlled by service objects.

```
public class Occurrence
{
    public string Id { get; set; }

    public string ReturnId { get; set; }

    public string OrderId { get; set; }

    public string OrderCauseId { get; set; }

    public string ReturnCauseId { get; set; }

    public string Observations { get; set; }
}
```

Code 1 – Anemic domain model object

In the beginning, this was not a problem, since software for occurrences was like a CRUD, and model only had the goal to represent the business and to carry its information as a persistence model. But with the platform evolution, specific requirements for orders and returns emerged. With this domain model strategy, domain logics are spread over the application services of the monolithic system, and sharing the model for two businesses (orders and returns), which are taking different paths in the business, creates even more constraints.

For developers, it is difficult the understanding of business concepts and rules in the code, requiring extra efforts in the moment of evolving the domain for new business logic or changes. Having the domain logics spread over the services inside a gigantic monolithic is the main reason for those constraints since entities and relationships should be completely common.

Main domain classes related to Occurrences are shared for both Order and Return businesses, but other classes that are only specific for Order Occurrences, or for Return Occurrences, are being built and mixed with the previous ones.

Using the domain model from figure 22, in the next table, the scope of domain business concepts are classified as specific for orders, for returns or in common. This will provide visibility about how the domain model is not supporting a clear understanding of business in the code:

Table 5 - Domain Model Entities scopes

Scope	Concepts
Order	Order, OrderPriority, OrderCause, OrderInfoField, OrderValidation, Stock
Return	Return, ReturnCause, ReturnInfoField, ReturnValidation, ShippingProvider
Shared	Occurrence, Cause, FormField, FieldType, Validation, TextField, ComboField

In the previous table, it is possible to evaluate that the core concepts for Occurrences are shared, and the other ones are specific for orders or returns. Since main concepts are in common for both businesses, is required to take a double attention when developing new requirements to Order Occurrences or Return Occurrences, since one change in Order Occurrences can affect directly Return Occurrences, and vice-versa, creating the need to test and do all the regression to both businesses to make sure the system requirements are being met.

3 Value Analysis

Value analysis is defined as a systematic analysis and evaluation of applied techniques and functions in multiple areas of a specific problem, with the objective to explore improvements that might improve a particular product value (Singh, 2013).

It involves an innovative approach to discover what are the unnecessary costs. These costs occur on a product or service and are not needed, having any impact on efficiency or quality, neither provide any additional value to the customer. Removing these costs, the associated cost of product or service can be reduced, increasing profits during sales (Singh, 2013).

Value analysis is considered an effective tool for cost reduction, economizing expenditure and increasing productivity, whereas value analysis probes into economic attributes for value. It is possible to improve performance, increase product value and reduce associated costs, applying a continuous process for a planned action (Singh, 2013).

In the next sections, the value analysis for the solution of this project is detailed using an appropriate method.

3.1 New Concept Development Model (NCD)

New Concept Development (NCD) Model provides a common language and definition of key components associated with the fuzzy front end (FFE). The engine (culture, leadership and business strategy) represents senior and executive-level management support, powering five elements of the model (Koen *et al.*, 2001):

- Opportunity Identification;
- Opportunity Analysis;
- Idea Generation and Enrichment;
- Idea Selection;

- Concept Definition.

The engine and these five elements are placed on top of influencing factors. The circular shape of the NCD model suggests that ideas and concepts are expected to be iterated over the five elements. The arrows pointing to inside the model represent starting points, indicating that projects begin at either opportunity identification or idea generation and enrichment. The exit arrow represents how concepts leave the model and enter a new product development process (Koen *et al.*, 2001). This definition is represented by the next figure:

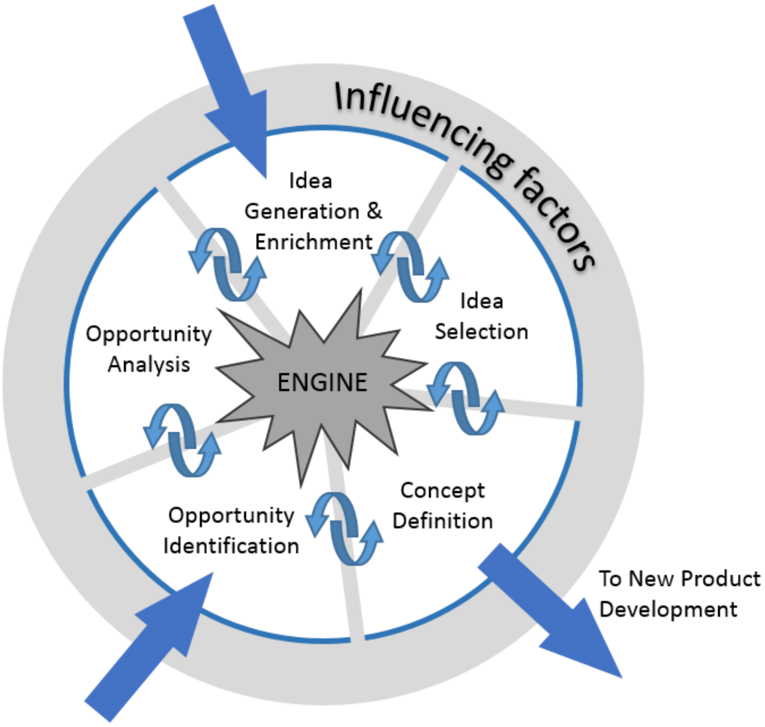

Influencing factors
Idea Generation & Enrichment
Idea Selection
Opportunity Analysis
ENGINE
Concept Definition
Opportunity Identification
To New Product Development

Figure 24 - NCD Model
(Koen et al., 2001)

The next topics will explain the interaction of each element of the NCD model with this project.

3.1.1 Opportunity Identification

Opportunity Identification is the element responsible for identifying opportunities that an organization might want to pursue. Opportunities such as business and technological are explicitly considered so that resources can be allocated into new areas of market growth, operating effectiveness, and efficiency (Koen *et al.*, 2001).

The solution applied for the contextualized problem arises from organization structural changes related to software architecture.

The continuous migration of monolithic applications into microservices is being performed in the organization. This means that the company is investing in software evolution in order to scale its platform for more customers and sales. Each team is responsible to implement its microservices as desired, taking into consideration the organization guidelines, and using technologies that are compatible and supported by the infrastructure.

For that, it is an opportunity for “ST” team to migrate Occurrences functionalities, from monolithic to a microservices oriented architecture, having the opportunity to solve the domain constraints during this transition.

3.1.2 Opportunity Analysis

Opportunity Analysis is the element in which an opportunity is assessed to confirm if it is worth to pursue. For this, is necessary to obtain additional information to translate the opportunity identification into specific business and technology opportunities. This involves the need to make early and often uncertain technology and market assessments (Koen *et al.*, 2001).

The topic of the previous section, which is related to the migration to microservices in the organization overall, is analyzed in order to verify and describe the opportunities that it can accomplish.

As previous related, the domain model related to Order and Return Occurrences is creating constraints in the developers' business understanding, and it is difficult to evolve for new business requirements.

Having this, the microservices architecture approach is an opportunity to evolve Occurrences software, in order to solve the domain constraints during the migration, with the help provided by a Domain Driven Design strategy.

3.1.3 Idea Generation and Enrichment

Idea Generation and Enrichment concern the birth, development, and maturation of a concrete idea. In this element, ideas are built up, torn down, combined and modified. An idea can step through many iterations or changes while it is studied, discussed and developed with other elements of the NCD model (Koen *et al.*, 2001).

In this section, it is presented the ideas that are formed to follow a solution strategy, selecting the most adequate one for this project problem.

3.1.3.1 MD1 Idea

The first idea, that hereafter is identified as “MD1”, is the creation of a generic microservice with a domain capable to be associated with every occurrence in the platform. This means, it would be prepared for the Order and Return Occurrences, but also would be prepared for occurrences in another business different from orders and returns, having the responsibility to manage all types of occurrences in the overall platform.

With this, the domain would only represent the business concepts and behavior associated with occurrences, without representing any other terms that connect with it such as orders, returns, etc. This solution would be as clean as possible, and the responsibility to manage business rules that relates Occurrences with Orders or Returns would be handled inside the corresponding services, which are responsible for those orders or returns. The microservice would be built following a Domain Driven Design approach, with a Rich Domain model that implements concepts and behavior associated with occurrences directly on classes related to the domain.

This idea has the goal to create a completely generic domain that is prepared to have occurrences associated for other businesses, for example, occurrences in product management, without having the need to implement a new service, and having only the need to apply the business rules in the service responsible for product management. On the other hand, this approach is significantly complex and must be handled carefully, in a way to coordinate all the rules associated with the occurrences and the corresponding business.

3.1.3.2 MD2 Idea

The second idea, that hereafter is identified as “MD2”, is the distribution of the different business contexts into microservices.

For that, it is necessary to delineate the business contexts that must be migrated and to define which microservices are necessary to be implemented. Each microservice would be built following a Domain Driven Design approach, with a Rich Domain model that implements concepts and behavior associated with its business directly on classes related to the domain.

This idea has the goal to create the necessary microservices to manage different business contexts. With that, concepts, relationships, and behaviors of each domain would be entirely dedicated and specific for its business context. On the other hand, microservices must handle coherent contexts, which might be complex to identify and have the risk to create unnecessary services, adding complexity to the platform.

3.1.3.3 MD3 Idea

The third idea, that hereafter is identified as “MD3”, is the creation of a new version of the domain model inside the same monolithic service.

For that, there is no need to create new services, but to create a new version of the domain model in the same monolithic service, reducing efforts of microservices development. This idea would be built following a Domain Driven Design approach, with a Rich Domain model that implements concepts and behavior associated with each business (Order Occurrences and Return Occurrences) directly on classes related to the domain.

With this, the domain would be refactored in a way to represent correctly the business concepts and behavior associated with Order and Return Occurrences. Business understanding of the application code would be improved to overcome the difficulties that are announced in the problem.

This idea has the goal to improve the domain without much development efforts. On the other hand, this idea has a poor software evolution strategy, countering the microservices trend existent in the platform, keeping the business of Occurrences attached to monolithic.

3.1.4 Idea Selection

Idea Selection is the selection of ideas to pursue in a way to achieve the most value for the business. Perform a good selection is critical for the success of the business. For that, there is no single process that will guarantee a good selection. Normally, Idea selection involves an iterative series of activities that includes multiple passes through opportunity identification, opportunity analysis, idea generation, and enrichment, often with new insights from the influencing factors and new directives from the engine (Koen *et al.*, 2001).

Analytic Hierarchy Process (AHP) is the chosen method to select the idea for the project solution. This method derives relative scales using judgment or data from a standard scale, and how to perform the subsequent arithmetic operation on such scales (Linford, 1994).

One of the benefits of its usage is that it allows the focus to judge separately on each of several properties to make a decision. The most effective way to concentrate judgment is to take a pair of elements and compare them on a single property without concern for other properties or other elements (Linford, 1994).

The next figure illustrates the first step of how this method is applied to this context, where the goal and corresponding criteria are exposed to select the most valuable idea for the project solution:

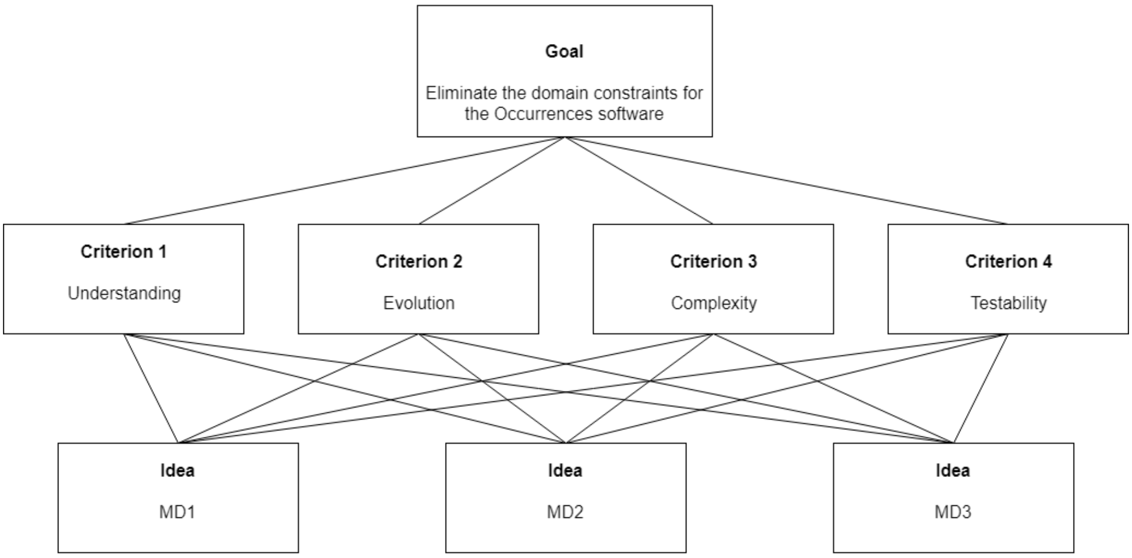


Figure 25 - AHP hierarchy for Idea Selection

The first layer represents the goal that is intended to achieve, which is the elimination of the domain constraints for the Occurrences software.

The second layer represents the criteria which contribute to the goal:

1. **Understanding**: This criterion defines how easy the domain is understood by developers, about its business, concepts, relationships, and behavior;
2. **Evolution**: This criterion defines how easy the domain is, for developers, to evolve software for new business requirements and changes;
3. **Complexity**: This criterion defines how simple is the domain;
4. **Testability**: This criterion defines how much the domain facilitates the testing of system requirements.

The third layer represents the defined ideas as alternatives to reach the target goal. With this hierarchy, it is possible to evaluate the best approach to follow as a solution to the problem.

Before presenting the next step of AHP, comparison of criteria will be accomplished using values from one to nine, using Saaty's (Saaty, 2012) as a numerical scale.

Table 6 - Saaty's numerical scale
(Saaty, 2012)

Verbal judgment	Numeric Value
Extremely important	9
	8
Very Strongly more important	7
	6
Strongly more important	5
	4
Moderately more important	3
	2
Equally important	1

The next table represents the second step of AHP, using a matrix with a pairwise comparison of criteria to define what values are the most important to be accomplished in order to reach the goal, setting an evaluation for each comparison, defining a final sum for each criterion. For this, the previous scale (Saaty, 2012) will be used.

Table 7 - AHP Criteria comparison

Goal	Understanding	Evolution	Complexity	Testability
Understanding	1	0.5	3	4
Evolution	2	1	3	4
Complexity	0.333	0.333	1	2
Testability	0.25	0.25	0.5	1
Sum	3.583	2.083	7.5	11

After having the matrix with the corresponding values, it is possible to normalize it in order to obtain the overall criteria priorities. Normalization can be applied by adding the values in each column, and after that, dividing each cell by the total of the column. Therefore, it is possible to obtain priorities by calculating the average value of each row.

Table 8 - AHP Normalized comparison

Goal	Understanding	Evolution	Complexity	Testability	**Average**
Understanding	0.279	0.240	0.400	0.364	0.320
Evolution	0.558	0.480	0.400	0.364	0.451
Complexity	0.093	0.160	0.133	0.182	0.142
Testability	0.070	0.120	0.067	0.090	0.087
Sum	1	1	1	1	1

With the previously calculated values, it is possible to claim which criteria are the most priority:

Table 9 - Criteria Priorities

Priority Position	**Criterion**	**Priority Rate**
1	Evolution	45.1%
2	Understanding	32%
3	Complexity	14.2%
4	Testability	8.7%

It is possible to conclude that Evolution and Understanding are the main factors to be considered to choose a solution idea. Complexity and Testing are still relevant but are not weighted as important criteria.

All ideas agree with a Domain Driven Design approach, but the idea MD3 does not implements a microservices oriented architecture, which compromises the domain evolution, but also manages to couple the domain concepts and behavior to the monolithic domain, creating difficulties to its understanding.

Looking forward, the idea MD1, contrary to MD3, implements a microservices architecture but with a very generic and restrict business context. Creating a microservice to manage all kind of occurrences in platform might not favor its domain evolution, since new business concepts and behavior would be tough develop, being spread over other services (monolithic and microservice ones), not having a service that connects the Occurrences with its business context (such as Orders or Returns for example), difficulting its understanding in the platform.

Lastly, the MD2 solution implements a microservices architecture, identifying the business contexts to be migrated into different services, where each one concentrates its business concepts and behavior in its domain. This idea is supported by the Bounded Context approach of Domain Driven Design, and delineating the bounded contexts and its microservices, would allow each service to hold a clear domain delineated by its context, improving its business understanding, but also allowing the domain to be easier to evolve inside its context. Having this, it is possible to conclude that MD2 is the most compatible idea for Evolution and Understanding factors.

3.1.5 Concept Definition

Concept Definition is considered as the final element of the NCD model, which provides the only exit to this model. For this, the innovator must make a compelling case for investment in the business or technology proposition. This case consists of both qualitative and quantitative information, which gatekeepers use to make a determination (Koen *et al.*, 2001).

After idea selection, it is possible to define a process that must be executed to reach success in the provided solution. To approach this, it is necessary to follow the next steps:

1. Considering the business contexts, delineate the necessary microservices to be implemented, using Domain Driven Design to support it;
2. Design and implement technical aspects related to Domain Driven Design, in order to achieve a rich domain for each microservice;
3. Evaluate if the applied solution solved the project problem.

3.2 Value Proposition

Value Proposition is the definition of a business statement that a company uses to specify why a customer should use a service or buy a product. This statement tries to convince the consumer that one product or service adds more value or solves better its problem than the similar offerings (Kenton, 2018).

This project has the mission to provide a new solution to solve the Occurrences domain constraints, so the understanding of business concepts and rules, in the code, can be clear to developers, and to prepare the microservices domains to facilitate the software evolution for new functionalities and changes required for the business.

3.3 Business Model Canvas

Business Model Canvas is a visual representation of an existent or new business model, usually performed by strategic managers. This model provides a holistic view of the business as a whole, and it is especially useful in running a comparative analysis of the impact of an increase in investment that may have on any of the contributing factors. It provides a common language for the community so they can evaluate traditional processes and bring innovation into their business models (Osterwalder, 2010).

Business Model Canvas categorizes processes and internal activities of the target business into 9 separate blocks, each one representing a building block in the creation of the product or service (Osterwalder, 2010):

1. **Customer Segments**: segments based on the way an organization's products or services address a specific need;
2. **Value Propositions**: a combination of products and services the organization provides to its customers;
3. **Channels**: medium through which an organization provides its value proposition to its customer segment;
4. **Customer Relationships**: relationships that an organization will have with its customer segment in order to create financial success and sustainability;
5. **Revenue Streams**: a methodology that an organization follows to get its customer segments to buy its product or service;
6. **Key Resources**: organization's assets that are fundamental to know how it provides value to its customer segment. Resources can be categorized as human, financial, physical and intellectual;
7. **Key Activities**: key activities to producing the company's value proposition;
8. **Key Partners**: a network of suppliers and partners who complement each other in helping the organization to create its value proposition;
9. **Cost Structure**: costs associated with running a business according to a particular model.

Therefore, a Business Model Canvas is applied to this project context, in order to describe its business idea applied to Occurrences software.

Table 10 - Business Model Canvas

<table>
<tr>
<td rowspan="2">**Key Partners**
- Platform
- Development team</td>
<td>**Key Activities**
- Delineate the necessary microservices to be implemented
- Design and implement technical aspects related to Domain Driven Design</td>
<td rowspan="2">**Value Propositions**
- Domain with high capacity to evolve for new business requirements
- Implemented business requirements easy to understand on the domain</td>
<td>**Customer Relationships**
- Meetings with developers to design and build the domain of each microservice
- Meetings with the product owner to clarify doubts about the business</td>
<td rowspan="2">**Customer Segments**
- Development team</td>
</tr>
<tr>
<td>**Key Resources**
- The development team will follow and validate the solution approach</td>
<td>**Channels**
- Microservices documentation
- Wiki platform</td>
</tr>
<tr>
<td colspan="2">**Cost Structure**
- Developers
- Infrastructure costs</td>
<td colspan="3">**Revenue Streams**
- Provide a clear understanding of business in the domain
- Provide microservices with domains that are capable to evolve software without difficulties</td>
</tr>
</table>

4 Analysis and Design

This chapter presents the analysis and design of the solution to be implemented, explaining the approaches related to Domain Driven Design, that the solution should implement in order to solve the constraints of the project problem.

4.1 Domain Driven Design

The Domain Driven Design approach was chosen to be followed since the problem is related to business understanding and evolution capacity. DDD is a mature approach and it contains patterns capable to solve the referred constraints related to the project problem.

DDD allows the team to focus mainly on the domain, which was not considered before, during software development on Occurrences software. With that, it is expected that this strategy defines the number of microservices to be implemented by analyzing the bounded contexts (section 2.2.4). Overall, DDD should enrich the microservices domain with behavior, that was missing in the project problem.

With the definition of a Ubiquitous Language for the bounded contexts, it is expected the increase in domain understanding by the team developers and product owners.

Domain Driven Design provides many patterns that team elements are concerned about, such as entities, value objects, services, repositories, layered architecture, which some are already implemented by the team elements in the projects. Besides that, most every team element has some knowledge about DDD, which helps in the building of these services following correctly the aspects related to Domain Driven Design.

Therefore, Domain Driven Design can be integrated with some software architectures, such as CQRS or Clean Architecture. With this, the team elements have an opportunity to evolve its knowledge and experience about DDD and other patterns, embracing a project with a domain that is considered complex, but not highly complex, which is interesting learning for the team.

4.2 Architectural Design with Context Mapping

When talking about Architectural Solution Design, it is helpful to think about Bounded Context. It provides the logical frame inside of which the model evolves. When having multiple models, it is necessary to define the borders and the relationships between them. Each one having its own Bounded Context (Evans, 2003).

Analyzing the business related to the domain model provided in figure 22 from section 2.3, it is possible to gather its concepts and to delineate multiple bounded contexts associated with it.

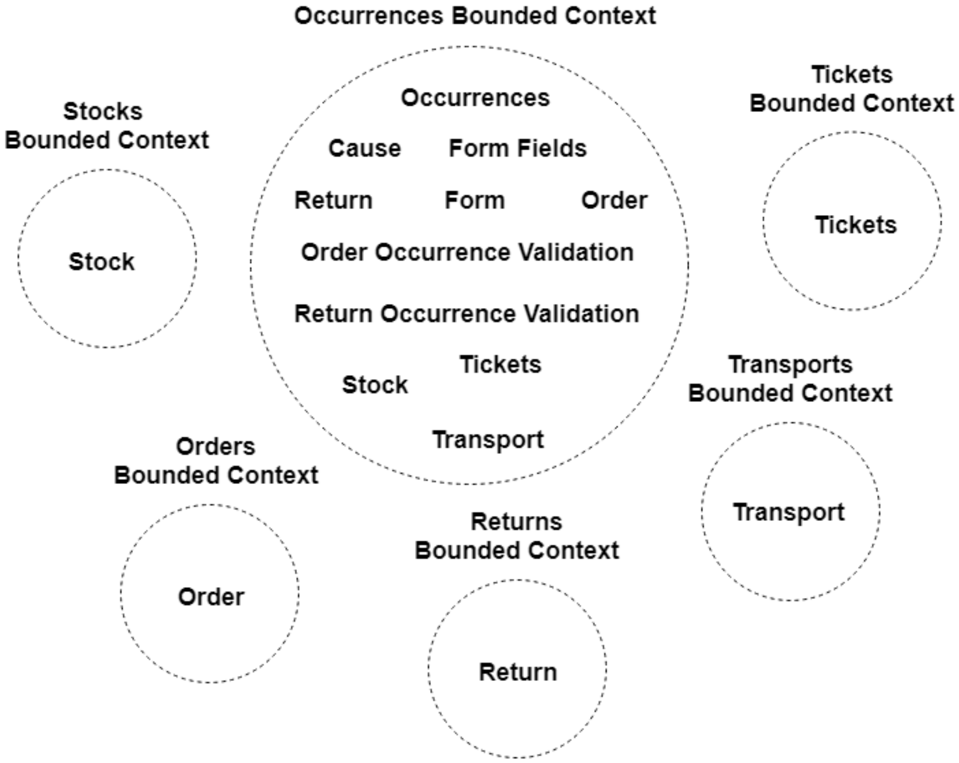


Figure 26 - Bounded Contexts related to Occurrences concepts

Occurrences are related to multiple business concepts, such as Orders, Returns, Tickets, and others. Many of these concepts might be defined in different bounded contexts. For example, an Order might be a concept delineated in Orders bounded context, and also in Occurrences bounded context. However, this concept differs between them, because in Orders context, the aspects of the order that matters are the attributes used for order management, while in Occurrences context the attributes are used to identify the order that the Occurrence was created from.

In the previous figure, it is possible to verify that many bounded contexts might exist with the concepts related to Occurrences, however, the Occurrences bounded context is not well delineated. Its domain model is being shared for Orders and for Returns, trying to decouple Occurrences concepts from Orders and Returns without success.

It does not succeed because Occurrences concepts cannot be generic, since they are associated with other concepts directly related to Orders and Returns, causing constraints with domain business understanding and evolution capacity. For example, an Order Occurrence was created by an order-specific cause, and its validation needs information about Stock. While a Return Occurrence was created by a return-specific cause and its validation needs information about Transport.

Order Occurrences should have been implemented, at least, in Orders bounded context, while Return Occurrences should follow the same strategy, by being implemented in Returns bounded context. Implementing one bounded context for Occurrences with the expectations that it would support the domain with Orders and Returns, was not a good idea, causing the constraints that were already referred before.

Having this, the idea to solve the problem constraints is to brake the Occurrences bounded context into two new ones: Order Occurrences and Return Occurrences contexts.

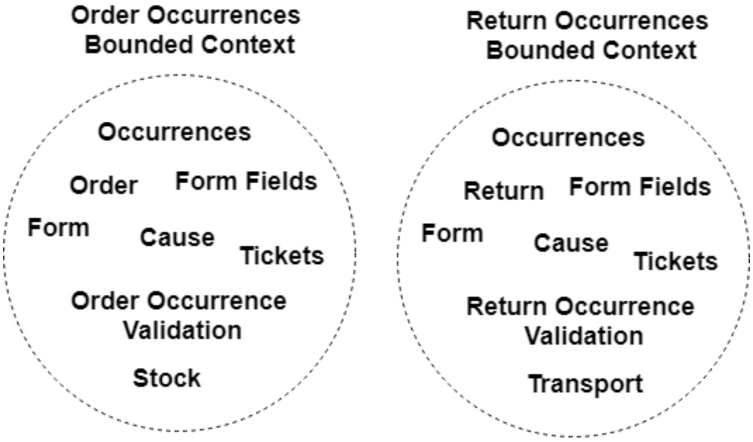

Order Occurrences Bounded Context
Occurrences
Order
Form Fields
Form
Cause
Tickets
Order Occurrence Validation
Stock
Return Occurrences Bounded Context
Occurrences
Return
Form Fields
Form
Cause
Tickets
Return Occurrence Validation
Transport

Figure 27 - Order Occurrences and Return Occurrences bounded contexts

With this approach, it is not necessary to migrate Occurrences related with Orders to the Orders bounded context, neither to migrate Occurrences related with Returns to the Returns bounded context. Currently, Orders and Returns contexts are already overloaded, having many concepts and behavior with complex domains. Adding Occurrences context to them would only create even more complexity, and possibly would not have benefits and not solving the problem about this project.

Delineating two different bounded contexts allows focussing on each business scope, simplifying the implemented domains, since each context only needs to implement concepts with the minimum information. For example, Order Occurrences context only needs the basic information of an Order and a Stock, while Return Occurrences context only needs the basic information of a Return and a Transport.

Following this strategy, each bounded context has a size that fits perfectly on a microservice. And since the platform is following a microservices oriented architecture trend, the idea would be to implement one microservice for each context: Order Occurrences Service and Return Occurrences Service.

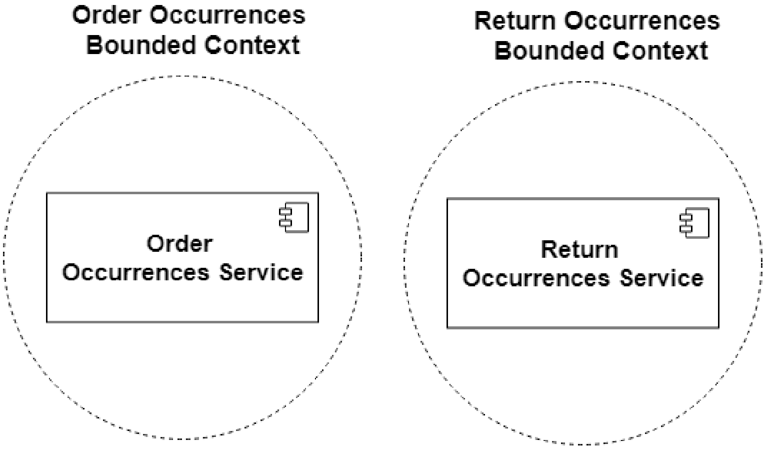

Order Occurrences Bounded Context
Return Occurrences Bounded Context
Order Occurrences Service
Return Occurrences Service

Figure 28 - Order Occurrences and Return Occurrences services

Each microservice implements its related bounded context, and it is completely independent. Even though they have identical concepts, each one defines its Ubiquitous Language, being also capable of evolute its domain by introducing new business concepts or behavior, related to the business context in which it is inserted.

For possible new business requirements, each bounded context might have more implemented microservices. The development of those services in each bounded context might be evaluated taking into consideration the domain concepts and rules, that are necessary to achieve the business requirements to be implemented.

Furthermore, the team can be fully responsible for the Order Occurrences and Return Occurrences services, since the related bounded contexts have a size that both can be handled by a single team.

4.3 Software Architecture

Having defined two independent microservices, each one has the freedom to be implemented with its own domain and architecture solution. But there are some restrictions:

- Organization infrastructure supports it;
- Elements of Architecture Team approves it.

With that, it is possible to define the same software architecture solution for both services, without any problem.

Using the same software architecture solution would speed up the developments for both services since the team would be familiar with it. But in contrast, the team would not be able to learn and explore more architecture implementations, which is their interest. With this, each microservice will implement a different software architecture solution, but taking always in consideration the DDD.

Next, the software architecture solution for Order Occurrences and Return Occurrences services are defined and detailed.

4.3.1 Order Occurrences Service

Order Occurrences service will implement a Layered Architecture from Domain Driven Design, with four main layers: **Presentation, Application, Domain**, **Infrastructure**. Each layer responsibility was already explained in table 2 from section 2.2.4, but Presentation Layer will display a REST interface, instead of presenting information for the users, for example, as a Frontend application.

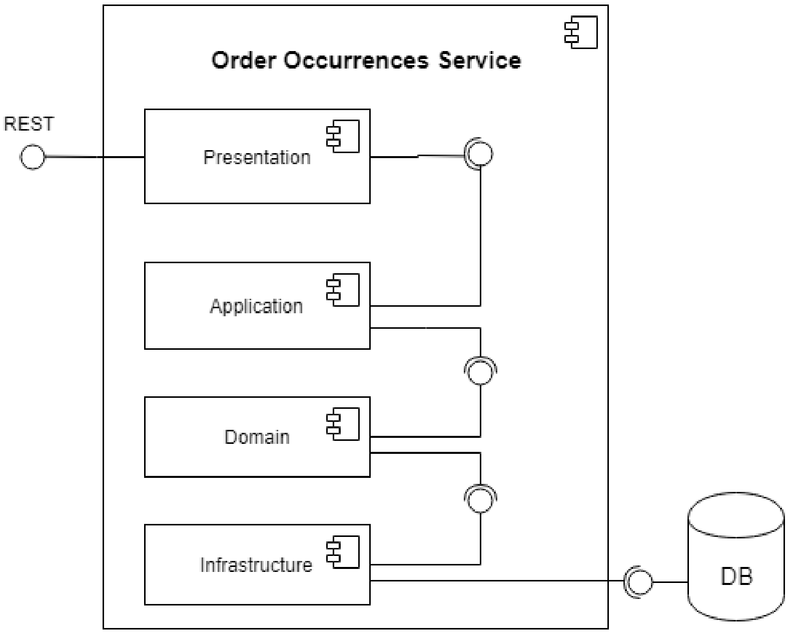


Figure 29 - Order Occurrences Service Architecture

Since Layered Architecture is very emphasized with Domain Driven Design, it is expected that the DDD approach will be easier to implement using this architecture solution.

This service will be also capable to publish/subscribe domain events between aggregates and to publish/subscribe integration events with an event bus, to communicate with outside, such as other microservices or bounded contexts.

4.3.2 Return Occurrences Service

Return Occurrences service will implement a Clean Architecture (section 2.2.3.5) with CQRS (section 2.2.3.4), with four main layers:

- **Presentation**: Similar to Presentation from Layered Architecture, this layer is responsible for displaying information to be consumed via REST.
- **Use Cases:** This layer contains the application-specific Use Cases. These orchestrate the flow of data to and from the entities and direct those entities to use their business rules to achieve the goals of the Use Case. It does not contain business logic, neither hold the state of business objects, but it can hold the state of an application task progress.
- **Domain**: This layer is responsible to encapsulate business rules. It can be an object with methods, or it can be a set of data structures and functions.
- **Data**: This layer is responsible for a set of adapters to convert data from the format most convenient for the use cases and entities, to the format most convenient for some external software such as database or service.

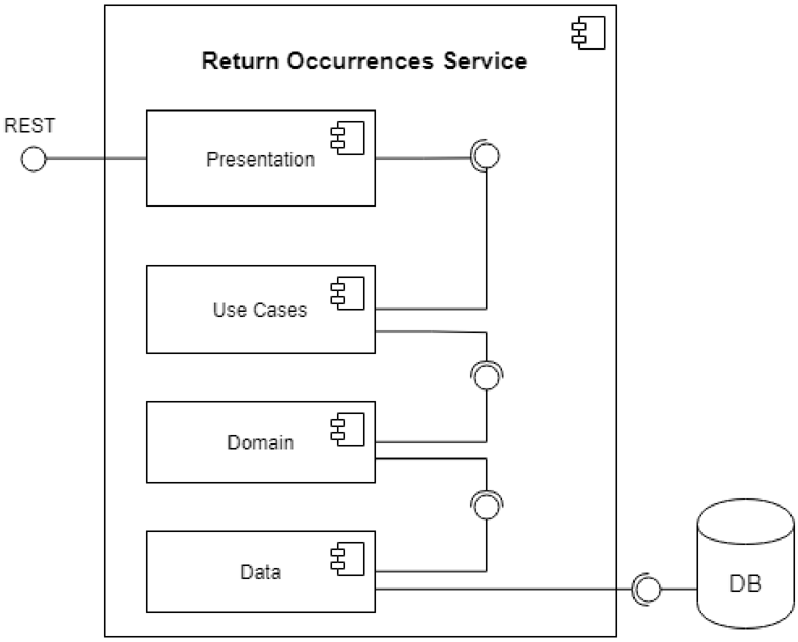


Figure 30 - Return Occurrences Service Architecture

Even using a Clean Architecture with CQRS, DDD approach must be applied in the same way, taking its concepts into consideration during this service development. It is not expected that this software architecture solution affects the quality of DDD implementation since its concepts can be applied without constraints.

If necessary, this service will be capable to publish/subscribe domain events between aggregates and to publish/subscribe integration events with an event bus, to communicate with outside, such as other microservices or bounded contexts.

4.4 Building a Ubiquitous Language

As referred before, building a Ubiquitous Language has a clear outcome: the model and the language are strongly interconnected with one another, and a change in the language should become a change to the model. This Ubiquitous Language connects all the parts of the design, creating the premise for the design team to work with success (Evans, 2003).

In this section, concepts about Order Occurrences and Return Occurrences bounded contexts are defined and clarified. With this, it is possible to build a domain model using these concepts, which are used in the communication between the team itself and with product owners, taking advantage of the benefits that the Ubiquitous Language provides.

4.4.1 Order Occurrences

In this section, concepts about Order Occurrences bounded context are defined.

Order

Everyone in the company has a notion about the meaning of an order. However, as it was explained in section 2.1.2, one sales order may contain items from different merchants, so it is necessary to split it into different merchant orders, so these ones can be processed by the corresponding partners.

Order is a concept that is currently used in many bounded contexts, having different meanings. With that, it is important to keep its definition on Order Occurrences context:

"*Order is the merchant order that was separated from its original sales order, and which is being processed by the corresponding partner. The Order is used to identify the origin of an Order Occurrence.*"

Order Occurrence

Occurrence is a concept that is constantly spoken by the team and by product owners. But Occurrence is always a generic concept, that only has value if it derives from an Order or Return. With that, for this specific business context, it is defined that Order Occurrence is the right name for this concept:

"Order Occurrence is an anomaly that occurs during order processing, and which is reported by a partner."

Partner

The partner is an interesting concept because a partner is the same thing as a merchant. But the name "merchant" is a clearer concept than "partner". But both the team and product owners always refer to the "merchant" with "partner". With that, it is important to reinforce the characteristics of this concept, in order to represent in domain model in the same way that it is communicated inside the team and the company.

"Partner is the merchant responsible for order processing, which has the possibility to report occurrences about it."

Cause

The cause is another concept that is constantly spoken by the team and by product owners. "Reason" was an alternative name for this concept, but since "Cause" is already a settled in communications, it was defined to keep it.

"Cause describes the reason for an order processing anomaly, which justifies the generation of an order occurrence."

Form Template

The Form Template concept is currently used during communications, having a simple definition.

"The Form Template is a template that is required to be filled about a cause, during the creation of an order occurrence."

Form Template Field

The Form Template Field is related to a field that is required to be present in some form template.

"The Form Template Field is a field that must be present in some form template. This field might be available to gather information about an order occurrence, such as text, dates, numbers, files or other types of information."

Filled Form

This concept is related to a form that is filled during the Order Occurrence creation.

"A Filled Form is when a form related to a cause is filled during the order occurrence creation."

Form Field

The Form Field is already a known concept by all the elements, and its function is already well defined.

"A Form Field is a field that is filled during the order occurrence creation, which gathers information related to some attribute related to the cause, in the shape of text, dates, numbers, files or other types of information."

Priority

Priority is a concept that was not clear and that could be misunderstood by some team elements. This concept was known as "Order Priority", but that does not mean that has always to do with order.

This priority is about the Occurrence, and it is defined by the partner. A partner can define this priority because of the order importance, or just because a mistake needs to be solved urgently by the partner support team.

With that, "Priority" will be the chosen name for this concept, since it is not restricted to one context, and its meaning is only important for the Order Occurrence and nothing more else.

"The priority that a related Order Occurrence must be embraced by the partner support team."

Requirement

For a cause, exists defined requirements that must agree with the filled form fields, in order to have valid information inside the form about an Order Occurrence. This concept was hidden on service objects and created many difficulties in its understanding of the code by developers. It was a constraint for software evolution since it was difficult to maintain and to apply new business requirements.

"Requirement is a rule established for a specific cause, that must be in agreement with the filled form fields, in order to gather valid information about an order occurrence."

Order Occurrence Validation

This concept is related to Order Validation from the Occurrences domain model. Its name changed because of a reason: it does not mean that validation will be always directly about the order. This validation can be about other things, such as Stock. With that, this concept changed the name for "Order Occurrence Validation", because the validation is always about it.

An Order Occurrence to be valid must be according to numerous business rules. With that, naming this concept as "Order Validation" will always induce team members and product owners to think only about validations in order, when it is not only about it. It is expected that naming it "Order Occurrence Validation" will not create these confusions anymore.

"Order Occurrence Validation is a validation that must be executed about a defined business rule, related to Order Occurrences."

Ticket

The Ticket is a concept that is represented poorly on the existent domain model. It does not represent its importance and it is somehow disconnected from the remaining model. Besides not being the core of the model, this concept must be clearer and more representative.

“Ticket is a formal presentation of an order occurrence, that allows the support partner team to visualize it and solve it.”

Ticket Priority

The Ticket Priority is a concept that represents the urgency that the support partner team must embrace a specific ticket. Basically, it represents a formal representation of the Priority value.”

“Ticket Priority a formal representation of Priority, defining the urgency that the support partner team must embrace a specific ticket related to an order occurrence.”

Ticket Field

The Ticket Field is a concept that formally represents a field that was filled during the Order Occurrence creation.

“Ticket Field represents some field that was filled during the order occurrence creation, in order to its value be visualized by the support partner team.”

Stock

The Stock is a concept that is associated with an order and a partner. It represents the stock that a partner should have so it can fulfill the order.

“Stock consists of the necessary items that a partner should have available to fulfill the order associated with it”.

4.4.2 Return Occurrences

In this section, concepts about Return Occurrences bounded context are defined.

Return

Similar to an Order, a Return (section 2.1.3) is currently known by everyone in the company, also being used in many bounded contexts with different meanings.

“Return is when the customer sends a product back to the seller, being processed by the corresponding partner. The Return is used to identify the origin of a return occurrence.”

Return Occurrence

For this specific business context, it is defined that Return Occurrence is the following this concept:

"Return Occurrence is an anomaly that occurs during return processing, and which is reported by a partner."

Partner

This concept has an identical meaning as the Partner in Order Occurrences context.

"Partner is the merchant responsible for return processing, which has the possibility to report occurrences about it."

Cause

This concept has an identical meaning as the Cause in Order Occurrences context.

"Cause describes the reason for a return processing anomaly, which justifies the generation of a return occurrence."

Form Template

This concept has an identical meaning as the Form Template in Order Occurrences context.

"The Form Template is a template that is required to be filled about a cause, during the creation of a return occurrence."

Form Template Field

This concept has an identical meaning as the Form Template Field in Order Occurrences context.

"The Form Template Field is a field that must be present in some form template. This field might be available to gather information about a return occurrence, such as text, dates, numbers, files or other types of information."

Filled Form

This concept has an identical meaning as the Filled Form in Order Occurrences context.

"A Filled Form is when a form related to a cause is filled during the return occurrence creation."

Form Field

This concept has an identical meaning as the Form Field in Order Occurrences context.

"A Form Field is a field that is filled during the return occurrence creation, which gathers information related to some attribute related to the cause, in the shape of text, dates, numbers, files or other types of information."

Requirement

This concept has an identical meaning as the Requirement in Order Occurrences context.

"Requirement is a rule established for a specific cause, that must be in agreement with the filled form fields, in order to gather valid information about a return occurrence."

Return Occurrence Validation

This concept has an identical meaning as the Order Occurrence Validation in Order Occurrences context.

"Return Occurrence Validation is a validation that must be executed about a defined business rule, related to Return Occurrences."

Ticket

This concept has an identical meaning as the Ticket in Order Occurrences context.

"Ticket is a formal presentation of a return occurrence, that allows the support partner team to visualize it and solve it."

Ticket Field

This concept has an identical meaning as the Ticket Field in Order Occurrences context.

"Ticket Field represents some field that was filled during the return occurrence creation, in order to its value be visualized by the support partner team."

Transport

The Transport is a concept that is normally present in many bounded contexts. It can be used to describe the transport of an Order or a Return. In this context, the transport has the following concept:

"The Transport describes the shipping provider, shipping method and the route used in the transportation of the returned items from the customer to the partner."

4.4.3 Summarizing

Even though two bounded contexts are delineated, many concepts still very identical to each other. But now, these concepts are very specific for the context which are inserted in. For example, a Cause in Order Occurrences context describes the reason for an order processing anomaly that justifies the generation of an Order Occurrence, while in Return Occurrences describes the reason for a return processing anomaly that justifies the generation of a Return Occurrence. It is not generic anymore, it is now related to Orders and Occurrences, and Returns and Occurrences, and this happens with many other concepts.

Furthermore, specific business concepts such as Stock, Priority, and Transport are now inside the corresponding bounded contexts, which allows understanding from what business contexts they belong to.

The definition of a Ubiquitous Language for each bounded context allows clarifying the definition of concepts meaning and its comprehension by the team and product owners.

These concepts are very important to have a clear domain model, and they must change on the corresponding model when they are not according to the business environment. With that, is possible to connect the existent communication of the team and product owners about these concepts with the domain model represented on software.

4.5 Domain Modeling

In this section, Domain Modeling is designed for each bounded context: Order Occurrences and Return Occurrences.

Models will be designed taking into consideration the built Ubiquitous Language of the previous section, using its definitions to represent the business concepts and relationships with a clear understandable domain and to be prepared to evolve for new functionalities and changes required for the business, having behavior inside it. To achieve this, both domain models will be implemented as Rich Domain Models.

Model for Order Occurrences is designed to be implemented in Order Occurrences Service, and the model for Return Occurrences is designed to be implemented in Returns Occurrences Service.

4.5.1 Order Occurrences

In this section, the domain model for Order Occurrences is designed, and its details are explained.

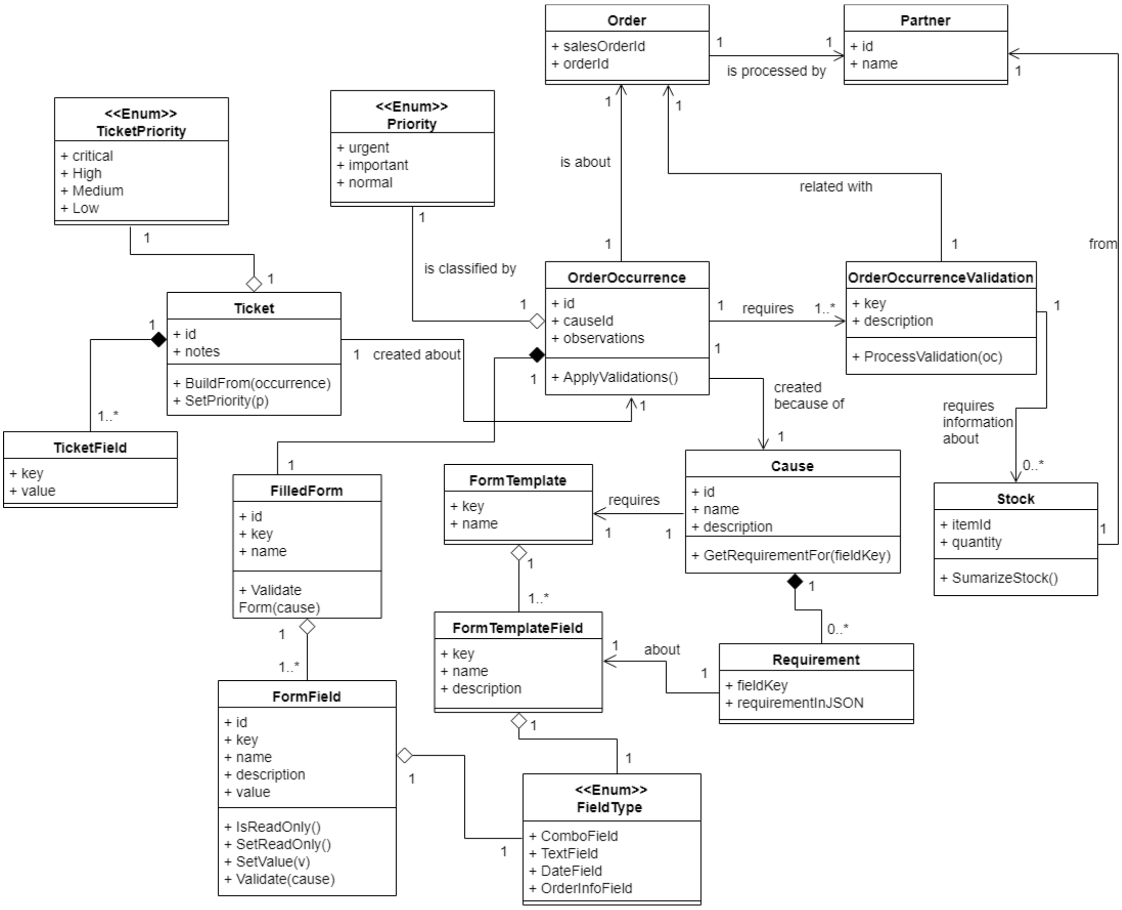


Figure 31 - Domain Model for Order Occurrences

Entities

Following the definition of an Entity provided in Domain Driven Design (Section 2.2.4), the next domain classes are considered as Entities:

Table 11 - Entities in Order Occurrences Domain Model

Entity	Detail
Order Occurrence	An Order Occurrence that is created because of a cause, by a partner, during order processing, and which is always associated with an order.
Cause	Cause that defines why an Order Occurrence was created.
Filled Form	A form that is filled during the creation of an Order Occurrence.
Form Field	A field that gathers information related to the Order Occurrence, such as text, dates, numbers, files or other types of information, complying with the defined requirements of the form.
Ticket	A formal presentation of an Order Occurrence, that allows the support partner team to visualize it and solve it.

Value Objects

Following the definition of a Value Object provided in Domain Driven Design (Section 2.2.4), the next domain classes are considered as Value Objects:

Table 12 - Value Objects in Order Occurrences Domain Model

Value Object	Detail
Order	The merchant order that was separated from its original sales order.
Partner	The merchant responsible for order processing, having the possibility to create occurrences about it.
Order Occurrence Validation	A validation about a defined business rule, related to Order Occurrences.
Priority	The priority that a related Order Occurrence must be embraced.
Form Template	A form that is formed by a set of fields. It is required by a cause and must be filled during the creation of an Order Occurrence.
Form Template Field	A field that is required to be filled for a specific form template.
Requirement	A rule established for a specific cause, that must be in agreement with the filled form, in order to gather valid information about an Order Occurrence.
Field Type	Type of the form field. For fields related to Order Occurrences, only four types are existent: TextField, ComboField, DateField and OrderInfoField.
Ticket Priority	The priority that a related Ticket must be embraced.
Ticket Field	The field that is responsible to have specific information to be presented from a Ticket.
Stock	The necessary items that a partner should have available to fulfill the order associated with it.

Aggregates

Following the definition of an Aggregate provided in Domain Driven Design (Section 2.2.4), the next set of domain classes are considered as Aggregates:

- Cause
- Filled Form
- Ticket

These aggregates demarcated by boundaries, with their associated objects, are possible to be visualized in the next figure:

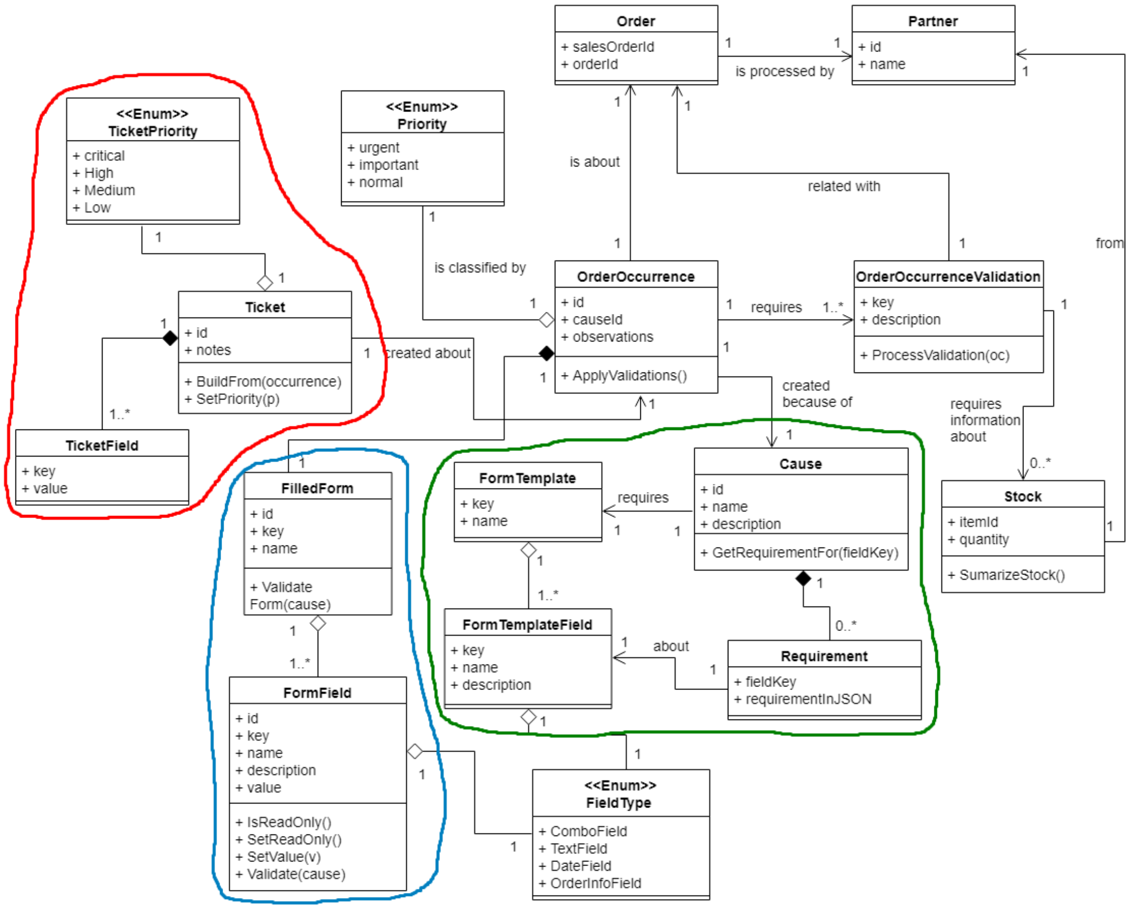


Figure 32 - Aggregates on Order Occurrences Domain Model

Each aggregate is delineated by a boundary, having its own root (Entity), and child elements cannot be accessed from outside its boundary. Operations are only executed in root elements, which, therefore, are responsible to manage these operations with the corresponding child elements.

With this approach, it is possible to group micro-business contexts, that are managed by a root element, ensuring data integrity, since changes cannot be applied directly on inner elements of the root by elements of outside. Operations, or any kind of changes, must be requested to the root element of the aggregate, and with that, it is possible to guarantee that the information of the aggregate is correct.

Events

In the Order Occurrences service, two domain events and one integration event will be published.

A domain event, known as Order Occurrence Created, will be published by the Order Occurrence entity whenever its validations are verified, and it will be subscribed and consumed by its handler, that posteriorly, creates the Ticket entity, which is an aggregate root and that is responsible to build its aggregate.

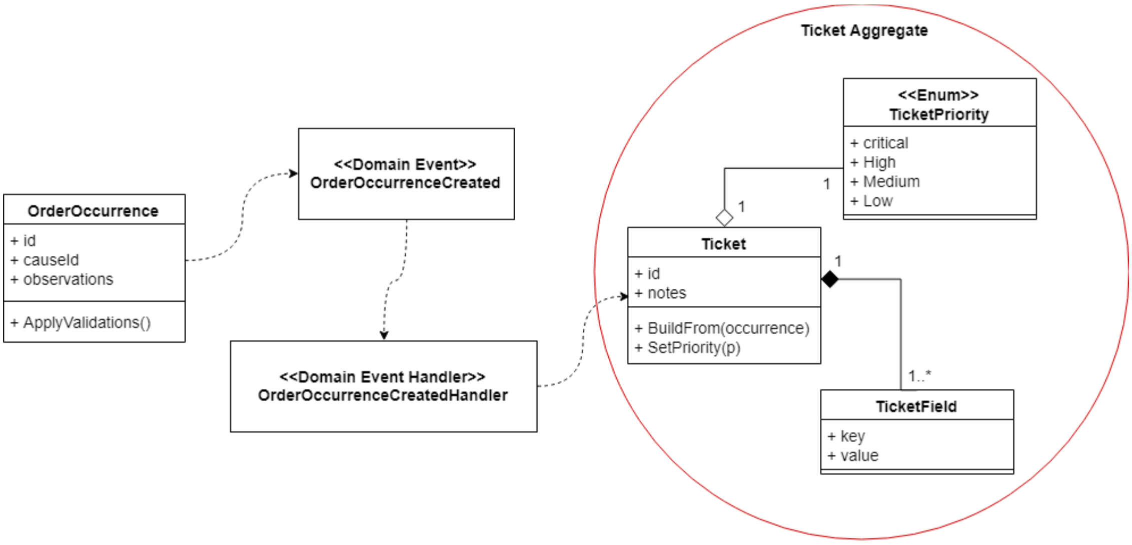


Figure 33 - Order Occurrence Created Domain Event

Therefore, when a Ticket aggregate is successfully built up, it publishes a domain event, known as Ticket Created, which, therefore, will be consumed by a handler that will publish it as an integration event. With this approach, the Ticket is responsible to publish a domain event, besides no domain objects are interested in it, but its information might be useful for external services, and because of that, its domain event is used to publish an integration event that will be available to be consumed outside the Order Occurrences Service. The integration event will be published after Ticket information is persisted in the database, contrary to a domain event.

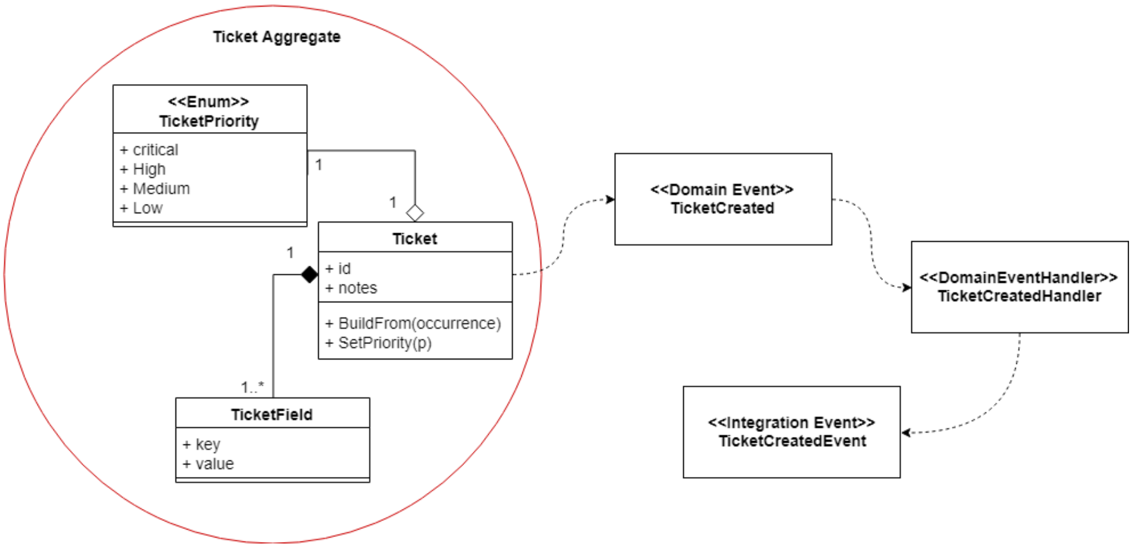


Figure 34 - Ticket Created Integration Event on Order Occurrences

Ticket Aggregate

As previous related, the Ticket concept in this bounded context represents a formal presentation of an Order Occurrence, which allows the support partner team to visualize it and solve it. The Ticket Aggregate has that mission, and that is why it is published in an Integration

Event, so it can be consumed by the interested bounded context, that in this scenario, is the Ticket bounded context.

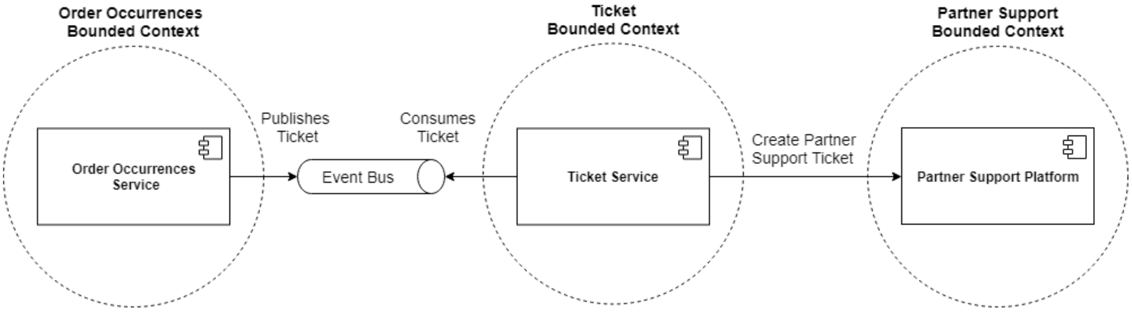

Order Occurrences Bounded Context
Ticket Bounded Context
Partner Support Bounded Context
Order Occurrences Service
Publishes Ticket
Event Bus
Consumes Ticket
Ticket Service
Create Partner Support Ticket
Partner Support Platform

Figure 35 - Ticket across bounded contexts

The Ticket is consumed by a Ticket Service, that uses its information to create a contextualized Ticket to the Partner Support Platform, which belongs to the Partner Support bounded context. With this, it is possible to verify that the Ticket is published in the platform by Order Occurrences bounded context, in the form of integration event, and that is spread over different bounded contexts until it achieves its mission.

In the Order Occurrences context, the Ticket only represents the Order Occurrence formally. Then, when its information is exposed to the platform, its data is used by the different bounded contexts, being manipulated and transformed to the Ticket concept of each context, having a specific meaning for the bounded context that it is inserted to.

4.5.2 Return Occurrences

In this section, the domain model for Return Occurrences is designed, and its details are explained.

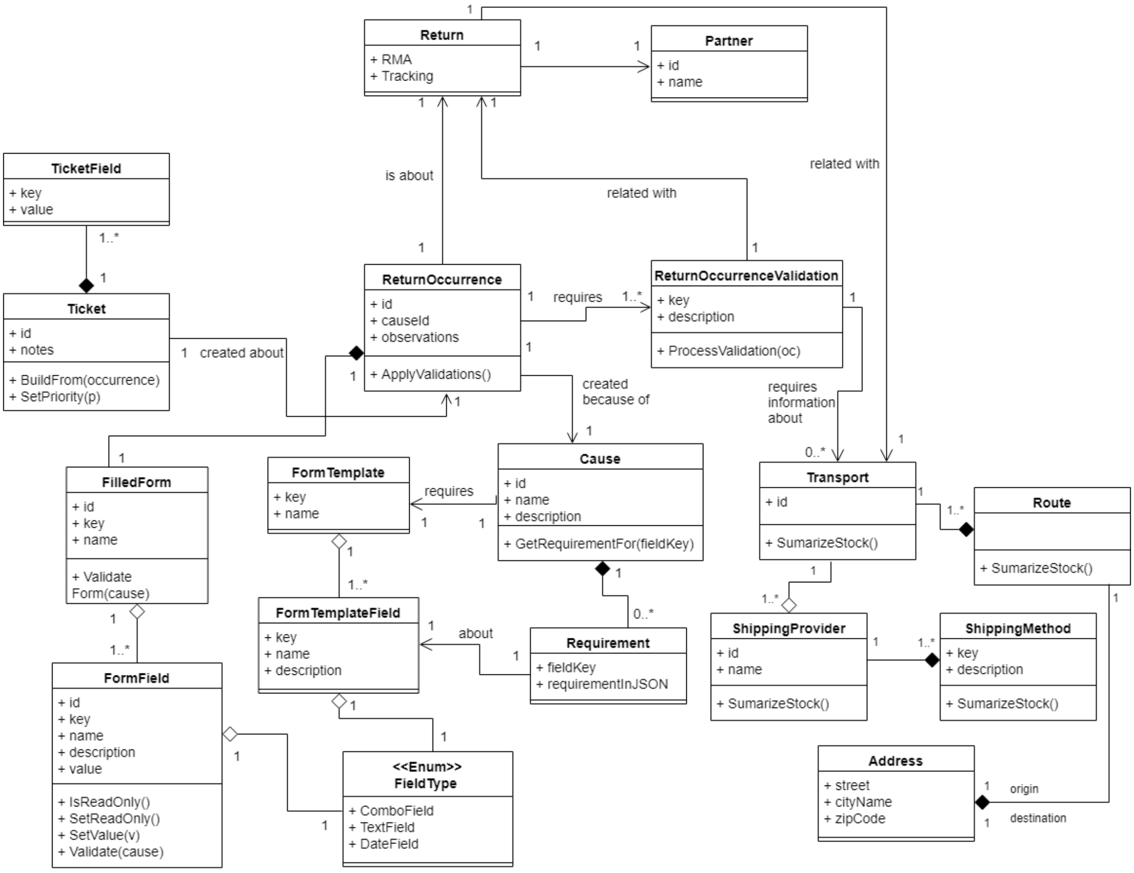


Figure 36 - Domain Model for Return Occurrences

Entities

Following the definition of an Entity provided in Domain Driven Design (Section 2.2.4), the next domain classes are considered as Entities:

Table 13 - Entities in Return Occurrences Domain Model

Entity	Detail
Return Occurrence	An occurrence that is created because of a cause, by a partner, during return processing, which is always associated with a return.
Cause	Cause that defines why a Return Occurrence was created.
Filled Form	A form that is filled during the creation of a Return Occurrence.
Form Field	A field that gathers information related to the Return Occurrence, such as text, dates, numbers, files or other types of information, complying with the defined requirements of the form.
Ticket	A formal presentation of an occurrence, that allows the support partner team to visualize it and solve it.

Transport	Transport used to return the items from the customer to the partner.

Value Objects

Following the definition of a Value Object provided in Domain Driven Design (Section 2.2.4), the next domain classes are considered as Value Objects:

Table 14 - Value Objects in Return Occurrences Domain Model

Value Object	Detail
Return	Return created by a customer, which sends a product back to the seller, being processed by the corresponding partner.
Partner	The merchant responsible return processing, having the possibility to create occurrences about it.
Requirement	A rule established for a specific cause, that must be in agreement with the filled form, in order to gather valid information about a Return Occurrence.
Return Occurrence Validation	A validation about a defined business rule, related to Return Occurrences.
Form Template	A form that is formed by a set of fields. It is required by a cause and must be filled during the creation of a Return Occurrence.
Form Template Field	A field that is required to be filled for a specific form template.
Field Type	Type of the form field. For fields related to Return Occurrences, only three types are existent: TextField, ComboField, and DateField.
Ticket Field	The field that is responsible to have specific information to be presented from a Ticket.
Shipping Provider	Shipping provider that is used to transport the returned items.
Shipping Method	Method of shipping that a provider is capable to execute.
Route	The origin address and the destination address of a Transport.
Address	An address that identifies some location worldwide.

Aggregates

Following the definition of an Aggregate provided in Domain Driven Design (Section 2.2.4), the next set of domain classes are considered as Aggregates:

- Cause
- Filled Form
- Ticket

- Transport

These aggregates demarcated by boundaries, with their associated objects, are possible to be visualized in the next figure:

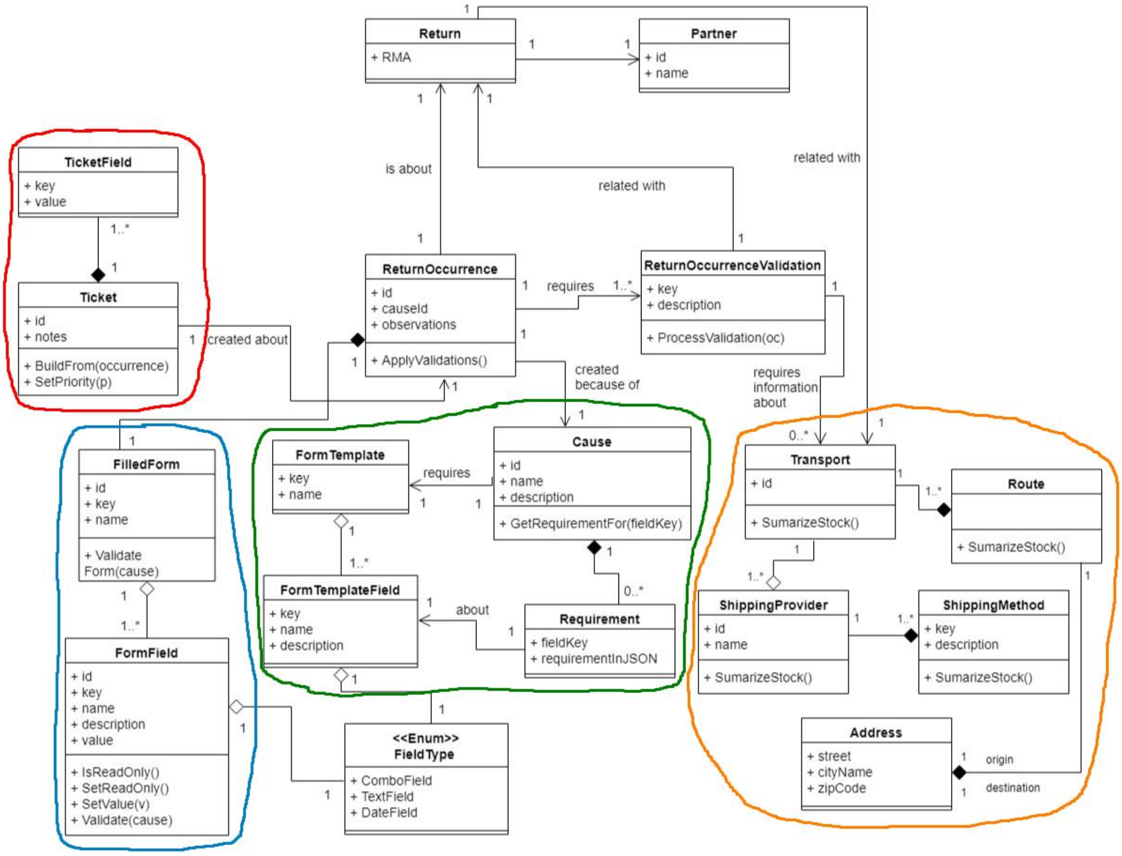


Figure 37 - Aggregates on Return Occurrences Domain Model

Each aggregate is delineated by a boundary, having its own root (Entity), and child elements cannot be accessed from outside its boundary. Operations are only executed in root elements, that therefore are responsible to manage these operations with the corresponding child elements.

With this approach, it is possible to group micro-business contexts, that are managed by a root element, ensuring data integrity, since changes cannot be applied directly on inner elements of the root by elements of outside. Operations, or any kind of changes, must be requested to the root element of the aggregate, and with that, it is possible to guarantee that the information of the aggregate is correct.

Events

In the Return Occurrences service, two domain events and one integration event will be published.

A domain event, known as Return Occurrence Created, will be published by the Return Occurrence entity whenever its validations are verified, and it will be subscribed and consumed by its handler, that posteriorly, creates the Ticket entity, which is an aggregate root, and that will be responsible to build its aggregate.

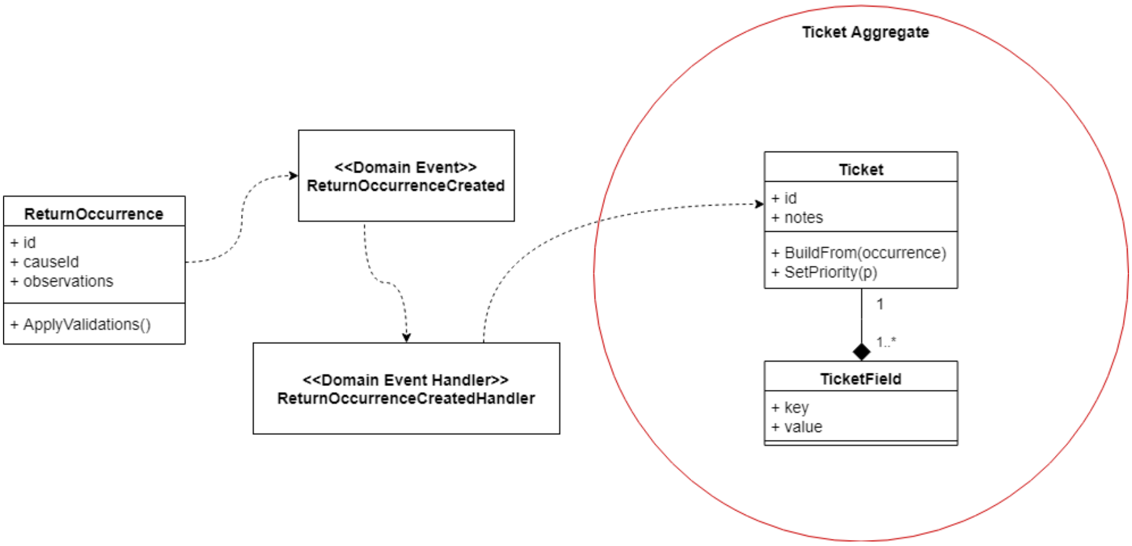


Figure 38 - Return Occurrence Created Domain Event

Therefore, when a Ticket aggregate is successfully built up, it publishes a domain event, known as Ticket Created, which, therefore, will be consumed by a handler, that will publish it as an integration event. This approach will be the same as in Order Occurrence Service, where the Ticket is responsible to publish a domain event, besides no domain objects are interested in it, but its information might be useful for external services, and because of that, its domain event is used to publish an integration event that will be available to be consumed outside the Return Occurrences Service. The integration event will be published after Ticket information is persisted in the database, contrary to the domain event.

Ticket Aggregate

The Ticket Aggregate of Return Occurrences bounded context has a similar mission as Order Occurrences since it is used to formally present a Return Occurrence. When this Ticket is created, it also follows the same process as a Ticket created by Order Occurrences, being consumed by the Ticked bounded context, which uses its information to create a contextualized Ticket to the Partner Support Platform.

4.5.3 Summarizing

Domain models for Order Occurrences and Return Occurrences were designed thinking about Domain Driven Design, implementing a Rich Domain Modeling, where entities and relationships represent the business and the data that is carried in the software, but also having behavior.

Each domain model is clear and specific for its business context, with behavior present directly on the model, having implemented methods with business logic responsibilities.

With this strategy, is not necessary to have an application service layer controlling all the business logic, since that responsibility is assigned to the domain layer with its own services and objects.

Furthermore, both models are identical, since concepts of both Order Occurrences and Return Occurrences bounded contexts are similar to each other. But as referred before, business contexts are evolving in different paths, and now concepts are completely specific and focused in their bounded context, where each one is defined in its corresponding Ubiquitous Language.

5 Implementation

In this chapter, the implementation of this project is presented, taking into consideration the Design from the previous chapter, and creating a technical solution that aims to solve the project constraints. In many scenarios of this chapter, decisions are justified, in order to understand why they were made. During this chapter, common technical implementations are explained for both Order Occurrences and Return Occurrences contexts, as also the specific ones for each service.

5.1 Frameworks and Patterns

In this section, it is presented the programming language used on each service implementation, as also related Patterns and frameworks. The goal of this section is only to provide some implementation context.

5.1.1 ASP.NET Core

C# is used as a programming language for the microservices implementation. As a framework, ASP.NET Core (Microsoft, 2019b) is used, version 2.1. ASP.NET Core is a cross-platform, high-performance, open-source framework for building modern, cloud-based, internet-connect applications. With this framework (Microsoft, 2019b) is possible to:

- Build web applications and services;
- Develop on Windows, macOS, and Linux;
- Deploy to the cloud.

The usage of C# and ASP.NET Core is a company requirement since its platform is based on Microsoft. Its usage does not create any constraint in the implementation of this project.

5.1.2 Mediator

A pattern that is currently used during implementation is the Mediator. This pattern is used to handle domain and integration events, as also communications between Presentation and Domain layers.

The Mediator is a software design pattern that decouples direct communication between objects, defining an object that encapsulates how a set of objects interacts. The Mediator promotes loose coupling by keeping objects from referring to each other explicitly (Gamma *et al.*, 1994).

The Mediator pattern suggests that all direct communication between components, which are intended to make dependent on each other, must be ceased. Instead, these components must collaborate indirectly, by calling a special mediator object, that is responsible for redirecting the calls to appropriate components. As a result, the components depend only on a single mediator class, instead of being coupled to other ones (Shvets, 2018).

The framework used to implement this pattern is MediatR (Bogard, 2019). It is a simple mediator implementation for in-process messaging without any dependencies. It supports request/response, commands, queries, notifications, and events, synchronous and async with intelligent dispatch via C# generic variance. It was chosen as a mediator framework since it is open source.

5.1.3 Dependency Injection

Another pattern that is implemented is the Dependency Injection (DI). It is a software design pattern that implements an Inversion of Control (IoC). This means that objects do not have the need to create other objects on which they rely to do their work. Instead, they get those objects from an outside source, for example, a configuration file (injector) that is responsible to resolve these dependencies (Microsoft, 2013).

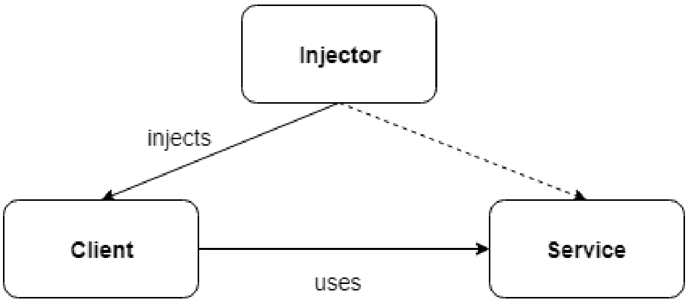


Figure 39 - Dependency Injection

At a high level, the goal of the Dependency Injection pattern is that a client class needs something that satisfies an interface. It does not have the need to know its concrete type (Microsoft, 2013).

One of the benefits provided by DI is low coupling for the classes. With Dependency Injection, it is possible that classes communicate with objects (other classes for example) using interfaces, without having the need to instantiate them. These objects are instantiated by a configuration, somewhere in the application.

With ASP.NET Core, it is possible to assign concrete implementations for the corresponding interfaces, assigning them in a service container at the application start. With that, it is possible to change the concrete implementation of those interfaces without having the need to change its consumers.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IServiceInterface, ServiceA>();
}
```

Code 2 - Dependency Injection Configuration

Interfaces are injected on the constructor of the class that it is used. The framework takes on the responsibility of creating an instance of dependency, and to dispose it when it is no longer needed.

```
public class ServiceB
{
    private readonly IServiceInterface service;

    public ServiceB(IServiceInterface service)
    {
        this.service = service;
    }
}
```

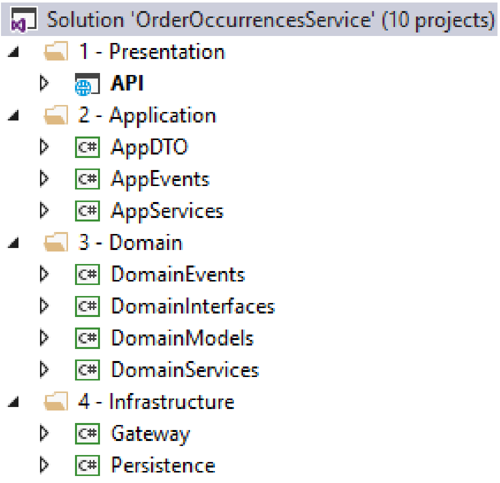
Code 3 - Dependency Injection Usage

5.2 Software Architecture Implementation

In this section, the software architecture designed in section 4.3, is implemented and detailed for each service. For each one, it is presented the project structure and code samples that have the goal to represent the architectural solution implementation.

5.2.1 Order Occurrences

For Order Occurrences service, a Layered Architecture from Domain Driven Design patterns is implemented, as it was represented in figure 29, with four layers: Presentation, Application, Domain, and Infrastructure. Each layer has a responsibility, and it can be divided on multiple projects. For that, a folder was defined for each layer, and each folder may have more than one project associated with it.



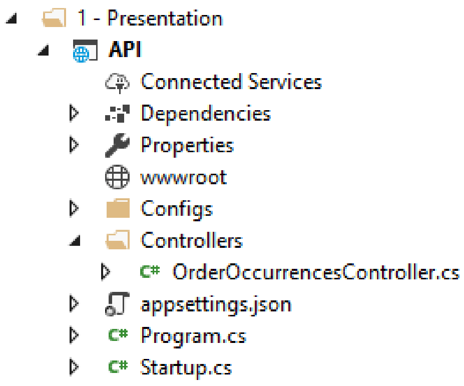
```
Solution 'OrderOccurrencesService' (10 projects)
  1 - Presentation
    API
  2 - Application
    AppDTO
    AppEvents
    AppServices
  3 - Domain
    DomainEvents
    DomainInterfaces
    DomainModels
    DomainServices
  4 - Infrastructure
    Gateway
    Persistence
```

Figure 40 - Architecture in Order Occurrences

When a layer has more than one project, means that its main responsibility is divided on multiple “micro” responsibilities, in order to have a more modular project and a better understanding of the solution structure. The goal is to have the minimum projects, but always taking into consideration the Single Responsibility Principle from SOLID.

5.2.1.1 Presentation

The presentation layer has the responsibility of displaying a REST interface. For this, only one project is necessary. It contains the application Controllers, and the configurations that must be loaded on the application start.



```
1 - Presentation
  API
    Connected Services
    Dependencies
    Properties
    wwwroot
    Configs
    Controllers
      OrderOccurrencesController.cs
    appsettings.json
    Program.cs
    Startup.cs
```

Figure 41 - Presentation Layer in Order Occurrences

A Controller is responsible for controlling the way that an external client interacts with the application.

In this scenario, it is possible to verify a code sample from a Controller, related to getting an Order Occurrence by its identification.

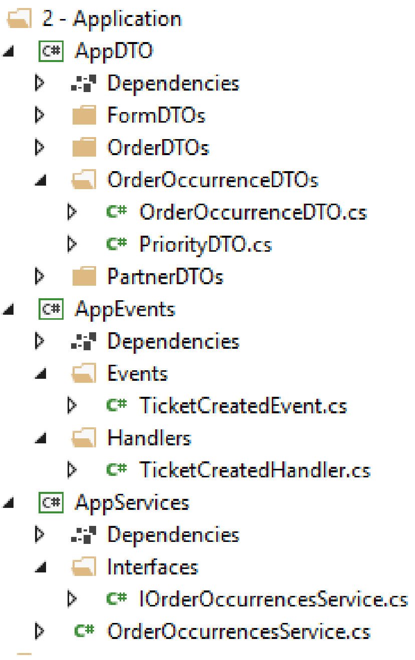
```
  [Route("api/[controller]")]
  [ApiController]
  public class OrderOccurrencesController : ControllerBase
  {
     private readonly IOrderOccurrencesService orderOccurrencesService;

     public OrderOccurrencesController(IOrderOccurrencesService
orderOccurrencesService)
     {
        this.orderOccurrencesService = orderOccurrencesService;

        [HttpGet("{id}")]
        public async Task<IActionResult> GetOrderOccurrence(int id)
        {
           var orderOccurrence =
                await this.orderOccurrencesService.GetOrderOccurrence(id);

            return Ok(orderOccurrence);
        }
     }
  }
```

Code 4 - Order Occurrences Controller

The previous code sample uses Dependency Injection to connect this layer to the layer above (Application). When a request is made for the Controller, it orchestrates the call for the corresponding application service.

5.2.1.2 Application

The Application layer has the responsibility of coordinating the application activity. It does not contain business logic, neither hold the state of business objects, but it can hold the state of an application task progress.



```
2 - Application
◢ AppDTO
    ▷ Dependencies
    ▷ FormDTOs
    ▷ OrderDTOs
    ◢ OrderOccurrenceDTOs
        ▷ OrderOccurrenceDTO.cs
        ▷ PriorityDTO.cs
    ▷ PartnerDTOs
◢ AppEvents
    ▷ Dependencies
    ◢ Events
        ▷ TicketCreatedEvent.cs
    ◢ Handlers
        ▷ TicketCreatedHandler.cs
◢ AppServices
    ▷ Dependencies
    ◢ Interfaces
        ▷ IOrderOccurrencesService.cs
    ▷ OrderOccurrencesService.cs
```

Figure 42 - Application Layer in Order Occurrences

As shown in the previous figure, this layer is divided into three projects: AppDTO, AppServices, and AppEvents.

Table 15 - Layers in Order Occurrences

Name	Description
AppDTO	This project has only one responsibility, which is holding the DTO's (Data Transfer Objects) references, that are used in the communication between Presentation and Application layers.
AppServices	This project has the responsibility of holding the references for application services interfaces and implementations. In this project, it connects with Domain Layer, and it is responsible for mapping the domain objects for its DTO's, and vice-versa.
AppEvents	This project has the responsibility of handle domain events, orchestrating and transforming its data in order to trigger integration events to outside.

Next, it is possible to see a code sample for an Application Service implementation, related to Order Occurrences.

```
    public class OrderOccurrencesService : IOrderOccurrencesService
    {
        private readonly IOrderOccurrencesRepository orderOccurrencesRepository;
        private readonly IMapper mapper;

        public OrderOccurrencesService(IOrderOccurrencesRepository
orderOccurrencesRepository, IMapper mapper)
        {
            this.orderOccurrencesRepository = orderOccurrencesRepository;
            this.mapper = mapper;
        }

        public async Task<OrderOccurrenceDTO> GetOrderOccurrence(int id)
        {
            var orderOccurrenceDomain
                = await this.orderOccurrencesRepository.GetOrderOccurrence(id);

            var orderOccurrenceDTO =
                  this.mapper.Map<OrderOccurrenceDTO>(orderOccurrenceDomain);

            return orderOccurrenceDTO;
        }
    }
```

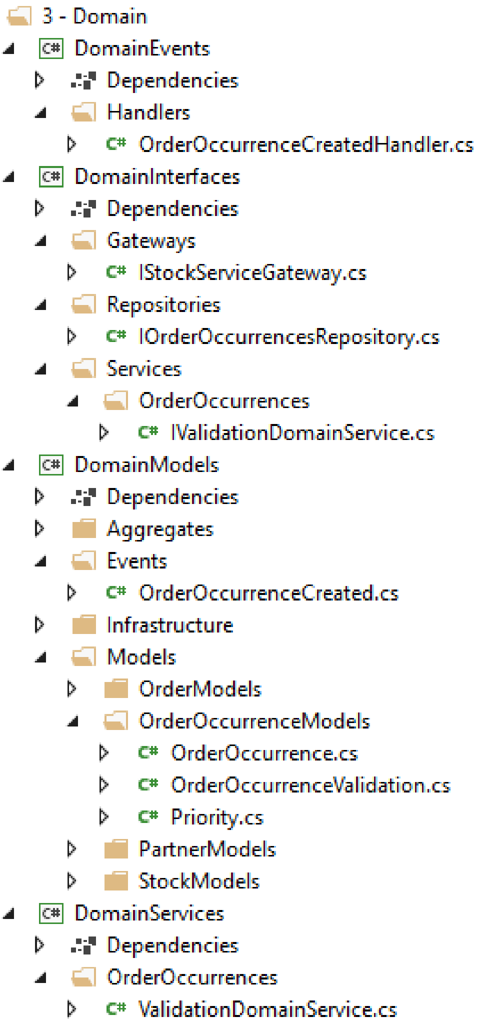
Code 5 - Order Occurrences Application Service

The previous code sample uses Dependency Injection to connect this layer to the layer above (Domain), and to connect with a Mapper interface, responsible for mapping DTO’s to domain objects, and vice-versa. When a request is made for this service, it orchestrates the call for the corresponding Domain interface (in this case, a repository), maps the domain object response to a DTO and returns it. This service could also orchestrate Domain services, Domain events or tasks nonrelated with the domain.

5.2.1.3 Domain

The Domain layer is responsible for holding information about the domain, and it is considered the heart of the business software. The state of business objects is held here, and the persistence of these objects, and possibly their state, is delegated to the Infrastructure layer.

In this section, it is only presented the domain layer structure, in order to understand what the goals of each existent domain project are. Details about how the domain model is structured, and about its entities, value objects and more, are presented in section 5.4.



```
3 - Domain
◢ DomainEvents
    ▷ Dependencies
    ◢ Handlers
        ▷ OrderOccurrenceCreatedHandler.cs
◢ DomainInterfaces
    ▷ Dependencies
    ◢ Gateways
        ▷ IStockServiceGateway.cs
    ◢ Repositories
        ▷ IOrderOccurrencesRepository.cs
    ◢ Services
        ◢ OrderOccurrences
            ▷ IValidationDomainService.cs
◢ DomainModels
    ▷ Dependencies
    ▷ Aggregates
    ◢ Events
        ▷ OrderOccurrenceCreated.cs
    ▷ Infrastructure
    ◢ Models
        ▷ OrderModels
        ◢ OrderOccurrenceModels
            ▷ OrderOccurrence.cs
            ▷ OrderOccurrenceValidation.cs
            ▷ Priority.cs
        ▷ PartnerModels
        ▷ StockModels
◢ DomainServices
    ▷ Dependencies
    ◢ OrderOccurrences
        ▷ ValidationDomainService.cs
```

Figure 43 - Domain Layer in Order Occurrences

As shown in the previous figure, this layer is divided into four projects: DomainEvents, DomainInterfaces, DomainModels, DomainServices.

Table 16 - Domain projects in Order Occurrences

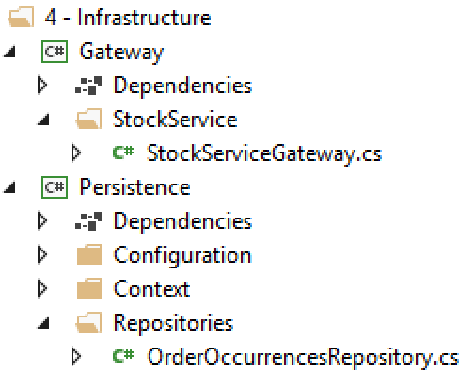
Name	Description
DomainEvents	This project is responsible for handling the domain events that are triggered by domain objects. When handling a domain event, it can apply some business logic, or communicate with other domain objects, such as aggregates.
DomainInterfaces	This project is responsible for holding domain interfaces, such as services, repositories or gateway interfaces.
DomainModels	This project is responsible to represent the domain model, with its entities, value objects, aggregates and domain events. Business concepts, relationships, and behaviors are mainly represented in this project objects.

DomainServices	This project is responsible for holding the services related to the domain. These services mainly orchestrate and apply business rules across different entities or aggregates, which cannot be applied by individual ones.

Domain layer, with its inner projects, holds everything related to business, and it is considered the most crucial layer, which this solution must really focus on, in order to solve the constraints related to the problem of this project. With that, a further explanation about this layer is presented in section 5.4, so it is possible to analyze the details about this layer implementation.

5.2.1.4 Infrastructure

The infrastructure layer is responsible for supporting all the other layers. It implements persistence for business objects and communicates with external services.



```
4 - Infrastructure
◢ Gateway
    ▷ Dependencies
    ◢ StockService
        ▷ StockServiceGateway.cs
◢ Persistence
    ▷ Dependencies
    ▷ Configuration
    ▷ Context
    ◢ Repositories
        ▷ OrderOccurrencesRepository.cs
```

Figure 44 - Infrastructure Layer in Order Occurrences

As shown in the previous figure, this layer is divided into two projects: Gateway and Persistence.

Table 17 – Infrastructure projects in Order Occurrences

Name	Description
Gateway	This project has the responsibility of holding the references for the external services, which implements the domain interfaces related to gateways. It communicates with external services and it adapts external information to the domain objects of the application.
Persistence	This project has the responsibility of holding the references for the concrete repositories, which implements the domain interfaces related to repositories. It communicates with the database (abstracting it from the application), filling the domain objects with its persisted information.

Next, it is possible a code sample for a Repository related to Order Occurrences.

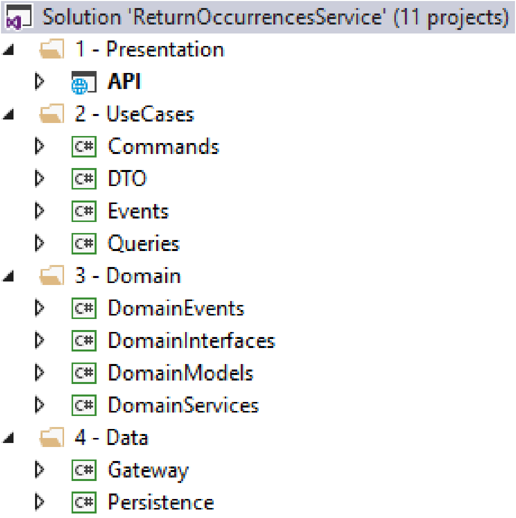
```
public class OrderOccurrencesRepository : IOrderOccurrencesRepository
{
    private readonly IDatabase database;

    public OrderOccurrencesRepository(IDatabase database)
    {
        this.database = database;
    }

    public async Task<OrderOccurrence> GetOrderOccurrence(int id)
    {
        // Implementation Body
    }
}
```

Code 6 - Order Occurrences Repository

The previous code sample uses Dependency Injection to connect this layer to the database. When a request is made for this repository, it is responsible to connect with the database and to manipulate/query its data. When it gathers information, it always responses with domain objects.

5.2.2 Return Occurrences

For Return Occurrences service, a Clean Architecture with CQRS is implemented, as it was represented in figure 30, with four layers represented: Presentation, Use Cases, Domain and Data. This approach is very similar to Occurrences in Services service, where each layer has a responsibility, and it can be divided on multiple projects. With that, a folder was defined for each layer, and each folder may have more than one project associated with it.



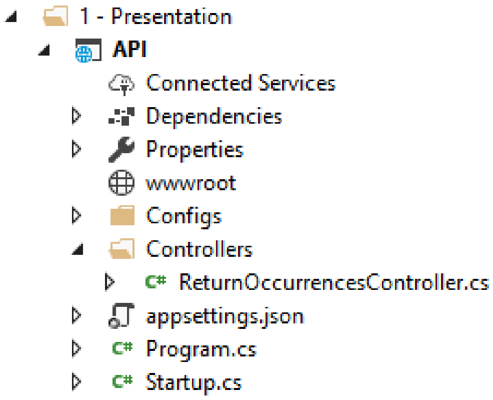
```
Solution 'ReturnOccurrencesService' (11 projects)
◢ 1 - Presentation
    ▷ API
◢ 2 - UseCases
    ▷ Commands
    ▷ DTO
    ▷ Events
    ▷ Queries
◢ 3 - Domain
    ▷ DomainEvents
    ▷ DomainInterfaces
    ▷ DomainModels
    ▷ DomainServices
◢ 4 - Data
    ▷ Gateway
    ▷ Persistence
```

Figure 45 - Architecture in Return Occurrences

As referred before, when a layer has more than one project, means that its main responsibility is divided on multiple “micro” responsibilities.

5.2.2.1 Presentation

The presentation layer has the responsibility of displaying a REST interface. For this, only one project is necessary. This contains the application Controllers, and the application configurations that must be loaded on the application start.



```
◢ 1 - Presentation
  ◢ API
      Connected Services
    ▷ Dependencies
    ▷ Properties
      wwwroot
    ▷ Configs
    ◢ Controllers
      ▷ ReturnOccurrencesController.cs
    ▷ appsettings.json
    ▷ Program.cs
    ▷ Startup.cs
```

Figure 46 - Presentation Layer in Return Occurrences

Related with Presentation structure, there are no differences architectural differences between this project and the one that is implemented in Order Occurrences service.

In this scenario, it is possible to verify a code sample from a Controller, related to the getting of an Occurrence in Return by its identification.

```
[Route("api/[controller]")]
[ApiController]
public class ReturnOccurrencesController : ControllerBase
{
    [HttpGet("{id}")]
    public async Task<IActionResult> GetReturnOccurrence(int id)
    {
        var returnOccurrence =
            await Mediator.Send(new GetReturnOccurrenceQuery(id));

        return Ok(returnOccurrence);
    }
}
```

Code 7 - Return Occurrences Controller

Contrary to Order Occurrences service, where the controllers have the dependencies injected in its constructors, this project uses a Mediator Pattern (section 5.1.2) to communicate from the Presentation layer to the layer above (Domain Layer). An example of this communication flow, for a query, can be represented in the next figure:

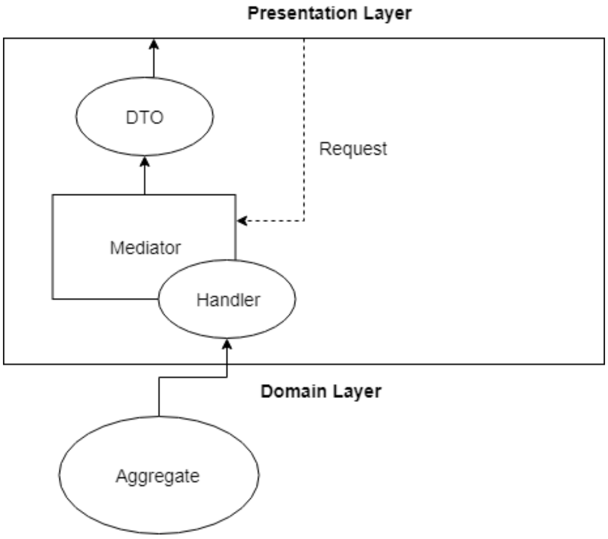


Figure 47 - Mediator usage in Return Occurrences

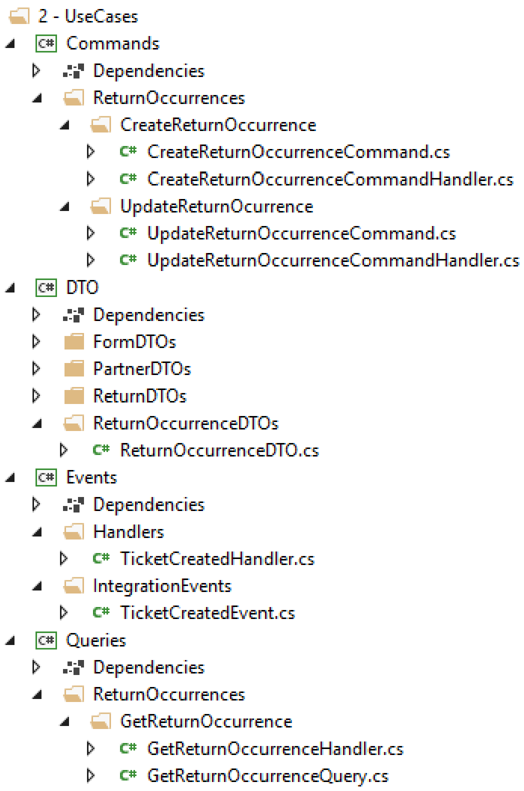
With a Mediator, it is possible to handle requests for DTO’s by the Presentation Layer. The Aggregate sends its own response through the Mediator. In this scenario, a Handler will be registered with the Mediator class.

In this architecture, the handlers are the Use Cases. When the Mediator gets a Query that can be handled, then its responsible Handler is called and sends the DTO back to the Presentation Layer. For this service, The Mediator Pattern is supported by the framework MediatR.

5.2.2.2 Use Cases

This layer can be compared to the Application Layer from the Order Occurrences service. Even though both are implemented completely distinct from each other, they have the same responsibilities. This one contains the application-specific Use Cases. These orchestrate the flow of data to and from the entities and direct those entities to use their business rules to achieve the goals of the Use Case.

Similar to an Application Service, it does not contain business logic, neither hold the state of business objects, but it can hold the state of an application task progress.



```
2 - UseCases
  Commands
    Dependencies
    ReturnOccurrences
      CreateReturnOccurrence
        CreateReturnOccurrenceCommand.cs
        CreateReturnOccurrenceCommandHandler.cs
      UpdateReturnOcurrence
        UpdateReturnOccurrenceCommand.cs
        UpdateReturnOccurrenceCommandHandler.cs
  DTO
    Dependencies
    FormDTOs
    PartnerDTOs
    ReturnDTOs
    ReturnOccurrenceDTOs
      ReturnOccurrenceDTO.cs
  Events
    Dependencies
    Handlers
      TicketCreatedHandler.cs
    IntegrationEvents
      TicketCreatedEvent.cs
  Queries
    Dependencies
    ReturnOccurrences
      GetReturnOccurrence
        GetReturnOccurrenceHandler.cs
        GetReturnOccurrenceQuery.cs
```

Figure 48 - Use Cases Layer in Return Occurrences

As shown in the previous figure, this layer is divided into four projects: Commands, Queries, DTO, and Events.

Table 18 - Use Cases projects in Return Occurrences

Name	Description
Commands	This project contains the commands that are executed, and its handlers, in order to trigger transactions, data updates or any changed state in the system. These handlers are considered as Use Cases.
Queries	This project contains the queries that are executed, and its handlers, in order to query data from the system. These handlers are also considered as Use Cases.
DTO	This project has only one responsibility, which is holding the DTO's references, that are used in the communication between Presentation and Use Cases layers.
Events	This project has the responsibility of handle domain events, orchestrating and transforming its data in order to trigger integration events to outside.

Next, it is possible to see the code sample for a Query Handler (Use Case for querying information from the system) related to Return Occurrences.

```
public class GetReturnOccurrenceHandler :
MediatR.IRequestHandler<GetReturnOccurrenceQuery, ReturnOccurrenceDTO>
    {
        private readonly IReturnOccurrencesRepository
returnOccurrencesRepository;
        private readonly IMapper mapper;

        public GetReturnOccurrenceUseCase(IReturnOccurrencesRepository
returnOccurrencesRepository, IMapper mapper)
        {
            this.returnOccurrencesRepository =
returnOccurrencesRepository;
            this.mapper = mapper;
        }

        public async Task<ReturnOccurrenceDTO>
Handle(GetReturnOccurrenceQuery request, CancellationToken
cancellationToken)
        {
            var returnOccurrenceDomain
                 = await
returnOccurrencesRepository.GetReturnOccurrence(request.Id);

            var returnOccurrenceDTO =
mapper.Map<ReturnOccurrenceDTO>(returnOccurrenceDomain);

            return returnOccurrenceDTO;
        }
    }
```

Code 8 - Get Return Occurrence Use Case Handler

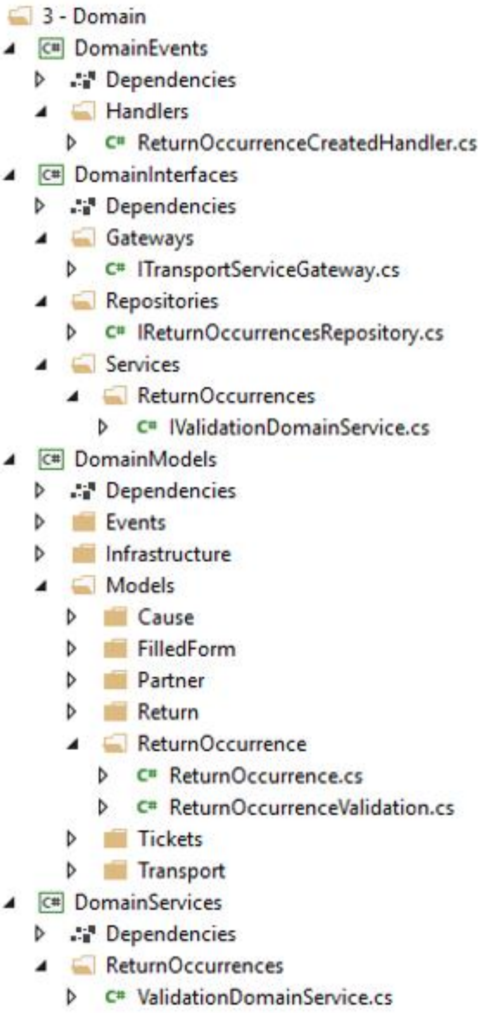
Similar to Application Services from Order Occurrences, the previous code sample uses Dependency Injection to connect this layer to the layer above (Domain), and to connect with a Mapper interface, responsible for mapping DTO's to domain objects, and vice-versa. When a Use Case is requested, it orchestrates the call for the corresponding Domain interface (in this case, a repository), maps the domain object response to a DTO and returns it. A Use Case can also orchestrate Domain services, Domain events or tasks nonrelated with the domain.

The previous Use Case uses the MediatR framework to know when it is requested and to define the Query that it handles, and the DTO that it answers.

5.2.2.3 Domain

This Domain layer is implemented in the same way as the Domain layer of Order Occurrences service, except for the folders structure related to domain models. Since this is the most crucial layer of both projects, it is intended to use the same architectural approach, in order to not cause constraints related to domain understanding.

In this section, it is only presented the domain layer structure, in order to understand the goals of each existent domain project.



```
3 - Domain
  DomainEvents
    Dependencies
    Handlers
      ReturnOccurrenceCreatedHandler.cs
  DomainInterfaces
    Dependencies
    Gateways
      ITransportServiceGateway.cs
    Repositories
      IReturnOccurrencesRepository.cs
    Services
      ReturnOccurrences
        IValidationDomainService.cs
  DomainModels
    Dependencies
    Events
    Infrastructure
    Models
      Cause
      FilledForm
      Partner
      Return
      ReturnOccurrence
        ReturnOccurrence.cs
        ReturnOccurrenceValidation.cs
      Tickets
      Transport
  DomainServices
    Dependencies
    ReturnOccurrences
      ValidationDomainService.cs
```

Figure 49 - Domain Layer in Return Occurrences

As shown in the previous figure, this layer is divided into four projects: DomainEvents, DomainInterfaces, DomainModels, DomainServices.

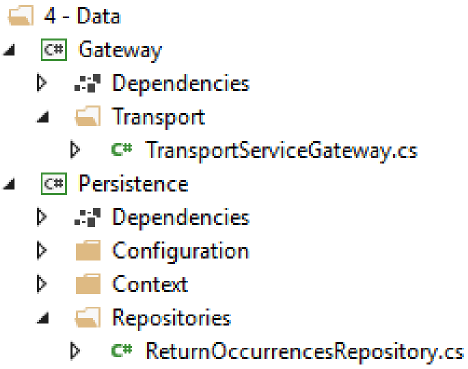
Table 19 - Domain projects in Return Occurrences

Name	Description
DomainEvents	This project is responsible for handling the domain events that are triggered by domain objects. When handling a domain event, it can apply some business logic, or communicate with other domain objects, such as aggregates.
DomainInterfaces	This project is responsible for holding domain interfaces, such as services, repositories or gateway interfaces.
DomainModels	This project is responsible to represent the domain model, with its entities, value objects, aggregates and domain events. Business concepts, relationships, and behaviors are mainly represented in this project objects.

DomainServices	This project is responsible for holding the services related to the domain. These services mainly orchestrate and apply business rules across different entities or aggregates, which cannot be applied by individual ones.

5.2.2.4 Data

This layer is responsible for a set of adapters to convert data from the format most convenient for the use cases and entities, to the format most convenient for some external software such as database or service. Basically, it follows the same approach of the Infrastructure layer from Order Occurrences service.



- 4 - Data
  - Gateway
    - Dependencies
    - Transport
      - TransportServiceGateway.cs
  - Persistence
    - Dependencies
    - Configuration
    - Context
    - Repositories
      - ReturnOccurrencesRepository.cs

Figure 50 - Data Layer in Return Occurrences

As shown in the previous figure, this layer is divided into two projects: Gateway and Persistence.

Table 20 - Data projects in Return Occurrences

Name	Description
Gateway	This project has the responsibility of holding the references for the external services, which implements the domain interfaces related to gateways. It communicates with external services and it adapts external information to the domain objects of the application.
Persistence	This project has the responsibility of holding the references for the concrete repositories, which implements the domain interfaces related to repositories. It communicates with the database (abstracting it from the application), filling the domain objects with its persisted information.

Next, it is possible a code sample for a Repository related to Return Occurrences.

```
public class ReturnOccurrencesRepository : IReturnOccurrencesRepository
   {
       private readonly IDatabase database;

       public ReturnOccurrencesRepository(IDatabase database)
       {
           this.database = database;
       }

       public async Task<ReturnOccurrence> GetReturnOccurrence(int id)
       {
           //Implementation body
       }
   }
```

Code 9 - Return Occurrences Repository

The previous code sample uses Dependency Injection to connect this layer to the database. When a request is made for this repository, it is responsible to connect with the database and to manipulate/query its data. When it gathers information, it always responses with domain objects.

5.3 Domain Modeling Common Infrastructure

Before thinking about the concrete Domain Modeling implementations in both services, it is interesting to create a common infrastructure, that is used for the corresponding domain models.

Since Domain Driven Design is being implemented in both Order Occurrences and Return Occurrences services, it is valuable to create an infrastructure that helps to guarantee that some DDD elements are correctly implemented. For this, five elements were chosen to be supported by this infrastructure: **Entities**, **Value Objects**, **Aggregate Roots** and **Aggregate Elements**, and finally, **Domain Events**.

These elements were never highlighted in previous domain models, which difficulties the domain understanding, and besides that, these ones could be easily broken. With that, a smaller infrastructure was created to be used in both projects, in order to mitigate this kind of constraints. This infrastructure ensures the following rules:

1. An Entity has always an identification, and it is only compared with another Entity by its identification, and not by its own attributes;
2. An Aggregate Root is always an Entity;
3. A Value Object is compared with another Value Object by its own attributes;
4. A Domain Event is notified to the Domain;

Having an infrastructure that guarantees the previous rules, allows a better comprehension of the Domain Modeling since each domain model object is highlighted and it might have restrictions, for example, an object that is considered an Entity must have identification, like an Aggregate Root aswell.

Infrastructure Implementation

To implement this infrastructure, three abstract classes were created: **Entity**, **Value Object**, **Aggregate Root**. Finally, two interfaces were created: **Aggregate Element** and **Domain Event**.

Starting with the Entity class, it contains four essencial methods: Equals(), IsTransient(), and two methods for operators == and !=. These can be visualized in the next code:

```
public abstract class Entity<TId>
{
    public TId Id { get; protected set; }

    public ICollection<IDomainEvent> Events { get; }

    public bool IsTransient()
    {
        return this.Id.Equals(default(TId));
    }

    public override bool Equals(object otherEntity)
    {
        return this.Equals(otherEntity as Entity<TId>);
    }

    public bool Equals(Entity<TId> otherEntity)
    {
        return this.Id.Equals(otherEntity.Id);
    }

    public static bool operator ==(Entity<TId> left, Entity<TId> right)
    {
        return left.Equals(right);
    }

    public static bool operator !=(Entity<TId> left, Entity<TId> right)
    {
        return !(left.Equals(right));
    }
}
```

Code 10 - Domain Entity Infrastructure

In the previous code, it is possible to check that the corresponding abstract class contains operations that ensure the correct comparison between Entities, with its Equals() method and operators. At the same time, using a generic type TId, it guarantees that the Entity contains an identifier. Furthermore, the method IsTransient() allows verifying if the Entity is transient, which means that it allows checking if the Entity is not persisted yet by verifying its identification value.

Finally, it contains a collection of domain events. These events are not raised instantly to the domain, instead, they are added to a collection of events, and further, they are processed by the system. With that, the entity has only the responsibility to create a domain event, and to add it to its events list, knowing that those events will be notified in the future for the domain.

This Entity class can be used with heritage by every entity in the domain model. The next code example shows how the Entity defines the generic type of its identification as an integer.

```
public class ExampleEntity : Entity<int>
{
    public string Name { get; private set; }

    public string Value { get; private set; }
}
```

Code 11 - Domain Entity implementation example

Taking the next step for the Value Object class, similarly as Entity, it also contains three essential methods: Equals() and two methods for operators == and !=, and these can be visualized in the next code:

```
public abstract class ValueObject
    {
        public override bool Equals(object otherObject)
        {
            return Equals(otherObject as ValueObject);
        }

        public bool Equals(ValueObject otherObject)
        {
            //Code to compare fields of both Value Objects
        }

        public static bool operator ==(ValueObject left, 
ValueObject right)
        {
            return left.Equals(right);
        }

        public static bool operator !=(ValueObject left, 
ValueObject right)
        {
            return !(left.Equals(right));
        }
    }
```

Code 12 - Value Object Infrastructure

In the previous code, it is possible to check that the corresponding abstract class contains operations that ensure the correct comparison between Value Objects, with its Equals() method and operators. This comparison is made by comparing each field between the Value Objects, even though it is not demonstrated in the code, because it is too complex, and this code is only intended to demonstrate the class functionalities.

Next, the Aggregate Root class is one of the most simple implementations, since it is associated with an Entity by heritage as the following code demonstrates:

```
public abstract class AggregateRoot<TId> : Entity<TId> { }
```

Code 13 - Aggregate Root Infrastructure

With this, it is assumed that an Aggregate Root is always an Entity and that it also follows the Entity guidelines.

Also associated with aggregates, the Aggregate Element is an empty interface without having anything in there, as the following code demonstrates:

```
public interface IAggregateElement {}
```

Code 14 - Aggregate Element Infrastructure

This interface only exists to signal the domain objects that belong to an Aggregate. These objects might be Entities or Value Objects, and implementing this interface will allow better visibility for the developers to visualize that these objects belong to an Aggregate and that they must have attended in the actions that are exposed on it. Actions to these elements, that are requested by outside of the aggregate, must be handled by the aggregate root and using this interface in those objects signals that developers must be careful in the developments of those requested actions, in order to not break the Aggregate rules from Domain Driven Design.

Finally, the Domain Event is an empty interface that inherits from Notification, which is associated with MediatR framework.

```
public interface IDomainEvent : INotification { }
```

Code 15 - Domain Event Infrastructure

This interface must be implemented by the Domain Event objects that are raised by domain. Those domain objects contain relevant information about the event, that posterior will be consumed by the responsible handlers. These domain events are possible to exist with the help of MediatR framework, which is responsible to act as a mediator between the component that creates an event and the component that consumes it.

5.4 Order Occurrences Model Implementation

Domain Model is considered, by Eric Evans, as the heart of software. For this project, the Domain Model layer is crucial for the success of the project. For Order Occurrences, this model was already designed in section 4.5.1, with its entities, value objects, and aggregates. Software patterns and architectures were also approached in turn of this model, and during this chapter of Implementation, it is possible to check how everything is built. The goal is to achieve the correct usage of Domain Driven Design in this project, to solve the constraints of the project problem.

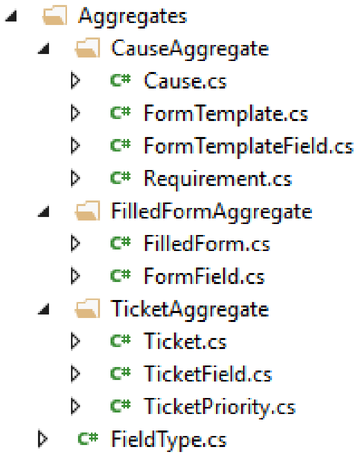
In this section, the main aspects of Domain Model related to Order Occurrences are represented and explained. As previous related, the Domain Model was already, and the goal is not to demonstrate its complete implementation, but to demonstrate a base implementation of that model, that allows to the team developers to comprehend it, and to continuously develop it, for current and new business requirements using a Domain Driven Design strategy.

5.4.1 Domain Model Structure

The project of Domain Models is responsible to represent the domain model, with its entities, value objects, aggregates, relationships, behaviors, and even domain events. For this, a structure was defined with four main folders: Aggregates, Events, Infrastructure, and Models.

Aggregates

The Aggregates folder contains every aggregate (as subfolder) that is existent on the Domain. In this case, the Domain model contains three aggregates: Cause, Filled Form and Ticket. For that was created three folders, with its contained domain objects:



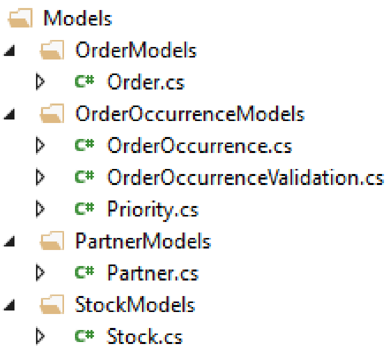
```
Aggregates
  CauseAggregate
    Cause.cs
    FormTemplate.cs
    FormTemplateField.cs
    Requirement.cs
  FilledFormAggregate
    FilledForm.cs
    FormField.cs
  TicketAggregate
    Ticket.cs
    TicketField.cs
    TicketPriority.cs
  FieldType.cs
```

Figure 51 - Domain Aggregates in Order Occurrences

Comparing to the designed model, it is possible to verify that every aggregate contains its inner domain objects. With the splitting of each aggregate in one folder, it is expected that it allows to distinguish them from the rest of the domain model, and that provides a better comprehension of what are the aggregates present in the model.

Models

The Models folder contains every domain model object that is not integrated into an Aggregate. This folder contains many subfolders, where each one belongs to a specific concept, as for example Order and Partner.



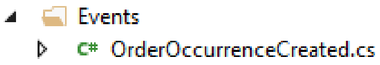
```
Models
  OrderModels
    Order.cs
  OrderOccurrenceModels
    OrderOccurrence.cs
    OrderOccurrenceValidation.cs
    Priority.cs
  PartnerModels
    Partner.cs
  StockModels
    Stock.cs
```

Figure 52 - Domain Models in Order Occurrences

Even though the models of these subfolders do not belong to any Aggregate, they still belong to some business concept. It is expected that with the splitting of each business concept in one subfolder, it allows distinguishing the different areas of the domain model, with its inner domain objects.

Events

The Events folder contains every domain event object that is triggered by domain model objects.



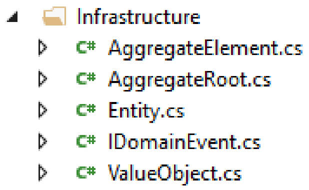
```
Events
  OrderOccurrenceCreated.cs
```

Figure 53 - Domain Events in Order Occurrences

This folder does not contain any subfolder, because, at the moment, there are not many domain events that are triggered by the domain. So, adding subfolders may create some unnecessary noise. Although this does not mean that these subfolders will never exist, because, if the number of domain events increases, adding subfolders would split the domain events into different areas, increasing its comprehension.

Infrastructure

The Infrastructure folder contains every object related to domain infrastructure, as it was presented in the previous section 5.3. For now, there are five objects:



```
Infrastructure
  AggregateElement.cs
  AggregateRoot.cs
  Entity.cs
  IDomainEvent.cs
  ValueObject.cs
```

Figure 54 - Domain Infrastructure in Order Occurrences

These objects are referenced across the Domain Model project, by Aggregate Roots and its elements, Entities, Value Objects, and Domain Events. This Infrastructure folder only contains objects that have the goal to support the other domain model objects, in order to ensure a correct implementation of Domain Driven Design.

5.4.2 Entities Implementation

As previously related, Entities are considered as a category of objects that have an identity that remains the same throughout the states of the software. In this section, an example of an Entity implementation is provided, with the goal to demonstrate the guidelines that are followed in Entities development in this project.

Every domain object that is considered as an Entity must inherit from Entity object from Domain Infrastructure (section 5.3). In this scenario, the entity Order Occurrence from the domain model related to Order Occurrences inherits from the Entity object:

```
public class OrderOccurrence : Entity<int> {}
```

Code 16 - Order Occurrence Entity definition

Since the identification of the Order Occurrence entity is an integer number, then the Order Occurrence inherits from an Entity with identification of integer type.

Related with the Order Occurrence attributes, this entity was designed with six attributes: Id, Order, Cause Id, Observations, Priority and Filled Form. Inheriting from an Entity, already forces this class to have the Id attribute. So, it is only necessary to implement other attributes:

```
public class OrderOccurrence : Entity<int>
{
    public Order Order { get; private set; }
    public int CauseId { get; private set; }
    public string Observations { get; private set; }
    public Priority Priority { get; private set; }
    public FilledForm FilledForm { get; private set; }

    public OrderOccurrence(Order order
        int causeId,
        string observations,
        Priority priority,
        FilledForm filledForm)
    {
        Order = order;
        CauseId = causeId;
        Observations = observations;
        Priority = priority;
        FilledForm = filledForm;
    }
 }
```

Code 17 - Order Occurrences Entity implementation

In the previous code, it demonstrates that the attributes are implemented as public ones, but with private setters. This means, that only the Order Occurrence class can change these values internally. So, to change an attribute, a method to apply that change must be implemented. With private setters, it is possible to have much better control of the attribute values of the Order Occurrence entity, and safety, since the entity might verify if the value is a valid one. If public setters were used, consumers of this entity would have the freedom to change its values without any restriction, and that could induce to have an entity with wrong values, breaking the domain.

An entity is always initialized by its own constructor. This allows that an entity can validate its initial values that are passed by the constructor, not allowing to be initiated with values that they are not in agreement with.

Behavior is something that also exists in an entity. Behavior means the domain logic that is applied about a business, and with that, an entity might have this logic in a constructor and in specific methods. Details about behavior in domain model objects, such as entities, are provided in section 5.4.6.

5.4.3 Value Objects Implementation

As previously related, Value Objects are objects that are used to describe certain aspects of a domain, and which do not have an identity. In this section, an example of a Value Object implementation is provided, with the goal to demonstrate the guidelines that are followed in value objects development in this project.

Every domain object that is considered as a Value Object must inherit from Value Object object from Domain Infrastructure (section 5.3). In this scenario, the value object Ticket Field from the domain model related to Order Occurrences inherits from the Value Object object:

```
public class TicketField : ValueObject {}
```

Code 18 - Ticket Field Value Object definition

Related with the Ticket Field attributes, this value object was designed with two attributes: key and value. These attribute values are related to the Form Field from Filled From aggregate, so the Ticket Field is not considered an entity but a value object. Next, it is demonstrated the attributes implementation:

```
public class TicketField : ValueObject
{
    public string Key { get; }

    public string Name { get; }

    public TicketField(string key, string name)
    {
        this.Key = key;
        this.Name = name;
    }
}
```

Code 19 - Ticket Field Value Object implementation

In the previous code, it demonstrates that Key and Name attributes are implemented as public ones, but having no setters. Such as an Entity, a value object is always initialized by its own constructor. This allows that a value object can validate its values that are passed by the constructor, not allowing to be initiated with values that are not in agreement.

Since it has no setters, the used values in its initialization are the same as the ones at the end of its cycle life. With this approach, once a value object is initialized, it forces immutability, not allowing changes applied to it, not even changes applied by itself. This means that if it is necessary a change on a value object, it forces to create a new instance instead of changing the current one, following the suggestions of Domain Driven Design related with Value Objects, providing more safety in the system data integrity.

A Value Object has no identity, and with that, it only can be compared with another Value Object by its attribute values. Since it inherits from a Value Object object infrastructure, then it only needs to implement the following method:

```
protected override IEnumerable<object> GetAtomicValues()
{
    yield return Key;
    yield return Name;
}
```

Code 20 - Ticket Field Atomic Values

This method is responsible for gathering the Value Object attributes that are necessary to compare it with another object. Then, its infrastructure object is responsible to compare these attributes one by one, abstracting this responsibility from the Value Object itself. With that, the Value Object only needs to define what are the values which are used during its comparison with other ones.

In the same way as an Entity, behavior exists in a Value Object. Details about behavior in domain model objects, such as value objects, are provided in section 5.4.6.

5.4.4 Aggregates Implementation

As defined in Domain Driven Design, Aggregates are groups of associated objects which are considered as one unit about data changes, being demarcated by a boundary that separates the objects inside from those outside. In this section, an example of an Aggregate implementation is provided, with the goal to demonstrate the guidelines that are followed in the aggregates development in this project.

Aggregates are constituted by an Aggregate Root and by its inner elements. Starting by the root, every domain object that is considered as an Aggregate Root must inherit from Aggregate Root object from Domain Infrastructure (section 5.3). In this scenario, the aggregate root object

Cause from the domain model related to Order Occurrences inherits from the Aggregate Root object:

```
public class Cause : AggregateRoot<int> {}
```

Code 21 - Cause Aggregate Root definition

Related with the Cause attributes, this aggregate root was designed with five attributes: Id, Name, Description, Form Template and Requirements. Inheriting from an Aggregate Root, it behaves like an Entity, forcing this class to have the Id attribute. So, it is only necessary to implement other attributes:

```
    public class Cause : AggregateRoot<int>
    {
        public string Name { get; private set; }

        public string Description { get; private set; }

        public FormTemplate FormTemplate { get; private set; }

        public IEnumerable<Requirement> Requirements { get; private set; }

        public Cause(string name, string description, FormTemplate
formTemplate, IEnumerable<Requirement> requirements)
        {
           this.Name = name;
           this.Description = description;
           this.FormTemplate = formTemplate;
           this.Requirements = requirements;
        }
    }
```

Code 22 - Cause Aggregate Root implementation

In the previous code, it demonstrates that the attributes are implemented in the same way as an Entity.

Having the root element implemented, it is necessary to develop the other elements that are contained inside the aggregate. These elements inherit from an abstract class such as Entity or Value Object, and additionally, implement the Aggregate Element interface. As previously related, this interface is empty and it is implemented only to sign that the element below to some aggregate.

In this scenario, a Requirement is recognized as a Value Object, bellowing at the same time to the Cause aggregate. With that it inherits from the Value Object object, and implements the Aggregate Element interface:

```
public class Requirement : Entity<string>, IAggregateElement {}
```

Code 23 - Requirement Aggregate Element definition

The Requirement attributes are implemented as a normal Value Object:

```
public class Requirement : ValueObject, IAggregateElement
{
    public string FieldKey { get; }

    public string RequirementInJSON { get; }

    public Requirement(string fieldKey, string requirementInJSON)
    {
        FieldKey = fieldKey;
        RequirementInJSON = requirementInJSON;
    }
}
```

Code 24 - Requirement Aggregate Element implementation

Furthermore, the aggregate root is always the responsible to execute actions inside the aggregate, that were requested by outside, having absolute control of it, and having the responsibility to guarantee the aggregate data integrity.

5.4.5 Events Implementation

In this section, domain and integration events implementation are explained.

5.4.5.1 Domain Events Implementation

Starting by domain events, when some domain action occurs, it might be necessary to notify the remaining domain that an event has occurred, in order to the other interested domain objects consume that information, and execute other tasks. Domain events help in expressing the domain rules, and also enable better separation of concerns among classes within the same domain.

Imagining an Order Occurrence creation, a service method to validate the occurrence, from Domain Layer, is called by an Application Service, and an Order Occurrence is passed:

```
        public async Task Validate(OrderOccurrence orderOccurrence)
        {
            //Business logic explained further in the document
            //If validation fails, it runs orderOccurrence.Invalid() and
an exception is thrown

            orderOccurrence.Valid();
        }
```

Code 25 - Order Occurrence validation

In the previous code, some business logic (explained in 5.4.6.2) is executed, and if its successes, the Order Occurrence is capable to be persisted in the system. In the next code sample, the flow of an Order Occurrence creation, in the application service, is demonstrated:

```
        public async Task<int> AddOrderOccurrence(OrderOccurrenceDTO
orderOccurrenceDTO)
        {
            var orderOccurrenceDomain =
mapper.Map<OrderOccurrence>(orderOccurrenceDTO);

            await this.validationService.Validate(orderOccurrenceDomain);

            var id = await
orderOccurrencesRepository.AddOrderOccurrence(orderOccurrenceDomain);

            await orderOccurrencesRepository.SaveChanges();

            return id;
        }
```

Code 26 - Order Occurrence creation at Application Service

The Order Occurrence DTO is mapped to a domain object. Thereafter, the domain object is validated by a domain service, and when it is valid, it is persisted by the Repository, and then, the repository atomically persists all the changes when it is called with SaveChanges().

After the Order Occurrence was considered as valid by the domain service, a domain event was created:

```
public class OrderOccurrenceCreated : IDomainEvent
{
    public OrderOccurrence OrderOccurrence { get; }

    public OrderOccurrenceCreated(OrderOccurrence orderOccurrence)
    {
        this.OrderOccurrence = orderOccurrence;
    }
}
```

Code 27 - Order Occurrence Created Domain Event

An Order Occurrence Created event was created by the domain after the Order Occurrence was considered as valid. This event contains the information about the Order Occurrence that was created, and it was generated in the Order Occurrence Valid() method:

```
public void Valid()
{
   Events.Add(new OrderOccurrenceCreated(this));
}
```

Code 28 - Domain Event added to the Entity List

In the previous code, it is possible to check that the Order Occurrence entity, adds an Order Occurrence Created event to its domain events list. With this, the generated domain event will be raised to the system at any moment, and be consumed by the interested. A sequence diagram resumes the executed flow about the previous domain event creation:

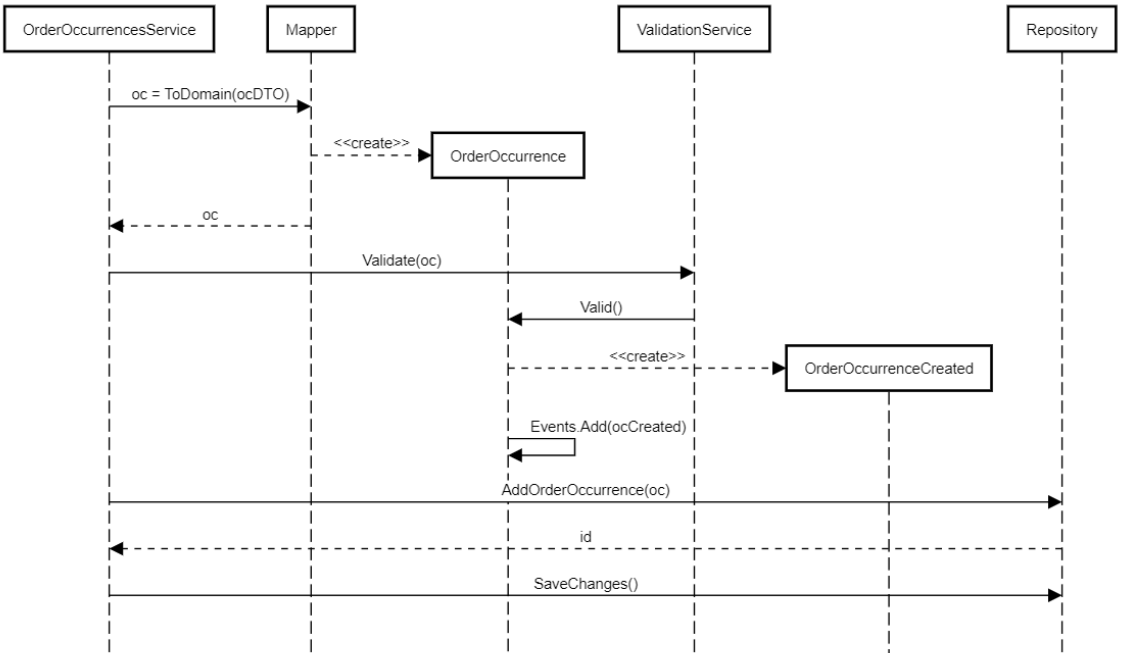


Figure 55 - Domain Event creation in Order Occurrences

The subscribers can consume an event by implementing the Notification Handler interface from MediatR framework, which is responsible to act as a mediator between the component that creates an event and the component that consumes it. Implementing this interface, it is necessary to define the intended event to be consumed by defining the generic type of Notification Handler interface, and then, the Handle method must be implemented with the desired action at the moment that the event is consumed:

```
    public class OrderOccurrenceCreatedHandler :
INotificationHandler<OrderOccurrenceCreated>
    {
        private readonly ITicketsRepository ticketsRepository;

        public OrderOccurrenceCreatedHandler(ITicketsRepository
ticketsRepository)
        {
            this.ticketsRepository = ticketsRepository;
        }

        public async Task Handle(OrderOccurrenceCreated orderOccurrenceCreated,
CancellationToken cancellationToken)
        {
            var ticket = new Ticket(orderOccurrenceCreated.OrderOccurrence);

            await ticketsRepository.AddTicket(ticket);
        }
    }
```

Code 29 - Order Occurrence Created Domain Event handling

In the previous code, the handler is responsible to consume the Order Occurrence Created event, and it is possible to verify that, at the moment it consumes the event, it creates a new Ticket and adds it to the database. This handler is responsible to establish a domain communication between different business contexts (Order Occurrence and Ticket), without delegating that responsibility, for example, to an application service. The domain events can be handled by application services, but only for the creation of specific tasks nonrelated to the domain. This will be further explained with integration events.

5.4.5.2 Integration Events Implementation

The purpose of integration events is to propagate transactions and updates to additional subsystems, whether they are other microservices, Bounded Contexts or even external applications. This is done by publishing integration events outside the microservice.

Imagining a domain event, known as Ticket Created, similar to the domain event related to the Order Occurrence creation. It can be handled by any subscriber in the Domain Layer, but also in the Application Layer. A Ticket creation is something that is interesting to publish outside the service, in order to provide information to the platform. This is useful for example, for support applications, that need these kinds of information, in a way to communicate them to support teams.

To handle this domain event to transform into an integration event, and to publish the created integration event, a handler was created on the Application Layer:

```
    public class TicketCreatedHandler : INotificationHandler<TicketCreated>
    {
        private readonly IEventPublisher publisher;
        private readonly IMapper mapper;

        public TicketCreatedHandler(IEventPublisher publisher, IMapper mapper)
        {
            this.publisher = publisher;
            this.mapper = mapper;
        }

        public async Task Handle(TicketCreated ticketCreated, CancellationToken
cancellationToken)
        {
            var ticketDto = this.mapper.Map<TicketDTO>(ticketCreated.Ticket);

            var ticketCreatedIntegrationEvent = new
TicketCreatedEvent(ticketDto);

            await this.publisher.Publish(ticketCreatedIntegrationEvent);
        }
    }
```

Code 30 - Ticket Created Integration Event creation

The previously demonstrated handler is very similar to the Order Occurrence Created handler from section 5.4.5.1. But this one is connected to a publisher interface, which is responsible to publish integration events into a bus, providing the event information outside the service. Due to that, when a Ticket is created, this handler is notified and maps its information into an object that can be consumed by outside, in this case, a DTO, and then creates an integration event with that information, publishing it by calling the publisher interface. This integration event object is very similar to a domain event object:

```
public class TicketCreatedEvent : IIntegrationEvent
{
    public TicketDTO Ticket { get; }

    public TicketCreatedEvent(TicketDTO ticket)
    {
        Ticket = ticket;
    }
}
```

Code 31 - Ticket Created Integration Event

It also implements an interface, known as Integration Event, which is an empty one, but that is used to signal this class for the system as an integration event.

Scaling a domain event to an integration event is clean, and does not require any extra efforts in its implementation, because the handler is created in the same way as a domain event handler, and it only has the responsibility to orchestrate the system in order to create an integration event.

5.4.6 Domain Behavior Implementation

Domain behavior represents the business logic rules, existent for the real-world. With this, the idea is to implement these behaviors on the Domain Layer, instead of doing it in other layers, as it is done in the monolithic.

Different types of domain behaviors are implemented on domain model objects, such as entities, value objects, aggregate roots, domain services, and events. During this section, these types of behaviors are explained.

5.4.6.1 Behavior on Domain Models

Behavior on Domain Model objects might be expressed on constructors or methods. For example, when the entity Cause is initialized, it must validate if its requirements have different field keys from each other because it is a business rule. This validation must occur on the entity initialization since it must have valid data after being initialized. With that, this validation must be executed in the Cause constructor:

```
        public Cause(string name, string description, FormTemplate formTemplate,
IEnumerable<Requirement> requirements)
        {
            Name = name;
            Description = description;
            FormTemplate = formTemplate;
            Requirements = requirements;

            if(RequirementsAreInvalid(requirements))
            {
                throw new Exception("Specified requirements are invalid");
            }
        }

        private bool RequirementsAreInvalid(IEnumerable<Requirement>
requirements)
        {
             return requirements.GroupBy(rr => rr.FieldKey)
                                .Any(c => c.Count() > 1);
        }
```

Code 32 - Cause Domain behavior

In the previous code, it is possible to verify that a private method, known as RequirementsAreInvalid(), is called at the moment of the initialization of the Cause object, in the constructor. If the requirements are not valid, an exception is thrown and the domain model object is not initialized.

The previous example of behavior occurs in an initial phase of the entity cycle life. But behavior might occur after an object (Cause) is initialized, with public methods that might be called by other objects:

```
        public void SetNewRequirement(string fieldKey, string json)
        {
            var keyAlreadyExists = this.Requirements.Any(r => r.FieldKey ==
fieldKey);

            if (keyAlreadyExists)
            {
                throw new Exception("Provided field key for new requirement
already exists");
            }

            var requirement = new Requirement(fieldKey, json);
            this.Requirements.Add(requirement);
        }
```

Code 33 - Set new Requirement method

In the previous code, this method allows setting a new requirement for a Cause. Since this object is an Aggregate Root, it is responsible to manage its aggregate elements. With that, when a new Requirement is necessary for a Cause, the method SetNewRequirement() is called.

This method validates if the provided field key already exists in the current Requirements of the Cause object, and a new Requirement is created and inserted into the aggregate, only if the field key does not exist in current requirements. This behavior exists after the Cause object is already initialized, and in some moment of its cycle life, a new requirement was requested to exist in the Cause object.

Behavior can also occur in the set of a property. For example, a Ticket has a priority, which has many levels. With a Ticket, to define a new priority, it would only be necessary to create a method that sets its:

```
public void SetPriority(TicketPriority newPriority)
{
    this.Priority = newPriority;
}
```

Code 34 - Set Ticket Priority method

But sometimes, this kind of setters are unnecessary, because they are only used to set the same value. In the Order Occurrences business context, a Ticket priority only changes to be maximized. Otherwise, it never changes. With that, it is not necessary to implement the previous method, instead, the next method could be implemented:

```
public void MaximizePriority()
{
    this.Priority = TicketPriority.Critical;
}
```

Code 35 - Maximize Ticket Priority method

The previous method is much more contextualized with the current business behavior, contrary to the first method. It abstracts the responsibility for other objects, to know what is the higher priority value, in order to set it.

This first method could be the better approach if the Ticket priority could change to different values during its cycle life, but since it only changes to the maximum value in the current business, the last method is a better option.

A third option would be the implementation of both methods. This would be valuable if the Ticket priority could change to different values, but that it constantly changed to maximum priority.

5.4.6.2 Behavior on Domain Services

The domain services are responsible services related directly to the domain. These mainly orchestrate and apply business rules across different entities or aggregates, which cannot be applied by individual ones. With this, domain services also hold behaviors about the domain.

Typically, a domain service is a class, that implements an interface, with methods that work together in order to achieve some behavior about business.

During an Order Occurrence creation, it must be validated, as it was related before, but this validation requires interaction between elements of an Aggregate (Cause) and other elements of the domain. Having this, it is necessary to create a domain service (ValidationDomainService) with the responsibility to validate an Order Occurrence:

```
        public async Task Validate(OrderOccurrence orderOccurrence)
        {
            var cause = await
causesRepository.GetCause(orderOccurrence.CauseId);

            var validations = cause.GetValidations();

            foreach (var validation in validations)
            {
                var validationResult =
validation.ProcessValidation(orderOccurrence);

                if (!validationResult.IsValid())
                {
                    orderOccurrence.Invalid(validationResult.Reason());
                    return;
                }
            }

            orderOccurrence.Valid();
        }
```

Code 36 - Order Occurrence validation with Domain Service

In the previous code, it is possible to verify that a method is responsible to validate the Order Occurrence. It gets the Cause, then it obtains the Cause validations, which are validations (of OrderOccurrenceValidation type) exported by the Cause, in order to validate its requirements outside its aggregate, and finally, each validation is processed with the information about the Order Occurrence. If some validation result is not valid, the Order Occurrence is invalidated with a reason, otherwise, it is validated.

The previous method is an example of a domain service, where it accesses to repositories to gather domain information, but where it also manages domain behaviors, such as validations, which could not be held by application services, because there was knowledge about the domain, and not only service task orchestrations.

5.4.6.3 Behavior on Domain Events

When a domain event is thrown, it is possible to handle multiple actions related to other aggregates in the same domain, but it is also possible to run additional application actions, connecting these events with integration events and the event bus.

These domain events represent actions that were executed inside the domain, which might have an impact on other sections of the same domain. In section 5.4.5.1, it is possible to verify how the domain talks with itself, exposing its behavior in an event-driven way. When an Order

Occurrence is validated, a new Ticket object from other aggregate is created by the handler that subscribes the Order Occurrence Created event, and nothing outside the layer is orchestrating the interaction of these domain publishers and subscribers.

Furthermore, with domain events, business behaviors are exposed by the domain for the system, and these notifications might be consumed by other layers, in order to expose relevant information provided by the application domain to outside, by using, for example, integration events as it is explained at section 5.4.5.2.

5.5 Return Occurrences Model Implementation

For Return Occurrences, this model was already designed in section 4.5.2, with its entities, value objects, and aggregates. Software patterns and architectures were also approached in turn of this model, and during this chapter of Implementation, it was possible to check how everything was built. The goal is also to achieve a correct usage of Domain Driven Design in this project, to solve the constraints of the project problem.

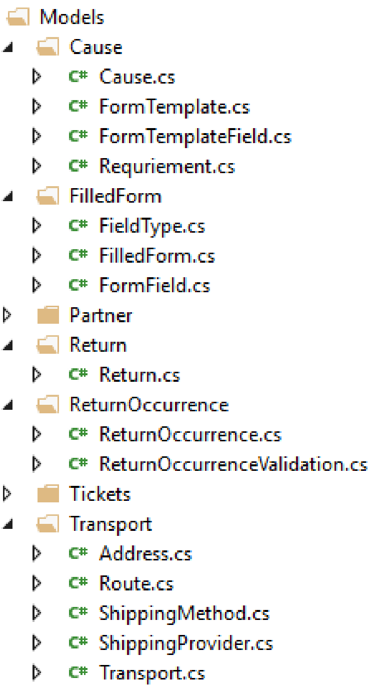
During the previous section related to Order Occurrences Model Implementation, many topics were approached: the Domain Model structure, the domain objects implementation such as entities, value objects, aggregates and events, and the behavior explanation. The implementation strategy for the Domain Model related to Order Occurrences is the same as the one used for the Return Occurrences, except for the Domain Model structure, which is similar but not the same.

In this project implementation, everything else related to Domain is implemented in the same way as Order Occurrences project. Using this approach, it is expected that the team has no constraints in the moment of understanding the domain implementation, and in the moment of evolve it, because both services implement the same domain model implementation, differing only in the software architecture, where one uses a Layered Architecture from DDD, and the other uses a Clean Architecture with CQRS.

Having this, the current section only demonstrates the Domain Model structure that is implemented in this project. For this, a structure was defined with three main folders: Events, Infrastructure, and Models.

5.5.1 Models

The Models folder contains every domain model object related to the project. This folder contains many subfolders, where each one belongs to a specific business concept or to a related aggregate:



```
Models
  Cause
    Cause.cs
    FormTemplate.cs
    FormTemplateField.cs
    Requriement.cs
  FilledForm
    FieldType.cs
    FilledForm.cs
    FormField.cs
  Partner
  Return
    Return.cs
  ReturnOccurrence
    ReturnOccurrence.cs
    ReturnOccurrenceValidation.cs
  Tickets
  Transport
    Address.cs
    Route.cs
    ShippingMethod.cs
    ShippingProvider.cs
    Transport.cs
```

Figure 56 - Domain Models in Return Occurrences

Having this, every domain model object is contained in the same parent folder Models, and it is possible to reunite every model object, without distinct the aggregates from the rest, the aggregates are contained in specific subfolders, in order to not being mixed with other model objects.

5.5.2 Transport Aggregate

One interesting fact about this Return Occurrences Model is how the Transport Aggregate is built. When one Return Occurrence is created, the system needs to handle some data related to Transport of a Return, in order to apply validations for that Return Occurrence.

To achieve this, the Return Occurrences Service must request data from Transport bounded context, by calling the desired method of Transportation Service interface.

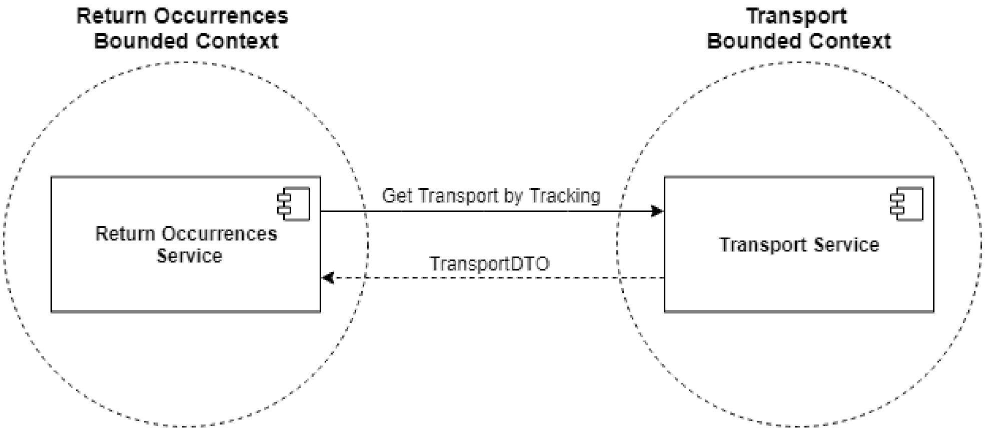


Figure 57 - Interaction between Transport and Return Occurrence bounded contexts

During the Return Occurrence creation, its corresponding Use Case handler is responsible to handle the request, executing the following code:

```
        public async Task<int> Handle(CreateReturnOccurrenceCommand request,
CancellationToken cancellationToken)
        {
            var returnOccurrence
                 = await
this.mapper.Map<ReturnOccurrence>(request.ReturnOccurrence);

            var transportDTO
                = await
this.transportServiceGateway.GetTransportByTracking(returnOccurrence.Return.Trac
king);

            var transport
                = new TransportBuilder()
                          .WithTransportId(transportDTO.Id)
                          .WithShippingProvider(transportDTO.Provider.Id,
transportDTO.Provider.Name)
                          .WithShippingMethods(transportDTO.Provider.Methods)
                          .WithRoute(transportDTO.OriginAddress,
transportDTO.DestinationAddress)
                          .Build();

            await this.validationService.Validate(returnOccurrence, transport);

            //... Code after
        }
```

Figure 58 - Return Occurrence creation

When the handler receives the request, it maps the Return Occurrence DTO into a domain class. After that, it is responsible to call the transport service gateway, in order to obtain information related to the Transport of the Return. When this gateway is called, Return Occurrences and Transport bounded contexts interact with each other, with a provided communication between the services.

When Return Occurrences Service receives the information about Transport, it calls a Transport Builder that is responsible to create the entire Transport Aggregate, by mapping the necessary attributes from the Transport DTO. With this, the Transport concept of Return Occurrences bounded context is created only with the classes and proprieties, that are required to fulfill the needs of its context. In this scenario, this aggregate is used for Transport validations about a Return Occurrence.

5.5.3 Events and Infrastructure

The Events and Infrastructure folders are organized in the same way as in Order Occurrences Model. There is nothing added differently in this project about this.

5.6 Summary

During this chapter, it is possible to see that SOLID principles are respected. An example of the Single Principle responsibility, are the value objects implementation, where each class selects the attributes necessary to compare with another object, and then the responsibility of this comparison is delegated to the Value Object infrastructure object.

Domain Driven Design is implemented in overall, even with different software architectures being used. Using this Domain Driven implementation, every domain object is identified. For example, every domain model object inherits from an infrastructure object, which announces if that object is an Entity or a Value Object for example. Also, aggregates and domain events are possible to be identified by its infrastructure object.

Business behavior is spread over the Domain Layer, by domain models, domain services and domain events. Application services are not responsible for business logic, contrary to the monolithic service. They are only responsible for orchestrating application tasks, respecting the Single Principle responsibility.

Furthermore, both services implement different software architectures, but they still very similar to each other, since they both follow an architecture based on layers.

6 Evaluation

The evaluation chapter describes the evaluation process for the implemented solution, presenting the measurements to use, defining hypotheses intended to be tested in order to support the work results, detailing the evaluation methodology and hypotheses testing.

The measurement property used in solution evaluation is the Domain Solution Suitability.

Next, this property is detailed, with the hypotheses being defined for the corresponding one. This chapter ends with the solution results provided in a survey by the team developers.

6.1 Indicators Identification and Information Sources

In this section, indicators identification and information sources about the solution evaluation are performed. These are necessary to define strategies for solution evaluation and to define what information must be gathered to support solution results.

6.1.1 Indicators Identification

Related with the solution evaluation, there is the main factor to be evaluated: Domain Solution Suitability. With this, it is necessary to identify the indicators for this factor, in order to have concrete parameters to evaluate the solution.

To evaluate the Domain Solution Suitability, two indicators are used: Business Understanding and Evolution Capacity. The following table describes the meaning of the related indicators:

Table 21 - Indicators for the Domain Solution Suitability

Indicator	Description
Business Understanding	Understanding of business concepts, relationships, and behavior, in code, provided by the solution.
Evolution Capacity	Domain evolution capacity provided by the solution, for new business functionalities or changes.

6.1.2 Information Sources

For solution evaluation, related to the previously defined indicators, it is necessary to gather information sources for the Domain Solution Suitability. These sources are important to be gathered, so the evaluation can use it to obtain a result, which can prove how successful the solution is.

Developers are the only information source that can be used to evaluate the provided solution. Since, they are the elements with the mission to develop new business requirements or related changes, having the need to code those requirements, and to work with the corresponding domain. Having this, these elements have the appropriate profile to evaluate the provided solution. With that, it is possible to gather their feedback about it.

6.2 Domain Solution Suitability

As previous related, team developers are responsible to evaluate the solution provided for the domain in Order Occurrences and Return Occurrences.

The Domain Solution Suitability is necessary to be evaluated in order to analyze if the implemented approach has a positive impact in the business understanding, and in the domain evolution capacity for new business functionalities, in each microservice, eliminating the constraints of the domain for the Occurrences software.

6.2.1 Evaluation Methodology

To measure the Domain Solution Suitability, a survey oriented for team developers was performed, with targeted questions for the evaluation of satisfaction related to the business understanding, and software evolution capacity provided by the solution. This survey was available after the development of both Order Occurrences and Return Occurrences services.

The inquiry is constituted by two groups of questions, one related to the factor of business understanding and the other related to the factor of domain evolution capacity. Each question, categorized as closed-ended, can be answered with a value of Likert scale.

Table 22 - Likert Scale
(Likert, 1932)

Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

The values of each group are collected, and with that, it is possible to calculate the average for each one in order to evaluate the success of the solution in the desired factors. The evaluation result associated with each factor will be classified taking into consideration the interval at which the average value is present. The following tables represent the defined intervals for success evaluation related to business understanding and software evolution capacity.

Table 23 - Interval for Business Understanding Evaluation

Interval	Result Meaning
[1-2]	The provided solution does not represent the business concepts, relationships, and behaviors, being hard to developers understand it. The evaluation of business understanding has failed.
[2-3]	The provided solution represents the business concepts, relationships, and behaviors in an insufficient or inefficient way, having issues that complicate the understanding for developers. The evaluation of business understanding has failed.
[3-4]	The provided solution represents the business concepts, relationships, and behaviors with an accepted understanding for developers, having room for improvements. The evaluation of business understanding has succeeded.
[4-5]	The provided solution represents the business concepts, relationships, and behaviors in an efficient way, with a clear understanding of developers. The evaluation of business understanding has succeeded.

Table 24 - Interval for Evolution Capacity Evaluation

Interval	Result Meaning
[1-2]	The provided solution is badly designed, and it is not prepared to evolve for new business requirements or changes. The evaluation for evolution capacity has failed.
[2-3]	The provided solution has design issues, and it will cause difficulties to evolve for new business requirements or changes. The evaluation for evolution capacity has failed.
[3-4]	The provided solution is well designed, and it is enough to evolve for new business requirements or changes, although it still has room for improvement. The evaluation for evolution capacity has succeeded.

[4-5]	The provided solution has a high-quality design, being efficient to evolve for new business requirements or changes. The evaluation for evolution capacity has succeeded.

Having the average results for these two factors it is possible to calculate how success the solution was. For this, the average result of business understanding and evolution capacity must be added together and then divided by two. Then, this result used by hypotheses test, which is explained in the next section.

6.2.2 Hypothesis Testing

To perform a hypothesis testing about the Domain Solution Suitability will be used a One-Tailed Testing. It allows to determine if one mean is greater or less than another mean, but not both. A direction must be chosen prior to testing. With this, one-tailed tests express the effect of a change in one direction and not the other (Vallee, 2015).

There are two kinds of hypotheses for a One-Tailed Testing: the null hypotheses and the alternative hypotheses.

H_0: The null hypothesis is the considered hypotheses that analyst hopes to reject.

H_1: The alternative hypothesis is the considered hypotheses that is supported by rejecting the null hypothesis.

Applying One-Tailed Testing to the property of the Domain Solution Suitability, the average result of the answers gathered in the survey about the business understanding and evolution capacity is the parameter used for the following formulated hypotheses:

H_0: The implemented solution did not eliminate the problem constraints.

$$H_0 : \mu \leq 3.5 \quad (1)$$

H_1: The implemented solution eliminated the problem constraints.

$$H_1 : \mu > 3\ .5 \quad (2)$$

With the use of One-Tailed Testing, it will be possible to define the evaluation of the Domain Solution Suitability related to the result of the project solution.

6.3 Evaluation

The evaluation was a continuous process during project development. Many phases were performed until the final survey was available to the developers, having an important role in order to evaluate this project's success. This evaluation process was divided into six interconnected steps.

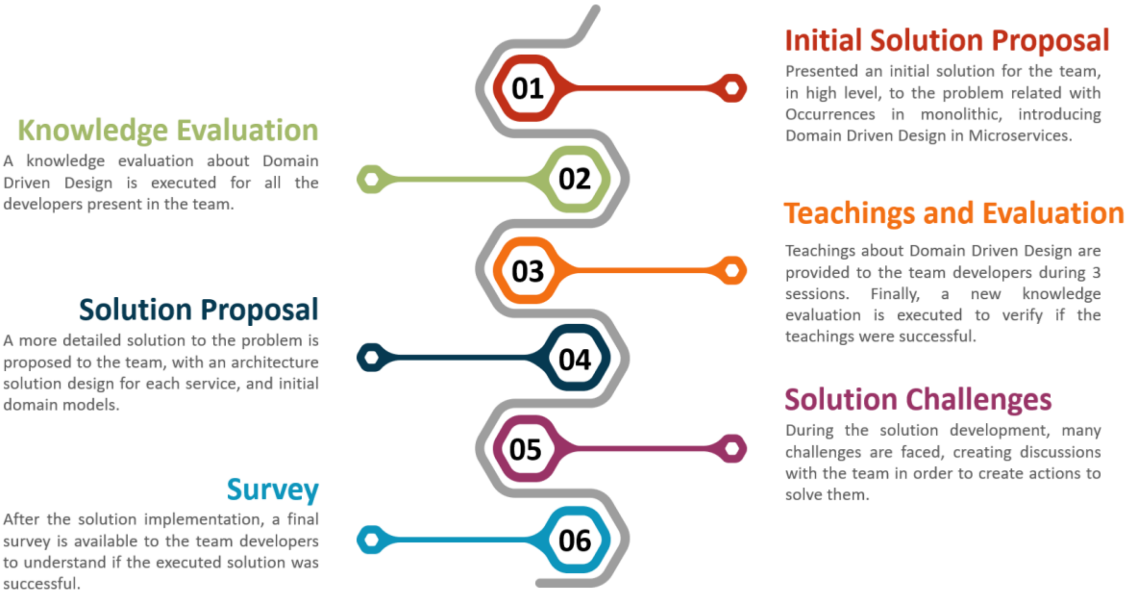

01
Initial Solution Proposal
Presented an initial solution for the team, in high level, to the problem related with Occurrences in monolithic, introducing Domain Driven Design in Microservices.
02
Knowledge Evaluation
A knowledge evaluation about Domain Driven Design is executed for all the developers present in the team.
03
Teachings and Evaluation
Teachings about Domain Driven Design are provided to the team developers during 3 sessions. Finally, a new knowledge evaluation is executed to verify if the teachings were successful.
04
Solution Proposal
A more detailed solution to the problem is proposed to the team, with an architecture solution design for each service, and initial domain models.
05
Solution Challenges
During the solution development, many challenges are faced, creating discussions with the team in order to create actions to solve them.
06
Survey
After the solution implementation, a final survey is available to the team developers to understand if the executed solution was successful.

Figure 59 - Evaluation Phases

These phases constitute a set of challenges that existed during the development of the solution to the project problem. Each one is explained in the further sections. The last one, the survey that is addressed to the team developers, quantifies the success of this solution implementation with its results.

In the next figure, it is possible to see the timeline of this project with the previous phases.

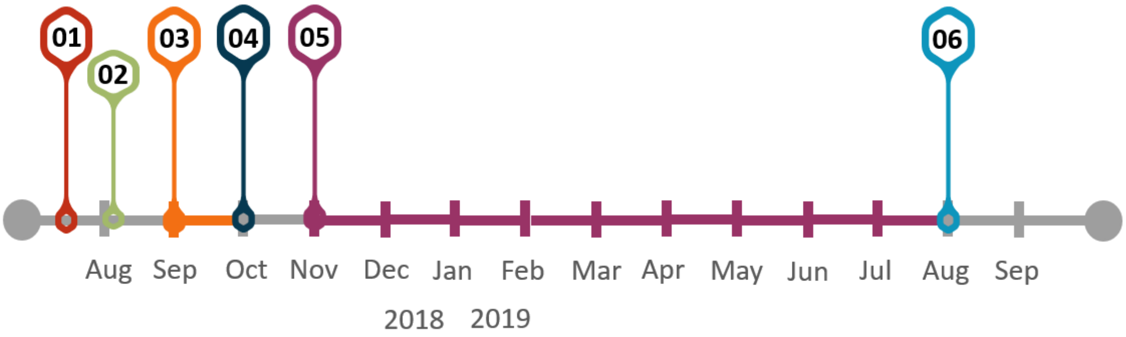


Figure 60 - Timeline of Evaluation Phases

During this section, the number of developers considered is always four, since the student does not count for the evaluation.

6.3.1 Initial Solution Proposal

This first phase started two months and a half before the beginning of this project implementation, with an initial solution proposal for the team, in a high level, to the problem related with Occurrences software inside the monolithic, by introducing the approach of Domain Driven Design in the context of Microservices.

To introduce this, a presentation was prepared and addressed to the team, with one hour of duration, by presenting many topics related to Domain Driven Design:

- What is Domain Driven Design?;
- Domain Driven Design Patterns;
- Domain Driven Design in Microservices;
- Domain Driven Design applied to the Occurrences software.

Every topic was briefly detailed in order to not consume much time of the presentation, and only to catch the team's interest in this kind of solution.

In the end, voting was taken with the team developers, in order to find out if they were interested to invest time in this solution. This voting happened in person.

Table 25 - Initial Solution Proposal voting

Vote: Are you interested to invest time to solve the Occurrences software constraints, by implementing a solution using a Domain Driven Design approach?	
Answer	**Number of votes**
Yes	3
No	1

With the result, three members were interested and one not interested, which results in a 75% positive feedback from the team developers to invest time in this solution. Only one member voted against, having the following justification: “I never experienced this approach, and I am afraid that the team is not fully capable to implement a solution using Domain Driven Design correctly”.

This was a very valid justification, and it was the first challenge that this solution had to face. With that, a second step was taken in this evaluation process: the knowledge evaluation.

6.3.2 Knowledge Evaluation

This phase was performed one week and a half after the first phase. It consisted of a knowledge evaluation about Domain Driven Design, executed for all the developers present in the team. Since the team was receptive to invest efforts in the proposed solution, it was necessary to evaluate their knowledge in order to understand how prepared is the team to face a different approach.

To do this, a survey with questions of open answers, related to Domain Driven Design, was performed. These questions were pure theorical, as for example, the following ones:

- Please describe the Bounded Context meaning;
- Please describe the Ubiquitous Language meaning;
- Please describe the characteristics of an Entity;
- Please describe the characteristics of a Value Object;
- Please describe the characteristics of an Aggregate;

Other questions were also performed, and with this, it was possible to check, at a high level, the knowledge of the team developers about this topic. To evaluate the results of this survey, the overall answers of each developer were evaluated subjectively by the student, with the support of an experienced architecture engineer, which does not belong to the team.

Table 26 - Knowledge Evaluation Results

Result	Number of Developers
The developer has a high degree of knowledge about Domain Driven Design.	1
The developer has any knowledge about Domain Driven Design.	2
The developer has little or no knowledge about Domain Driven Design.	1

Having the previous results, it is possible to verify that the team had knowledge about Domain Driven Design, but that needed teaching in order to be fully capable to implement this strategy with quality. One developer was considered already capable, while the other two had any knowledge but needed to learn more about it. Finally, one element was not comfortable with this approach, having little knowledge about Domain Driven Design.

With this, the goal was to invest in teachings to make the team have at least two members with a high degree of knowledge, and no one without little or no knowledge. To achieve this, the next phase was executed: teachings and evaluation.

6.3.3 Teachings and Evaluation

This phase was performed two weeks after the previous phase. It consisted of the teaching of Domain Driven Design for the team developers. To do this, three sessions of two hours were scheduled during the month of September. Each session approached topics related to Domain Driven Design, as it is demonstrated in the next table.

Table 27 - Teaching session topics

Session	Topics
1	Bounded Context, Ubiquitous Language
2	Domain Driven Design Patterns
3	Domain Driven Design in practice

The first two sessions were completely theorical, approaching the main aspects of Domain Driven Design, and the third one was a more practical session, where Domain Driven Design was presented with C# code samples related to the business of Occurrences.

The team enjoyed these sessions askings questions when it was necessary in order to understand all these concepts. Having this, it was expected that everyone improved their knowledge about Domain Driven Design and be more prepared for the challenge of the proposed solution implementation.

With that, a new evaluation was executed, in the same way as the second phase, having the following results:

Table 28 - Knowledge Evaluation Results after teaching sessions

Result	Number of Developers
The developer has a high degree of knowledge about Domain Driven Design.	2
The developer has any knowledge about Domain Driven Design.	2
The developer has little or no knowledge about Domain Driven Design.	0

With the previous table, it is possible to verify that knowledge results improved since the last time it was evaluated. This time, the answers defined the Domain Driven Design concepts significantly better than before, even for the ones that were not considered as a high degree. With success, no one had no knowledge about DDD, which was an important step.

Having this, it was expected that knowledge about Domain Driven Design would be improved during the project development, with the members of the team sharing their knowledge with each other, and also improving with the business obstacles ahead.

Ending this phase, it was time to propose a detailed solution for the team.

6.3.4 Solution Proposal

This phase was performed after the previous one has ended. With the team successfully engaged with the Domain Driven Design, it was time to propose a detailed solution for the problem.

To achieve this, the bounded contexts delineation about Occurrences was proposed, as also the initial domain model for each one. To present these, a session of two hours was scheduled.

6.3.4.1 Bounded Contexts Proposal

Starting with this topic, a context mapping similar to the one presented in section 4.2 was presented to the team. It proposed the delineation of the two bounded contexts Order Occurrences and Return Occurrences, explaining the context mapping to support that proposal. Also, supporting these bounded contexts, the architectural design was presented with the two microservices Order Occurrences service and Return Occurrences service.

The team agreed with this proposal since it had arguments to eliminate constraints related to the problem. The team was convinced about this proposal because of two major benefits:

- The possibility to define a Ubiquitous Language for each bounded context;
- Order Occurrences and Return Occurrences services fitted well in a microservices architecture and were enough to support their bounded contexts.

6.3.4.2 Domain Models Proposal

The domain model for each bounded context was drawn in a non-detailed way because it would not be interpreted and agreed by the team in one unique session. Instead, each domain model would be designed and explored with the continuous development of the project, where the team members communicate with each other and discuss the meaning of the business concepts in the domain model. Having this, the team only interpreted the idea of how each domain model was initially designed, defining the following objectives:

- Build a Ubiquitous Language for each bounded context;
- Design the domain models taking into consideration the Ubiquitous Language concepts;
- Implement the designed domain models in the corresponding microservices;
- Gather, align and refactor (if necessary) the Ubiquitous Language or the implemented domain models.

Following this strategy, it is necessary to follow the development process. That is where the next phase begins: Solution Challenges.

6.3.5 Solution Challenges

This phase was performed since the ending of the previous phase until the ending of the solution implementation. Having a solution proposal, it was necessary to start its implementation. During the developments, many meetings were scheduled, in order to discuss the challenges and to define actions to solve them, where each action had an impact on the final solution of the project.

In the next table it is possible to see the main challenges, with its actions, that existed during the project development:

Table 29 - Solution Challenges and Actions

Challenge	Action
Define Order Occurrences and Return Occorrences bounded contexts.	Gather the developers, and define the bounded contexts and its concepts.
Build a Ubiquitous Language for Order Occurrences.	Gather the developers, as often as needed, with the product owner, and define the language concepts and its definitions.
Design with high detail, the domain model for Order Occurrences.	Gather the developers and design the domain model, as often as needed, considering the initial domain model proposal.
Improve the Ubiquitous Language or the domain model of Order Occurrences.	Gather the developers, with the product owner, to align and refactor the domain model. At least one meeting per month should be scheduled in order to verify if the Ubiquitous Language or the domain model needs improvements.
Implement the software architecture of Order Occurrences service.	The student provides a skeleton implementation of each layer, creating the necessary projects and providing code samples.
Understand the meaning of each domain model class. For example, if a class is classified as an Entity or a Value Object.	Build a domain modeling common infrastructure. Additionally, the student provides code samples of Entities, Value Objects, and Aggregates implementation.
Define domain behaviors.	The student presents the different types of domain behaviors.
Provide communication between aggregates.	Design and implement domain events. The student provides an initial code sample for domain events implementation.
Provide the Tickets created to external services.	Design and implement integration events, using domain events. The student provides an initial code sample for integration events implementation.

Build an Ubiquitous Language for Return Occurrences.	Gather the developers, as often as needed, with the product owner, and define the language concepts and its definitions.
Design with high detail, the domain model for Return Occurrences.	Gather the developers and design the domain model, as often as needed, considering the initial domain model proposal.
Improve the Ubiquitous Language or the domain model of Return Occurrences.	Gather the developers, with the product owner, to align and refactor the domain model. At least one meeting per month should be scheduled in order to verify if the Ubiquitous Language or the domain model needs improvements.
Implement the software architecture of Return Occurrences service.	The student provides a skeleton implementation of each layer, creating the necessary projects and providing code samples.

This project had many challenges that forced the team to gather and to take some actions in order to solve them. In the previous table, it is possible to see that the Order Occurrences service had more challenges compared to the Return Occurrences one. This happened because the Order Occurrences context was designed and implemented first than Return Occurrences context, and with that, the team defined actions to the challenges of Order Occurrences, that were the same for Return Occurrences at the moment that it was developed.

6.3.6 Survey

Finally, this phase was executed in the last week of the project development. As previously related, a survey was provided to gather the team developers feedback, in order to make conclusions about the success of the implemented solution.

The possible answers were always with the same range, following the Likert scale as it was already demonstrated in table 22.

The survey aimed to be simple with few questions, but that at the same time, it gathered important answers to make conclusions about the solution. With fewer simple questions, it was believed that the survey was not tiring and that reliable answers were provided by the developers.

To answer this inquiry, a session of one hour was scheduled with the four developers, where the survey had an estimated duration of 30 to 45 minutes. In order to create and support the survey, Google Forms[1] platform was used.

[1] https://www.google.com/forms/about

6.3.6.1 Business Understanding

To evaluate the Business Understanding topic, 7 questions were performed:

1. Business concepts are well represented in the Domain Models;
2. Business relationships are clear in the Domain Models;
3. The Domain Models follow a Rich Domain Model strategy, with business behaviors represented on it;
4. Business relationships are clear in the Domain Models;
5. With this solution, a new joiner would have no difficulties in the understanding of business in the code;
6. The provided solution solved the problems related to business understanding, that existed on monolithic;
7. The provided solution can be easily applied to future projects, in order to have a clear business understanding in the code.

The questions above had the following results:

Table 30 - Inquiry results for Business Understanding

	Answers				
Question Number	1	**2**	**3**	**4**	**5**
1	-	-	-	1	3
2	-	-	-	2	2
3	-	-	-	4	-
4	-	-	1	2	1
5	-	1	1	1	1
6	-	-	-	1	3
7	-	-	1	2	1

Having compiled the previous answers, the result average of Business Understanding was 4.36, whose value is inside the range [4-5], which means that the provided solution represents the business concepts, relationships, and behaviors in an efficient way, with a clear understanding for developers, concluding that the evaluation for business understanding has succeeded.

6.3.6.2 Evolution Capacity

To evaluate the Evolution Capacity topic, 5 questions were performed:

1. With this solution, it is possible to implement new business requirements or changes, without difficulties;

2. This solution has a high-quality domain design, which is efficient to evolve the software for new business requirements or changes;
3. With this solution, a new joiner would have no difficulties to evolve the software for new business requirements or changes;
4. The provided solution solved the problems related to evolution capacity, that existed on monolithic;
5. This solution implemented can be easily applied to future projects, in order to have a significant software evolution capacity.

Table 31 - Inquiry results for Evolution Capacity

	Answers				
Question Number	1	**2**	**3**	**4**	**5**
1	-	1	2	1	-
2	-	-	1	1	2
3	-	1	1	2	-
4	-	-	-	2	2
5	-	-	1	2	1

Having compiled the previous answers, the result average of Evolution Capacity was 3.80, whose value is inside the range [3-4], which means that the provided solution is well designed, and it is enough to evolve for new business requirements or changes, although it still has room for improvements, concluding that the evaluation for evolution capacity has succeeded.

6.3.6.3 Hypothesis Testing

With an average of 4.36 for Business Understanding and 3.80 for Evolution Capacity, it is possible to calculate the final average, which totals in 4.08. Having this, it is possible to perform the hypothesis testing about the Domain Solution Suitability, with a One-Tailed Testing as it was described in section 6.2.2.

Since the average totaled 4.08, this means that the null hypotheses was rejected, since this value was superior to 3.5, supporting the alternative hypotheses.

With this, it is possible to conclude that the implemented solution eliminated the problem constraints and it was successful.

7 Conclusions

This chapter describes the realized objectives and future work, to improve in the developed project.

Finally, a short appraisal is made about the project as a whole and the learning at a personal level.

7.1 Achieved Objectives

The objective of this project was to create a solution that eliminates the constraints of the domain for the Occurrences software, during its migration for microservices, in a way that developers can have fewer difficulties and be satisfied during the implementation of new business requirements.

To achieve this object, two steps should be accomplished. Each one was classified as achieved or not achieved, having the corresponding justifications for its classification.

1. **First step**

The first defined step was to design a solution to migrate Occurrences software for a microservices oriented architecture, using Domain Driven Design, and applying design patterns whenever they are adequate for the problem.

Two bounded contexts were delineated, each one having a microservice. With this, it was possible to evolve the Occurrences software into a microservices architecture, being compliant with the platform trend. The definition of a Ubiquitous Language to each context, was also very important to gather and clarify business concepts into domain, assisting the domain models design.

During the project development, the solution was designed and redesigned many times as needed in order to approach a high-quality microservices implementation with Domain Driven Design. For this, many patterns of DDD were applied, such as Layered Architecture, Entities, Value Objects, Aggregates, Services, and Repositories. SOLID patterns were also carefully implemented, so its benefits could be achieved in the solution implementation.

Considering the previous justifications, this step was successfully achieved.

2. **Second step**

The second step was to implement technical aspects of Domain Driven Design related to the provided solution, that must be comprehended by the remaining elements of the team, and which helps the construction of a domain with quality, in the various microservices.

During the solution design, many aspects such as Software Architecture and Domain Driven Design patterns were implemented using code samples and examples that helped the team to understand its usage. This allowed them to use these implementations to develop the current and the next business requirements. The aspects that were implemented were always discussed in team, in order to everyone discuss and agree with it.

Considering the previous justification, this step was successfully achieved.

In conclusion, with the previous steps being achieved, it is possible to consider that the objective of this project was successfully accomplished.

7.2 Limitations

During the solution design and implementation of this project, one main limitation occurred.

The team developed the business requirements mainly in monolithic services, which were outdated or implemented without taking into consideration the best practices of software development. Having this, difficulties existed during the solution implementation in an initial phase, since the team had to develop in a DDD way, which was a different mindset from the usual, and that created many discussions and consequently some refactorings, what revealed some inexperience with domain implementation.

7.3 Future Work

With Order Occurrences and Return Occurrences services practically developed, still exist some improvements that should be implemented at the domain level, for possible new business requirements. During the solution implementation and its evaluation, many ideas for improvements were generated:

- Refactor Entities and Value Objects that which attributes are not totally well designed;
- Better separate some business concepts inside the aggregates;

- Migrate some business rules that are inserted in Domain Services but that could be implemented on Entities or Value Objects;
- Migrate business rules that do not belong to the Bounded Context of Order Occurrences and Return Occurrences services, to the services that should be responsible to hold those rules.

For possible new business requirements, some of these improvements can be implemented without many efforts. Others might require large efforts, and because of that, it may be not possible to implement. Having these ideas, the team together with the product owner have to estimate and decide what improvements are possible to be implemented, during the development of new requirements.

7.4 Personal Appraisal

Approaching Domain Driven Design made me learn a lot, and think that I presented a development strategy to the team that was really interesting, changing our mindset.

Implementing microservices with Domain Driven Design approach was a great challenge, that generated many discussions about the domain between developers and product owners, as it was expected by DDD, and that made us think and consider the consequences of domain definition and implementation, in our software. In my opinion, everyone understood how important is to have a high-quality domain, and how it spins the rest of the software around it. As a consequence of this project, we are much aware of the business in our applications, and in the next developments we are going to be more comfortable to design and to implement new business requirements.

Domain Driven Design was a lesson learned, since it fits really well in microservices, because different bounded contexts might be implemented on different microservices, having only the business concepts that are really related to it, simplifying the domain design and its implementation. Furthermore, the definition of a Ubiquitous Language was crucial, because now everyone speaks the same names for the business concepts, and these names are written in the code, improving the domain understanding on code from the developers and product owners.

Having this lesson, it is necessary to keep working, in order to have more experience with this strategy, so that Bounded Context delineation, Ubiquitous Language definition, and domain modeling, occur in a more natural way.

I am proud to achieve the objective of this project, even though the result was not perfect, having some improvements to do, but it was a good result for a mindset change like this, with a team, in general, with little practical experience about DDD. This was an opportunity to grow in personal and professional terms, improving my way of thinking and also communication capabilities.

The interest of the team in implementing this strategy was very important to its success. Without their support, the development of microservices with Domain Driven Design would be almost impossible. I really feel that this approach united the team elements with each other, as well as the product owner.

In short, I think this project was a very positive contribution not only for me but also for the whole team.

References

AccountingTools (2017) *Sales return*, *Accounting Tools*. Available at: https://www.accountingtools.com/articles/2017/5/16/sales-return.

AllJewelrybrands (2018) *Order Processing*. Available at: http://alljewelrybrands.com/services/order-processing/.

Avgeriou, P. (2005) '[Avgeriou05]Architectural Patterns Revisited – A Pattern Language.pdf', pp. 1–39.

Bogard, J. (2019) *MediatR*, *MediatR*. Available at: https://github.com/jbogard/MediatR.

Cunningham, W. (1992) *The WyCash Portfolio Management System*, *OOPSLA*. Edited by ACM. Vancouver, Canada. doi: 10.1145/157710.157715.

Dragoni, N. *et al.* (2017) 'Microservices: Yesterday, today, and tomorrow', *Present and Ulterior Software Engineering*, pp. 195–216. doi: 10.1007/978-3-319-67425-4_12.

Evans, E. (2003) *Domain-Driven Design - Tackling Complexity in the Heart of Software*. Alta Books.

Fowler, B. M. *et al.* (2002) *Patterns of Enterprise Application Architecture*. Addison Wesley.

Fowler, M. (2003) *Anemic Domain Model*, *Martin Fowler*. Available at: https://www.martinfowler.com/bliki/AnemicDomainModel.html.

Fowler, M. (2011) *CQRS*, *CQRS*. Available at: https://martinfowler.com/bliki/CQRS.html.

Fowler, M. (2015) *MonolithFirst*, *Martin Fowler*. Available at: https://martinfowler.com/bliki/MonolithFirst.html.

Di Francesco, P. (2017) 'Architecting microservices', *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pp. 224–229. doi: 10.1109/ICSAW.2017.65.

Gamma, E. *et al.* (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 Edition. Addison-Wesley Professional.

Kenton, W. (2018) *Value Proposition*, *Investopedia*. Available at: https://www.investopedia.com/terms/v/valueproposition.asp.

Knoche, H. (2016) 'Sustaining Runtime Performance while Incrementally Modernizing Transactional Monolithic Software towards Microservices', *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*, pp. 121–124. doi: 10.1145/2851553.2892039.

Koen, P. A. *et al.* (2001) 'FuzzyFrontEnd: Effective Methods, Tools, and Techniques LITERATURE REVIEW AND RATIONALE FOR DEVELOPING THE NCD MODEL'. Available at: http://www.stevens-tech.edu/cce/NEW/PDFs/FuzzyFrontEnd_Old.pdf.

Kwiecien, A. (2018) *HOW TO RESCUE ECOMMERCE MONOLITHIC ARCHITECTURE WITH*

MICROSERVICES?, *Divante*. Available at: https://divante.co/blog/how-to-rescue-ecommerce-monolithic-architecture-with-microservices/.

Lazzari, Z. (2018) *Sales Order vs. Sales Invoice*, *Chron*. Available at: https://smallbusiness.chron.com/sales-order-vs-sales-invoice-20610.html.

LH, S. (2019) *SOLID Principles: Explanation and examples*, *ITNEXT*. Available at: https://itnext.io/solid-principles-explanation-and-examples-715b975dcad4 (Accessed: 1 April 2019).

Likert, R. (1932) *A technique for the measurement of attitudes*. Ph. D. Columbia University.

Linford, G. J. (1994) 'Time-resolved xenon flash-lamp opacity measurements.', *Applied optics*, 33(36), pp. 8333–45. doi: 10.1016/0377-2217(90)90057-I.

Martin, R. C. (2000) 'Design principles and design patterns', *Object Mentor*, (c), pp. 1–34. Available at: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf%5Cnhttp://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf.

Martin, R. C. (2017) *Clean Architecture*. Edited by P. Hall.

Microsoft (2013) *Dependency Injection*, *Dependency Injection*. Available at: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-dependency-injection.

Microsoft (2018a) *Design a microservice domain model*, *Microsoft*. Available at: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/microservice-domain-model.

Microsoft (2018b) *Domain events: design and implementation*, *Domain events: design and implementation*. Available at: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/domain-events-design-implementation.

Microsoft (2018c) *Using a Three-Tier Architecture Model*, *Microsoft*. Available at: https://docs.microsoft.com/en-us/windows/desktop/cossdk/using-a-three-tier-architecture-model.

Microsoft (2019a) *Apply CQRS and CQS approaches in a DDD microservice in eShopOnContainers*, *Microsoft*. Available at: https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/microservice-ddd-cqrs-patterns/eshoponcontainers-cqrs-ddd-microservice.

Microsoft (2019b) *Introduction to ASP.NET Core*, *Introduction to ASP.NET Core*. Available at: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2.

Mikkonen, T. (2012) 'Current Trends in Web Engineering', 7059(February). doi: 10.1007/978-3-642-27997-3.

Mota, I. (2018) *Anaemic Domain Model vs. Rich Domain Model*, *Amido*. Available at: https://amido.com/blog/anaemic-domain-model-vs-rich-domain-model/.

Oliphant, E. (2018) *MICROSERVICES: REVOLUTION OR EVOLUTION?*, *Dovel Technologies*.

Available at: https://doveltech.com/innovation/microservices-revolution-evolution/.

Osterwalder, A. (2010) *Business Model Generation*. Edited by J. W. A. S. LTD.

Ozkaya, I., Nord, R. and Kruchten, P. (2012) ‘Technical Debt: From Metaphor to Theory and Practice’, in *IEEE Software*. IEEE Computer Society, pp. 18–21.

Reviews, C. and Kotler, P. (2017) *Just The Facts101 E-Study Guide For: Principles of Marketing*. 15th edn. Edited by Cram101.

Richardson, C. (2018a) *Microservice Architecture*, *Microservices.io*. Available at: https://microservices.io/patterns/microservices.html.

Richardson, C. (2018b) *Monolithic Architecture*, *Microservices.io*. Available at: https://microservices.io/patterns/monolithic.html.

Rouse, M. (2018) *e-commerce (electronic commerce or EC)*, *Search Cio*. Available at: https://searchcio.techtarget.com/definition/e-commerce.

Ruiz, J. (2018) *DDD – Part 4: Life cycle patterns*, *Comment-It*. Available at: https://comment-it.co.uk/ddd-part-4-life-cycle-patterns/.

Saaty, T. L. (2012) *Decision Making for Leaders: The Analytic Hierarchy Process for Decisions in a Complex World*. 3rd Revise. Pittsburgh: RWS Publications.

Shvets, A. (2018) *Dive Into Design Patterns*. Edited by Andrew Wetmore. Refactoring.Guru.

Singh, S. (2013) *Value Analysis: Meaning, Types and Procedure*, *Accounting Notes*. Available at: http://www.accountingnotes.net/cost-accounting/value-analysis/value-analysis-meaning-types-and-procedure/6427.

Sterling, C. (2010) *Managing Software Debt: Building for Inevitable Change (paperback) (Agile Software Development)*. Addison-Wesley Professional.

Terreno, A. (2013) *Microservices and SOLID principles of Object Oriented Design*, *The Arm*. Available at: https://the-arm.com/microservices-and-solid-principles-of-object-oriented-design-fab2e0a6a2c7.

Vallee, K. (2015) *The Difference Between One-Tailed & Two-Tailed Testing*, *Oracle*. Available at: https://blogs.oracle.com/marketingcloud/the-difference-between-one-tailed-two-tailed-testing.

Vich, P. (2018) *Rich Domain Model with DDD/TDD (Reviewed)*, *Paulo Vich*. Available at: https://paulovich.net/2018/07/29/rich-domain-model-with-ddd-tdd-reviewed/.

Wasson, M., Buck, A. and Celarier, S. (2018) *Microservices architecture style*, *Microsoft*. Available at: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices.