INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

Tackling Enterprise Software Development using Domain-Driven Design and Patterns

Marcin Włodarczyk - a40875

Work developed under the guidance of:

Prof. Paulo Jorge Teixeira Matos

Master in Informatics

2023-2024

Tackling Enterprise Software Development using Domain-Driven Design and Patterns

Final Dissertation submitted to Instituto Politécnico de Bragança to obtain the Master Degree in Informatics

Marcin Włodarczyk - a40875

2023-2024

A Escola Superior de Tecnologia e de Gestão não se responsabiliza pelas opiniões expressas neste relatório.

Declaro que o trabalho descrito neste relatório é da minha autoria e é da minha vontade que o mesmo seja submetido a avaliação.

Marcin Włodarczyk - a40875

Acknowledgments

This degree is a crowning achievement of my academic and professional career. Throughout this journey, several important people have stood by my side, and I would like to express my immense gratitude to:

My Supervisor, Prof. Paulo Matos, for his valuable time, patience, and guidance.

My beloved wife, Amanda, for her wholehearted support.

My mother, Donata, for her love and encouragement.

My best friend, Mohamed, for enduring friendship and presence.

Thank you!

Abstract

Professional software development shows that applying the appropriate architecture and programming patterns is the key to successful project delivery. Many technical teams must face the challenge of understanding complex software requirements and turning them into fully functional and scalable software products. It is not uncommon that complicated business rules and constant changes lead to poor product or unmet delivery deadlines. The usual cause of such issues is inadequate software architecture and an unmanageable codebase. In this document, the goal is to explore some technologies and methodologies for the presented problems such as Domain-Driven Design and appropriate programming patterns that accelerate the development process.

Keywords: enterprise, software architecture, programming patterns, DDD

Resumo

O desenvolvimento profissional de software demonstra que a aplicação da arquitetura e dos padrões de programação adequados é a chave para o sucesso na entrega de projetos. Muitas equipas técnicas enfrentam o desafio de compreender requisitos complexos de software e transformá-los em produtos de software totalmente funcionais e escaláveis. Não é incomum que regras de negócio complicadas e mudanças constantes resultem em produtos de baixa qualidade ou no incumprimento de prazos de entrega. A causa habitual desses problemas é uma arquitetura de software inadequada e uma base de código difícil de gerir. Neste documento, o objetivo é explorar algumas tecnologias e metodologias para os problemas apresentados, como o Design Orientado por Domínios (Domain-Driven Design) e padrões de programação adequados que aceleram o processo de desenvolvimento.

Palavras-chave: Empresa, arquitetura de software, padrões de programação, DDD

Contents

Listings

Acronyms

2FA Two-Factor Authentication. 19

API Application Programming Interface. 2, 4, 39, 41, 61

B2B Business-to-Business. 42

BIMI Brand Indicators for Message Identification. 14

CLI Command-Line Interface. 44

CQRS Command Query Responsibility Segregation. 32, 64

CRUD Create, Read, Update, and Delete. 3, 4

DAL Data Access Layer. 12

DDD Domain-Driven Design. 5, 6, 8, 11, 12, 34, 43, 52

DIP Dependency Inversion Principle. 37, 42

DRY Don't Repeat Yourself. 19

DTO Data Transfer Object. 35, 37

HTTP Hypertext Transfer Protocol. 35, 41, 42, 47

I/O Input/Output. 42, 61

IDE Integrated Development Environment. 17

MVC Model View Controller. 3

MVP Minimum Viable Product. 2

OCP Open-Closed Principle. 42

OOD Object-Oriented Design. 37

OOP Object-Oriented Programming. 3, 13

ORM Object-Relational Mapping. 32, 33

OTP One-Time Password. 19

REST Representational State Transfer. 39, 41, 44, 61

TDD Test-Driven Development. 18, 25, 58

UI User Interface. 32, 44

UUID Universally Unique Identifier. 23

UX User Experience. 32

VO Value Object. 12, 15, 18–20

Chapter 1

Introduction

1.1 Background

Nowadays, in the digital era, software can be encountered nearly everywhere. End users expect software to be performant, highly interactive, intuitive, and bug-free. These are only a few characteristics that describe a successful software product. In the highly competitive enterprise environment, developers face complex software architecture and development challenges as part of their daily routine, making numerous micro-decisions that subsequently contribute to the final product's success.

Taking a course of action that results in the general "*correct*" decision-making process is challenging and often only attributed to more senior software engineers. Meanwhile, junior engineers are frequently left alone and, without proper supervision, inadvertently make wrong decisions, resulting in poorly designed software that might not stand the test of time and ultimately fail.

Learning proper software architecture is a continuous process that must become a habit of every successful software engineer. However, finding suitable materials is somewhat tricky. Engineers encounter themselves overwhelmed with an enormous number of libraries, frameworks, and frequently changing tools. However, available learning materials are usually superficial tutorials explaining how to use a particular framework, not

addressing how to design and develop maintainable software correctly.

Insufficient emphasis and to some extent, lack of universally recognized paradigms in the software development industry stands as a distinct challenge. In other respected professions like doctors or lawyers exist well-established paradigms, ethical principles and best practices that serve as a guiding framework, forming a solid foundation for decision-making. Doctors adhere to medical ethics, lawyers obey legal principles, and these paradigms contribute to the reliability, consistency, and quality of their work.

Software engineers operate in a decentralized manner, facing a multitude of frameworks and tools, each advocating different approaches to problem-solving. This lack of common ground leads to a fragmentation of practices, making it challenging to establish a shared set of principles guiding software development decision-making.

1.2 Problem Statement

Over more than ten years of hands-on working experience in different enterprises, leading various software projects, I observed a particular problem that I want to address in this work.

In the context of a software house company, developers are frequently assigned to a project where they must solve a particular problem of the company's client. The term "*problem*" is very generic, and in this context is especially appropriate. There are various clients; some might have a simple problem, such as the demand for a website backed by a simple web Application Programming Interface (API), and other clients have very complex problems, such as the need for software to manage their container shipping across the globe.

Nowadays, Agile software development is a very popular methodology. The main idea of starting with a basic functional Minimum Viable Product (MVP) and, over time, through small iterations, adding new features is great, but must be accompanied by appropriate software architecture. Otherwise, with each release, not only will the number of features increase but also the effort required to introduce or adapt them.

Unfortunately, the typical "*academic*" learning curve lacks many important software architecture-related topics. Students usually tend to focus on learning a particular programming language, technology, or a new framework. Often, Object-Oriented Programming (OOP) concepts gradually fade out, since knowledge not supported by practical application diminishes.

Consequently, new developers approaching software development, especially when dealing with a complex enterprise problem without a well-defined architecture, armed with Create, Read, Update, and Delete (CRUD), Model View Controller (MVC), and a few frameworks in their toolbox, usually end up with "***Anemic Domain Model***" an anti-pattern first described by Martin Fowler:

> "The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together." [2]

As well as Vaughn Vernon in his book Implementing Domain-Driven Design:

> "Anemic Domain Models appear everywhere in our industry every day. The problem is that most developers seem to think this is completely normal and do not recognize that there is a serious side effect when employed on their systems. It's a real problem." [12]

A model with a consistent layered architecture, methods expressing intentions, validation encapsulated in entities, and business language expressed in code is referred to as a "***Rich Domain Model***".

To better illustrate this concept, let us consider the following partial code snippet of an imaginary web API:

```
app.post('/users/create', (req, res) => {
    const {name, email} = req.body;
    if (name.length > 5 && email.contains('@')) {
        dbConnection.insert('user', {name, email});
    } else {
        // error
    }
});
app.put('/users/:id/update', (req, res) => {
    const {name, email} = req.body;
    if (name.length > 5 && email.contains('@')) {
        dbConnection.update('user', {name, email}, {where: {id:
           req.params.id}});
    } else {
        // error
    }
});
```

Listing 1.1: Anemic Web API

It is a typical example of a CRUD-based system where validation and data manipulation happen entirely within a request handler function. This approach indeed works, and a lot of projects start like this. However, this strategy fundamentally has several problems.

Firstly, the user input validation logic must be implemented in every function modifying a user. While it might not look like a big issue in this simple example, it can quickly get out of control when more functions operating on the user model are introduced. Also, if developers add new properties to a `User`, validators must be implemented as well, meaning it is necessary to trace back every function modifying `User` and ensure

adequate validation. Such a problem occurred because, at its inception, the software did not have any sort of "*domain*" layer implemented. While in Domain-Driven Design (DDD), invariants validation through encapsulation is one example of how software can benefit from a rich domain model.

The following chart[11] represents the hypothetical required effort over time necessary to evolve the software:

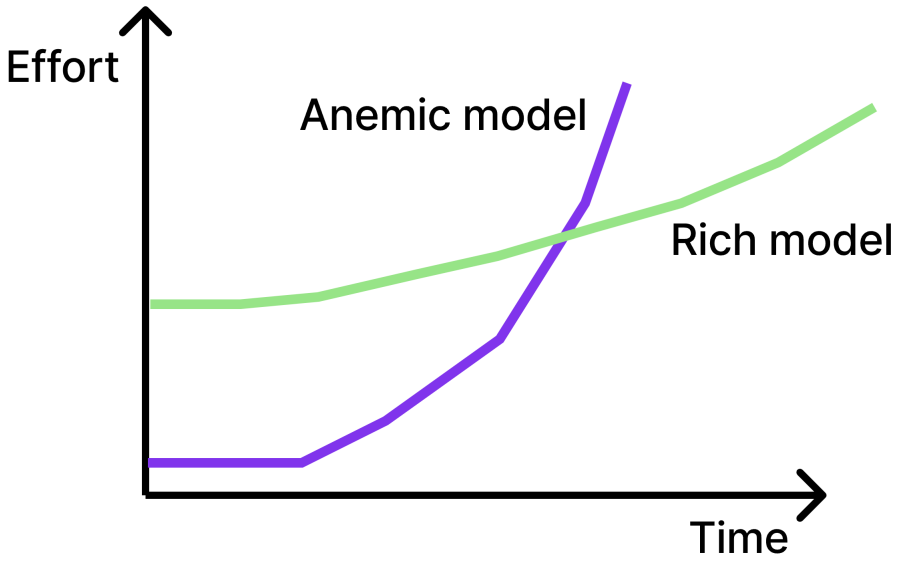

Figure 1.1: Effort Over Time

The "***Anemic Domain Model***" is the most straightforward approach, allowing features to be released quickly. Yet, over time, it becomes more and more difficult to change software or add new features without breaking the rest of it, and at some point, it might even stop being worth maintaining the current product.

In contrast, starting with a "***Rich Domain Model***" requires much more effort right at the beginning. For instance, a simple feature such as user creation could require entity and value object definitions, unit testing, and appropriate persistence via a repository. However, over time, the software can properly evolve; adding or modifying features requires less effort, and such changes can be introduced confidently with correct unit testing implemented.

The inspiration for this work is mainly the book "*Domain-Driven Design: Tackling Complexity in the Heart of Software*" by Eric Evans [1], where he introduced the concept of DDD, and also the work of Robert C. Martin, colloquially called "*Uncle Bob*", an American software engineer who, through his books, articles, and speeches, greatly promotes good software design principles.

This work primarily aims to focus on the problem of the "***Anemic Domain Model***" and how it can be mitigated by applying selected DDD building blocks and popularly recognized software design patterns.

1.3 Document Organization

This document is organized into five chapters:

Chapter 1 sets the context for the research, highlighting challenges related to enterprise software development. It emphasizes the problem of the "***Anemic Domain Model***" and how poor architectural decisions can lead to an unmanageable codebase.

Chapter 2 introduces the concept of a domain and describes how DDD helps to align software architecture with business processes. Real-world examples are used to help understand key differences between various domain types.

Chapter 3 examines in-depth the essential DDD building blocks such as Value Objects, Entities, Aggregates, Repositories, and Use Cases. This chapter explores their properties, applicability, and provides code examples of each building block, showing how they can contribute to the creation of maintainable and scalable software.

Chapter 4 focuses on hands-on experience and provides a practical refactoring walkthrough example of how to turn an "*Anemic*" into a "*Rich*" Domain Model.

Chapter 5 consists of conclusions, summarizing this work, and indicating a path for future research.

Chapter 2

Domain-Driven Design

2.1 Introduction

Delivering high-quality software while preserving a good level of productivity is crucial for success in today's highly competitive market. Unfortunately, many inexperienced teams suffer from declining productivity with each software release. This decline often stems from accumulating technical debt and increasing system complexity. Many developers believe in the common misconception that writing messy code allows for faster development and rapid product release. Although this is not entirely a fallacy, sacrificing clean code leads to several long-term problems that often require drastic refactoring or even rewriting the entire codebase. The short-term gains in speed are quickly overshadowed by the difficulties in maintaining and extending the software. To mitigate these issues, software engineers must embrace Software Architecture and recognize its importance. Clean architecture strives for modular, maintainable, decoupled, and testable software products, where developers can introduce changes with confidence without disrupting the entire system. Clean architecture results in easy unit testing, hassle-free integration of new features, and a robust structure for future development.

2.2 What is Domain-Driven Design?

> " Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain. " [4]

Eric Evans coined the DDD concept in his book published in 2003 [1]. His significant contribution introduces certain standards that should be adopted in the industry. He advocates for the usage of a "*Ubiquitous Language*" - common definitions conveyed throughout the entire development process used by developers and stakeholders. This shared language bridges the gap between technical and business teams, ensuring that everyone has a consistent understanding of domain concepts.

Eric Evans describes the classification of objects into **Entities**, **Value Objects**, and **Service Objects**, also introducing the concept of **Aggregates**. This classification helps to organize the codebase in a way that accurately reflects the real-world domain, facilitating better communication and understanding among team members.

2.3 What is a "Domain"?

Oxford's online dictionary defines a "*domain*" as:

> "1. an area of knowledge or activity; especially one that somebody is responsible for." [9]

In the context of software development, "***activity***" refers to whatever a particular company does, and "***knowledge***" pertains to a company's know-how. Each business possesses distinct, often highly specialized knowledge about its business activities.

A domain is a broad and generic concept. It usually consists of further, more specialized concepts such as:

- **Core Domain**: This is what makes the company unique. The core domain is the essence of the business activity and the reason why the company even exists in the

first place. Usually, all the features belonging to the core domain are implemented from scratch; therefore, it requires the highest level of importance and the best developers. Investing in the core domain provides a competitive advantage and should be the primary focus of development efforts.

- **Supporting Subdomain**: Less valuable to the business than the core domain; however, it is required for the company to succeed and conduct its operations. It is not generic either, meaning that the company cannot replace it by purchasing specific software, as it is somewhat related to the core domain. These subdomains support the core business processes but do not differentiate the company in the marketplace.

- **Generic Subdomain** This does not contain anything specialized and can be easily replaced by existing software solutions (e.g. user authentication, payment processing). Generic subdomains are common across many businesses and do not provide competitive differentiation. Outsourcing or using off-the-shelf solutions for these subdomains allows the company to focus resources on the core domain.

All subdomains are necessary for the entire software product to function. However, they require different levels of effort and attention. Prioritizing development efforts based on the significance of each subdomain ensures that resources are allocated effectively.

2.4 Practical example

To further explore the topic of various kinds of domains let's consider a real-world e-commerce system example identifying the following:

- **Core Domain: Product Recommendation Engine**: To provide the utmost user experience, the recommendation engine must use advanced algorithms and sometimes machine learning to offer personalized product recommendations. Accurately recommending products directly impacts sales, as customers are more likely to purchase additional items. This unique experience distinguishes this particular store from direct competitors. Because the shopping experience and product recommendations directly impact business success, it requires the highest attention and a tailor-made solution.

- **Supporting Subdomain: Inventory Management System** Although it is not the core of the business, the store cannot properly function without tracking stock levels or handling warehouse logistics. This solution must be customized to meet specific business needs.

- **Generic Subdomain: Payment Processing** Payment handling is a generic problem that can be easily outsourced to a third-party service such as Stripe or another payment gateway, therefore requiring the least development effort. As the business's core activity is radically different, it is not the main area of focus and should be outsourced to a business where payments are the core domain.

Chapter 3

Domain-Driven Design Building Blocks

In the software development field, one of the most challenging tasks is to accurately model complex business rules within a software solutions. DDD is a methodology to address these challenges by offering several building blocks facilitating this modeling process. The main presented building blocks include Value Objects, Entities, Aggregates, Repositories, and Use Cases (Application Services). Each of them plays a distinct role in the functional software solution, ensuring that the produced software accurately reflects real-world business logic and meets business needs. Understanding and effectively implementing these building blocks is essential to fully benefit from DDD's potential. By doing so, teams can navigate the complexities of business rules with confidence, swiftly adapt to new requirements, and finally deliver robust software products aligned with business objectives. This chapter dives into each block, exploring its purpose, interrelations, and best practices.

3.1 Value Object

Good software reflects a rich and encapsulated domain model. Achieving this goal means enforcing business invariants using two main primitive concepts: Entities and Value Object (VO)s.

3.1.1 Overview

VO is one of DDD's most basic and important concepts. Also, non-DDD projects can benefit significantly from adopting this building block.

The most primitive concern of software solutions is a widely understood "*validation*". At the very beginning of the project, validation is usually trivial and sometimes not given appropriate attention. It is frequently "*spread all over the place*" by being implemented in various software layers, such as the Data Access Layer (DAL), the database itself, or the infrastructural layer (inside the request handler). When software grows, this quickly leads to an "***Anemic Domain Model***", causing bugs and maintainability issues.

VOs are an excellent solution to these issues, and they can be easily adopted in most projects.

3.1.2 Properties

- **Identity**: At its essence, VO is identificable by its value. This means that two VOs with identical values are considered to be the same VO. Thus, they are interchangeable,
- **Immutability**: Because VO are interchangeable, they should always remain immutable. Instead of modifying the object's internal properties, it should be replaced with a new instance,
- **Coherency**: A VO is not only a wrapper or container of data; it can and should contain business logic. VOs should remain as small and coherent as possible, making them easily maintainable, testable, and reusable across the codebase.

3.1.3 Practical example

Disclaimer:

Throughout this work, various code snippets are shown, written mainly in a programming language of choice - TypeScript; however, the presented concepts can be adopted in any OOP Language. The main purpose of those snippets is to illustrate the given problem and possible solution. Because of its illustrative nature, code is usually incomplete and not production-ready.

Let us consider a simple `User` class containing basic properties such as `email` and `address`.

```
class User {
    email: string;
    address: {
        street: string;
        postalCode: string;
        city: string;
    }
}
```

Listing 3.1: User Class

Usually, the initial approach is to model those using a `String` datatype. Although it is relatively simple and easy to implement, it has several negative drawbacks.

- **Lack of context**: `String` is a very generic datatype; it can contain many different characters such as spaces, numbers, letters or special characters. Whereas an `Email` is a specific domain-related concept. It must not contain spaces, has a limited subset of special characters, and has a predefined standard format. Sometimes, email must adhere to business invariants such as: "*The email addresses of our users must belong to the gmail.com domain.*". In this situation, the trivial task of collecting user email becomes a challenging validation that must be enforced in many places.

- **Unnatural business operations implementation**: As the project progresses, developers will be challenged at some point with the implementation of business operations related to the user's properties. One of the requirements from the stakeholder could sound like this: "*We need to add a new property called nickname to the user. By default, this property should be set to the username part of the email address.*" or "*We need to implement a Brand Indicators for Message Identification (BIMI) retriever that fetches the email provider company's logo based on the user's email domain*". In either case, developers are tasked to operate on email property and by manipulating it to extract some important data. Because email implemented as a `String` is a primitive data type, such operation must be introduced as a standalone function.

- **Validation**: The `User` class is a core domain entity, and its properties will be used across the codebase. If developers choose to implement email validation in the infrastructure layer, for example, in the request handler, it will quickly become tedious to maintain. Considering two endpoints, one responsible for user creation and the other for user update, email validation now lies in two distinct places.

3.1.4 Introducing Email Value Object

The following code snippet shows a possible implementation of the `Email` VO and refactored `User` class.

```
class Email {
    value: string;

    private constructor(value: string) {
        this.value = value;
    }

    public static create(email: string): Email {
        if(!validator.isValidEmailAddress(email)) {
            throw new Error("Provided email address is invalid");
        }

        return new Email(email);
    }
}
```

Listing 3.2: Email Value Object

```
class User {
    email: Email;
    address: {
        street: string;
        postalCode: string;
        city: string;
    }
}
```

Listing 3.3: User with Email

Note on Factory Method Pattern

Usually, the *new* keyword is used to instantiate a class, therefore creating various objects. Unfortunately, using this approach implies implementing entire validation logic inside the constructor, making it, not the most elegant solution. However, object instantiation can be easily streamlined into a single static factory method by marking a constructor as private. This way, object instantiation is limited to a single method that can execute complex logic and even return validation results.

Email Value Object Class

This straightforward refactoring adds several advantages to our project.

Firstly, it brings context; instead of primitive data types such as strings, developers can now use rich domain objects. `Email` class encapsulates related business operations and input validation. Whenever developers operate on the `Email` instance, it is already explicit that the particular object contains a valid email address. Implementing business operations becomes more natural. Instead of dealing with a standalone helper function, the developer can implement relevant operations directly as a method within the `Email`.

For example, let us cover the first business requirement mentioned earlier: "*We need to add a new property called nickname to the user. By default, this property should be set to the username part of the email address.*"

Implementation of this operation could look like this:

```
class Email {
    // ...
    public getUsername(): string {
        return this.value.split('@')[0];
    }
    // ...
}
const email = Email.create('john@example.com');
```

```
9 email.getUsername(); // john
```

Listing 3.4: Nickname Extraction Example

This approach correctly encapsulates logic within the contextual business object: `Email`.

As ValueObjects can be instantiated only with the correct data, the implementation of the *getUsername* method is greatly simplified because the extra validation step is unnecessary; developers can safely assume that the value stored is already a valid email containing the "@" character.

Code becomes more readable and self-documented as modern Integrated Development Environment (IDE)s can render auto-complete suggestions based on the class definition. If a developer needs to know what kind of operations can be executed on a user's email, simple property access is sufficient: *user.email.<autocomplete>*

Unit Testing

Another crucial and sometimes neglected aspect is Unit Testing. The VO pattern encourages Test-Driven Development (TDD), which can be quickly adopted.

```
describe('Email', () => {
    it('Should create a valid email', () => {
        const input = 'john@example.com';
        const e = Email.create(input);
        expect(e.value).toBe(input);
    });

    it('Should return a correct username', () => {
        expect(
            Email
                .create('john@example.com')
                .getUsername()
        ).toBe('john');
    });

    it('Should fail with invalid input', () => {
        expect(
            () => Email.create('john.example.com')
        ).toThrow();
    });
});
```

Listing 3.5: Email Unit Testing

This snippet exemplifies three unit case scenarios: two positives and one negative, where failure is expected. A similar strategy can be adapted to almost any VO, increasing the code quality and contributing to a robust, well-tested codebase.

3.1.5 Code Reusability and Composite Value Objects

> "The beauty of a living thing is not the atoms that go into it, but the way those atoms are put together. — Carl Sagan, Cosmos" [10]

VOs are the smallest building blocks. We can think of them as "*atoms*" of our codebase. However, Value Objects themselves are not a panacea for all enterprise software complexity issues. Nevertheless, their composition and reusability make data validation easier and more manageable.

Value Objects Reusability

The previously presented `Email` object is not limited exclusively to the `User` entity. It can be successfully ported into different contexts, thus increasing the code reusability. Few examples:

- Product stock notifications or newsletter subscriptions require an email address
- Order confirmation and shipping updates usually require email notification
- Two-Factor Authentication (2FA) - a verification One-Time Password (OTP) code or link is sent to the email to complete the authentication process
- Customer support - email is needed to track and respond to customer support tickets and inquiries

Such reusability usually emerges naturally through the project development process and is highly recommendable as it adheres to the Don't Repeat Yourself (DRY) principle. In this particular example, complex validation logic and other email-related operations are implemented and adequately tested only once; various implementations can safely rely on this object. When adjustments are needed, developers have a centralized place where code can be amended and tested, and those changes will automatically propagate to the rest of the codebase.

Value Objects Composition

Sometimes, a single `Value Object` can become overly complex and require decomposition into smaller parts.

Let us focus on a second property of the `User` class called "address", which represents the physical location where a particular user resides. It consists of three sub-properties. The first is a "street" which represents the specific street name, including the building or apartment number and sometimes special characters, for example: "Rua das Flores, 123, 2º Esq.". The second property is a "postalCode", a series of letters and numbers assigned to a specific geographic area. Finally, the third and last property is "city", associated with the given postal code.

```
class User {
    email: Email;
    address: Address;
}

class Address {
    street: Street;
    postalCode: PostalCode;
    city: City;
    // ...
}

class Street { ... }
class PostalCode { ... }
class City { ... }
```

Listing 3.6: Value Objects Composition Example

The example presented above shows how a larger VO such as `Address` can be decomposed into several smaller VOs. Because all sub-parts of the address(*street* , *postalCode* , and *city*) require complex validation, it is not practical to implement this validation at

the `Address` level. Instead, it is more manageable to encapsulate validation in specific classes and import them within the parent `Address` class. An additional advantage is that unit testing becomes easier, as developers can focus their attention and testing code on specific problems such as `Street` validation rather than on the more generic `Address`.

At this point, it is worth stepping back to reevaluate the software design choices. Occasionally, developers tend to make an assumption about a particular property or data structure that might be wrong. Sometimes, a particular real-world concept can be modeled as a `Value Object` in one context and as an `Entity` in another.

In this example, two identical addresses are considered as the same address. Since objects are identifiable by their value, a user address is a perfect candidate to be modelled as a `Value Object`.

3.2 Entity

Following exploration of Value Objects, now let us consider another fundamental building block in Domain-Driven Design: the `Entity`. While Value Objects are defined by their attributes, `Entities` are distinguished by their unique identity, which remains consistent throughout their lifecycle. `Entities` are crucial when a concept in the domain requires an identity that persists beyond its attributes. This persistence allows `Entities` to enforce important business rules and invariants. In the following section, it will be examined how `Entities`, like Orders in an E-commerce domain, manage state and ensure the integrity of business operations.

Eric Evans introduces Entities in his book as:

> "Many objects are not fundamentally defined by their attributes, but rather by a thread of continuity and identity."[1]

This brief sentence encapsulates the most significant characteristics: **continuity** and **identity**. In essence, an `Entity` possesses a lifecycle and a distinct identity that remains consistent regardless of its attributes or behaviors.

3.2.1 Overview

The identity of an `Entity` is determined by a unique ID, which is generated when it is created and remains unchanged throughout its lifecycle. Two Entities of the same type are considered equal only when their IDs are identical.

3.2.2 What are "business invariants"?

An invariant is a rule that must always be upheld, regardless of what actions are attempted within the system. Software architects typically reveal these invariants during requirements elicitation sessions with key stakeholders, often referred to as "*domain experts*"

3.2.3 Entity Properties

- **Identity**: Unlike a Value Object, an `Entity` is not identified by its value but by a unique identifier (often a Universally Unique Identifier (UUID)) assigned at creation. This unique ID persists throughout the `Entity`'s lifespan. Only Entities of the same type should be compared, and they are considered equal only when their IDs are the same.

- **Mutability**: In contrast to Value Objects, Entities are mutable. However, it's best practice to keep internal properties private and only modify them when necessary through specific methods (verbs) that correspond to business operations. This preserves the integrity of the `Entity` while allowing controlled changes in response to business needs.

- **Business Invariant Enforcer**: Similar to Value Objects, an `Entity` is responsible for enforcing business invariants but operates at a higher level. While Value Objects handle low-level value validation, Entities manage the more complex relationships between values or even between other Entities.

3.2.4 High-level Invariants Enforcement Example

Let's consider a practical example from the E-commerce domain to illustrate how Entities enforce business invariants. During requirements elicitation, it was discovered that `Order` is a crucial business concept with the following invariants:

1. An order cannot be fulfilled until payment is received.

2. An order cannot be empty; it must always contain at least one item.

3. An order cannot have a negative total price.

In a real-world application, this list would typically be much longer, often containing complex rules that need to be carefully analyzed and incorporated into the software solution.

Order Entity Example

This minimalist representation of an Order helps to explore the core purpose of an `Entity`. It is clear that many of the properties are not primitive data types but "*richer*" Value Objects. For example, the `Money` class could contain information about both the amount and the currency. Assuming the system only handles positive monetary values, the use of this Value Object inherently enforces the third invariant: an order cannot have a negative total price. The other two rules can be validated through careful `Entity` design:

```
interface OrderProps {
    items: OrderItem[];
    isPaid: boolean;
}

class Order extends Entity<OrderProps> {
    private constructor(props: OrderProps, id: UUID) {
        super(props, id);
    }

    public static create(props: OrderProps): Order {
        if(props.items.length < 1) {
            throw new ValidationError('Order must contain at
                least one item');
        }

        return new Order(props, new UUID());
    }
}
```

Listing 3.7: Order Entity

Enforcing Invariants Through Design

The Order `Entity` internally manages a collection of `Item` objects, which allows enforcing the second invariant: "*an order cannot be empty and must always contain at least one item*". The lifecycle of the Order begins when a new instance is created. Using the factory method pattern, it can be guaranteed that an `Order` can only be instantiated if the Items array contains at least one element. As the Order progresses through its lifecycle, items can be added or removed, but with constraints. For example, a fulfilled `Order` should not be modified, as it becomes immutable at this point. Additionally, removing the last item from the `Order` would break the object's consistency, so this action should be prohibited. These are just a few examples of how Entities enforce business invariants. In real-world systems, the complexity of business rules can be overwhelming. Therefore, it's critical to approach the design pragmatically and systematically. Techniques like TDD provide a safety net, enabling to introduce changes with confidence. Furthermore, employing appropriate software design patterns enhances maintainability and scalability, ensuring the long-term health of the system.

3.3 Aggregate

An `Aggregate` is a cluster of entities sharing the same lifecycle, meaning they are created, retrieved, and stored as a single unit. Aggregates must adhere to business invariants, ensuring they always remain in a consistent state. Conceptually, an Aggregate is owned by a single entity known as the `Aggregate Root`. The ID of this root entity is used to identify the entire `Aggregate`.

Several principles govern good `Aggregate` design:

- **Reference Through Root Only**: Each `Aggregate` must be referenced solely through its root. Internal objects, such as entities, must not reference objects from other `Aggregate`s.
- **Business Invariants Enforcement**: The `Aggregate` Root is responsible for enforcing all business invariants, ensuring it remains in a consistent state.
- **"*Less is More*"**: Since `Aggregates` are always retrieved and modified as a single unit, large `Aggregates` can negatively impact software performance. Software architects must be particularly cautious regarding unbounded one-to-many associations, such as collections (e.g. "*Post has many comments*"). If the collection grows significantly, it can lead to severe performance issues or even system reliability problems. Avoiding unbounded collections is a good practice. Small `Aggregates` are easier to manage and enforce business rules, and the use of immutable Value Objects over mutable entities is recommended.

When designing software entities, the above guidelines should be taken into consideration. Software architects must carefully analyze the business domain and determine whether an entity is a "*local object*" within an `Aggregate` or should be an `Aggregate Root` itself. Designing correct software is a challenging process that often requires profound business understanding, which can only be achieved through close collaboration with key stakeholders. Methods such as *EventStorming*, invented by Alberto Brandolini,

can greatly facilitate this process by enabling teams to visualize complex business processes and uncover domain insights.

Additionally, answering the following questions can help determine the correct role of an entity:

- **How will the entity be accessed in the software?**: If the project requires the development of a dedicated view or UI where the entity is accessed by its ID and its data is displayed, this is a strong indicator that the entity is an `Aggregate Root`.
- **How will the entity be modified?**: If the entity can be modified in isolation, it is likely an `Aggregate Root`. However, if changes to the entity involve other entities (i.e. it cannot be modified without changing another entity), then it is probably a local entity within an `Aggregate`.

3.3.1 Code Example

This section further examines the E-commerce business domain, this time focusing on the concept of an `Aggregate`.

Using a slightly modified invariant as an example: "*An empty order cannot be fulfilled; it should contain at least one item*"

Based on the given business rule, two main entities can be immediately identified: **Order** and **Item**. Because the **Order** holds the high-level validation responsibility and contains a collection of **Items**, it is a proper `Aggregate Root` candidate. The **Item** will become a local `Entity` within the `Aggregate`; thus, it will only have a local identity and should not be referenced by objects located outside of the `Aggregate`.

The design of entities should adequately reflect the real-world domain. This can be achieved through careful relationship design as well as defining entities' behavior using appropriate verbs (methods used to alter state).

In general, the following operations can be performed:

- Create a new empty order
- Create a new order with one or more items
- Remove an item from the order
- Add an item to the order
- Fulfill the order

Since **Order** can now be instantiated with an empty state, there is no need to enforce the "*non-empty*" restriction within the *create* static factory method. Instead, this validation can be delegated to state-altering operations: *addItem*, *removeItem* and *fulfill*.

A simplified implementation could look like this:

```
interface OrderProps {
    items: OrderItem[];
    isPaid: boolean;
}

class Order extends Entity<OrderProps> {
    private constructor(props: OrderProps, id: UUID) {
        super(props, id);
    }

    public addItem(item: OrderItem): void {
        if (this.props.items.length >= 20) {
            throw new ValidationError('Order cannot contain more
                than 20 items');
        }
        this.props.items.push(item);
    }
```

```
17
18      public removeItem(item: OrderItem): void {
19          const index = this.props.items.findIndex((i: OrderItem)
                => i.equals(item));
20          if (index !== -1) {
21              this.props.items.splice(index, 1);
22          }
23      }
24
25      public fulfill(): void {
26          if (this.props.items.length < 1) {
27              throw new ValidationError('Empty order cannot be
                    fulfilled');
28          }
29          // Proceed with fulfillment
30      }
31
32      public static create(props: OrderProps): Order {
33          return new Order(props, new UUID());
34      }
35  }
```

Listing 3.8: Aggregate Root Example

In this example, the `Order` class enforces business invariants within its methods. The *addItem* method prevents adding more than 20 items to an order, adhering to the new business rule: "*An order can contain a maximum of 20 items*". The *fulfill* method ensures that an empty order cannot be fulfilled, as per our original invariant.

If, for instance, another new invariant arises, such as "*An order must be paid before it can be fulfilled*" implementing this requirement is straightforward.

The *fulfill* method can be extended to check the *isPaid* property before proceeding with fulfillment:

```
public fulfill(): void {
    if (this.props.items.length < 1) {
        throw new ValidationError('Empty order cannot be
           fulfilled');
    }
    if (!this.props.isPaid) {
        throw new ValidationError('Order must be paid before
           fulfillment');
    }
    // Proceed with fulfillment
}
```

Listing 3.9: Aggregate Root Example

This approach demonstrates how business rules can be enforced within the `Aggregate Root`, keeping it in a consistent state. By centralizing validations and state changes within the `Aggregate Root` a clear and manageable codebase that accurately reflects the business domain is maintained.

3.4 Repository

Value Objects, Entities, and Aggregates act as sentinels overseeing business rules. These building blocks live at the heart of the software. However, software that is unable to persist data won't be of much use. That is why software engineers need a solution to tackle data persistence challenges.

3.4.1 Definition

Eric Evans defines a `Repository` as:

> "A repository represents all objects of a certain type as a conceptual set (usually emulated). It acts like a collection, except with more elaborate querying capability. [...] For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type." [1]

In short, a `Repository` is a persistent container of Aggregates, meaning that each Aggregate persisted via the `Repository` into the database of choice can be retrieved later (usually via its unique ID).

A basic `Repository` could consist of the following methods:

- **save**: Responsible to persist an entire Aggregate (its internal Entities and Value Objects) into the data storage.
- **findById** Responsible to query data storage and to return correctly instantiated Aggregate.
- **delete** Responsible to permanently delete an Aggregate from the data storage.

These basic methods allow fundamental control over an Aggregate's lifecycle. However, they are frequently not sufficient to fulfill all the system's requirements. Therefore, the addition of more specialized methods is necessary, such as methods to find Aggregates by certain criteria.

3.4.2 Code Example

A correctly implemented `Repository` pattern not only eases the burden of data persistence but also introduces another level of abstraction between application logic and the infrastructure layer where the database resides. A concrete implementation of a `Repository` can represent a particular database or specific technology (Object-Relational Mapping (ORM)) used.

Note on performance: The concept of Repositories persisting and retrieving entire Aggregates can have negative system performance implications, especially when dealing with complex Aggregates consisting of many objects. Considering a simple use case where the User Interface (UI) needs to render a list of Aggregates displaying just a few of their properties. The time wasted on querying and instantiating a list of complex Aggregates can negatively impact the User Experience (UX). A remedy to this issue can be the Command Query Responsibility Segregation (CQRS) [3] pattern, which separates read and update operations. This pattern allows for more efficient data retrieval when only certain properties are necessary. However, CQRS is beyond the scope of this work and won't be presented in detail.

Let us proceed through a practical code example that defines a Sequelize implementation of the `Order Repository`. Sequelize is a modern TypeScript ORM frequently used in NodeJS projects.

```
class SequelizeOrderRepo {
    public async findById(id: UUID): Promise<Order> {
        // ...
    }

    public async save(order: Order): Promise<void> {
        // ...
    }

    public async delete(order: Order): Promise<void> {
        await models.Order.destroy({
            where: {
                id: order.id,
            },
        });
    }
}
```

Listing 3.10: Sequelize Order Repository

The above code snippet exemplifies the *SequelizeOrderRepo* implementation with *findById*, *save*, and *delete* methods, where only the last one contains an actual implementation. In a real-world scenario, each method would include logic to interact with the database, ensuring that Aggregates are correctly persisted and retrieved.

3.5 Use Case

This chapter explores the last DDD building block presented in this work - `Use Case`. Most of the presented blocks closely relate to the domain layer, encapsulating important business logic. `Use Cases`, through orchestration, enable engineers to connect those pieces together in the **Application Layer**.

3.5.1 Definition

`Use Cases`, also known as *Application Services*, are described by Uncle Bob as:

> "The software in this layer contains application-specific business rules. It encapsulates and implements all of 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 enterprise-wide business rules to achieve the goals of the use case."[6]

In essence, a `Use Case` represents a functional piece of the software implemented in the application layer. Uncle Bob highlights that a `Use Case` should contain application-specific business rules, meaning that the focus shifts from domain invariants such as "*An order cannot be fulfilled until payment is received*" to application-specific rules like "*An order must exist to receive a payment*".

These are two very distinct concepts whose implementations reside in different software layers. The `Use Case` facilitates communication between the domain model and the rest of the world. It is responsible for fetching aggregates using repositories, invoking appropriate methods, handling transactions, and persisting changes in external storage.

3.5.2 Use Case Characteristics

- **Stateless**: A `Use Case` should remain stateless, meaning that it does not maintain any internal state between method calls. This design ensures that it is thread-safe and can be easily tested and scaled. Data required to perform operations should be passed through method parameters, often encapsulated in Data Transfer Object (DTO), which facilitate the transfer of data between layers.

- **Security Enforcer** Typically, a `Use Case` is invoked from within the infrastructure layer by an external entity. For example, a controller reacting to a user's Hypertext Transfer Protocol (HTTP) call might invoke the `Use Case`'s method. In this scenario, it is crucial to ensure that the user invoking the action possesses appropriate execute permissions. Implementing security checks within the `Use Case` ensures that business operations are protected regardless of the entry point. A good example is a situation where a regular user should have permission to only cancel and read their own orders, while system administrators, having a broader set of permissions, should be able to list all orders and modify their statuses.

- **Operations Orchestrator** Isolating orchestration from business logic might be an intricate task. Orchestration in this context means retrieving and invoking the correct domain objects in a particular order, inputting appropriate parameters, and returning the expected output. By orchestrating the sequence of operations, the `Use Case` ensures that complex business workflows are executed correctly. A `Use Case` might retrieve single or multiple Aggregates, instantiate Value Objects, interact with Aggregates' methods, persist changes into the Repository, and ultimately return an operation status or error.

3.5.3 Code Example

To complement the presented `Use Case` theoretical concepts, let us consider a practical example based on the following code snippet:

```
class ConfirmOrderPaymentUseCase {
    constructor(
        private orderRepo: IOrderRepo
    ) {}

    public async execute(dto: PaymentConfirmationDto) {
        if(!dto.orderId) {
            throw new MissingParameterError('Order ID must be
                defined');
        }

        const order = await this.orderRepo.getOrder(dto.orderId);
        if(!order) {
            throw new NotFoundError(`Order ${dto.orderId} not
                found`);
        }

        order.paid(dto.paymentId);
        await this.orderRepo.save(order);
    }
}
```

Listing 3.11: ConfirmOrderPaymentUseCase

This minimalistic snippet shows a possible orchestration of a payment confirmation. It is represented as a class that could potentially implement an interface to standardize the execute method, ensuring consistency across different `Use Cases` in the application layer.

The `Use Case` receives a single DTO input object that carries enough data to achieve the end goal, namely *orderId* and *paymentId*. This design allows to establish a clear contract between infrastructural and application layers, allowing to easily introduce more data into the DTO as software evolves.

The application logic orchestration is visible immediately. Firstly, the `Use Case` ensures that the given DTO contains the necessary parameters, validating the input to prevent errors. If the parameters are valid, an Order Aggregate is fetched from the data storage. The *paid* method, mutates Order by changing its status and storing the payment reference. This step encapsulates the domain logic within the Aggregate. Ultimately, the modified Aggregate is persisted back into the repository.

Note on Dependency Inversion Principle (DIP)

In the 2000 paper "*Design Principles and Design Patterns*" Robert C. Martin introduced the five Object-Oriented Design (OOD) principles, which establish good practices for developing maintainable and extensible software. One of those principles is the DIP defined as:

> "Depend upon Abstractions. Do not depend upon concretions. [...] The high level modules deal with the high level policies of the application. These policies generally care little about the details that implement them [...] Every dependency in the design should target an interface, or an abstract class. No dependency should target a concrete class..."[7]

The *ConfirmOrderPaymentUseCase* is a high-level module implementing the logic necessary to confirm an order payment. Importing a concrete implementation of the Repository, such as *MySQLOrderRepo*, would increase tight coupling, resulting in poor maintainability and the necessity to provision infrastructure to perform unit testing.

In the presented example, the `Use Case`'s dependency has been inverted. The high-level module depends on the abstract *IOrderRepo* interface, making it easy to test.

During class instantiation, engineers have the flexibility to decide about the concrete implementation used, which can be replaced without refactoring the `Use Case` logic.

This design allows for different data storage solutions to be plugged in, such as an in-memory database for testing or popular MySQL, PostgreSQL, or MongoDB, enhancing the application's adaptability.

3.6 Summary

This chapter explored the fundamental building blocks of Domain-Driven Design, namely `Value Objects`, `Entities`, `Aggregates`, `Repositories`, and `Use Cases`. Various practical examples included in this chapter demonstrated how each building block contributes to the creation of a rich and well-encapsulated domain model.

The following chapter delves into hands-on experience, showing how the presented theoretical concepts can be incorporated into practical refactoring to tackle complex software challenges with confidence.

Chapter 4

Refactoring Walkthrough

4.1 Anemic Domain Model Refactoring

This chapter provides a practical refactoring walkthrough of a hypothetical Representational State Transfer (REST)ful Web API, particularly focusing on a single feature responsible for order placement and currency exchange.

Disclaimer: The provided example is an incomplete TypeScript solution. Many parts such as error handling, variable extraction/declarations, and some function implementations are deliberately omitted for clarity. All presented examples are meant to primarily exemplify how any software, and especially web APIs, can benefit from refactoring towards a "***Rich Domain Model***".

The following snippet is a starting point:

```
import openExchangeService from 'services/openExchangeService';
import mysqlOrm from 'db/orm';

app.post('/orders', async (req, res) => {
    // ... variables extraction logic
    const {items, customer} = dto;

    // Validate order
    if(items.length < 1) {
        throw new ValidationError('At least one item is
           required');
    }
    // Base item price is in EUR
    const totalEUR = items.reduce((acc, item) => acc +
       item.price, 0);

    // Exchange to customer currency
    const rate = await openExchangeService.getRate({base: 'EUR',
       quote: customer.currency});
    const totalPrice = totalEUR * rate;

    // Persist order
    await mysqlOrm.Order.create({
        customerId: customer.id,
        totalPrice,
        items,
    });
});
```

Listing 4.1: Refactoring Starting Point

4.2 Example Overview

After a brief code analysis (and provided comments), it can be noticed that the main element is a request-handling function, invoked by an HTTP POST request to the "*/orders*" path. This function simulates an order placement request. First, necessary variables are extracted from the request. Assuming that connecting clients send a list of items, and their prices as well as a customer object is appended beforehand by some middleware (this mechanism is not relevant for this particular example).

Subsequently, a list of items is reduced into a single number representing a total order price in EUR currency. In this example, it is considered that each item object contains its unitary price represented as a number in EUR currency.

Next, an async method called *getRate* is invoked on the concrete implementation of Open Exchange Rates Service, which is a popular REST API to retrieve currency exchange data.

Lastly, the total order price is exchanged for the customer's currency, and ultimately an object representing a single order is inserted into the MySQL database.

4.3 Revealing Anemic Model

The presented software design has several problems which will be addressed later in the refactoring steps:

- **Lack of layered architecture** Infrastructure-related concepts such as HTTP request handling or a database connection are part of the application logic orchestration and domain invariants validation. Also, a change of the chosen web framework or underlying database engine will require refactoring of the entire app.

- **Tight coupling** The function responsible for placing orders is tightly coupled to the Open Exchange Rates Service as well as to the database connection, and such design violates the DIP. A simple decision, like replacing the exchange rate provider, requires refactoring of the order placement function. This kind of issue relates to the Open-Closed Principle (OCP); when a single addition requires many changes, the OCP is usually violated. Another problem created by tight coupling and dependency on concrete implementations is related to unit testing. Good software consists of many lightweight tests which can be easily implemented and executed. In this particular case, the implementation of tests requires a live database connection as well as a concrete implementation of an exchange rate provider, and probably even a live HTTP server. This significantly increases the complexity of testing and decreases test performance as real Input/Output (I/O) operations must be executed.

- **Leaking domain logic** Validation of items is implemented right in the request handler function; this issue is also referred to as "*leaking domain logic*". Firstly, if another endpoint is to place an order (maybe via some Business-to-Business (B2B) integration or order subscription mechanism), this validation must be duplicated. If ever a requirement of a minimum number of items changes, a developer must trace down all the occurrences of order items validation and alter each of them separately. This design is prone to mistakes and lacks centralization.

4.4 Refactoring

In the previous section, several software design issues were revealed. This section focuses on refactoring and how DDD building blocks and design patterns can be applied to remedy them.

4.4.1 Step 1 - Separation of Concerns

In this refactoring step, the main goal is to establish a proper layered architecture. There are, in fact, many architectural approaches such as *Hexagonal Architecture* or *Onion Architecture*. These architectural patterns are often very similar, and the main objective is the **separation of concerns**.

Robert C. Martin depicts a Clean Architecture using the following diagram:

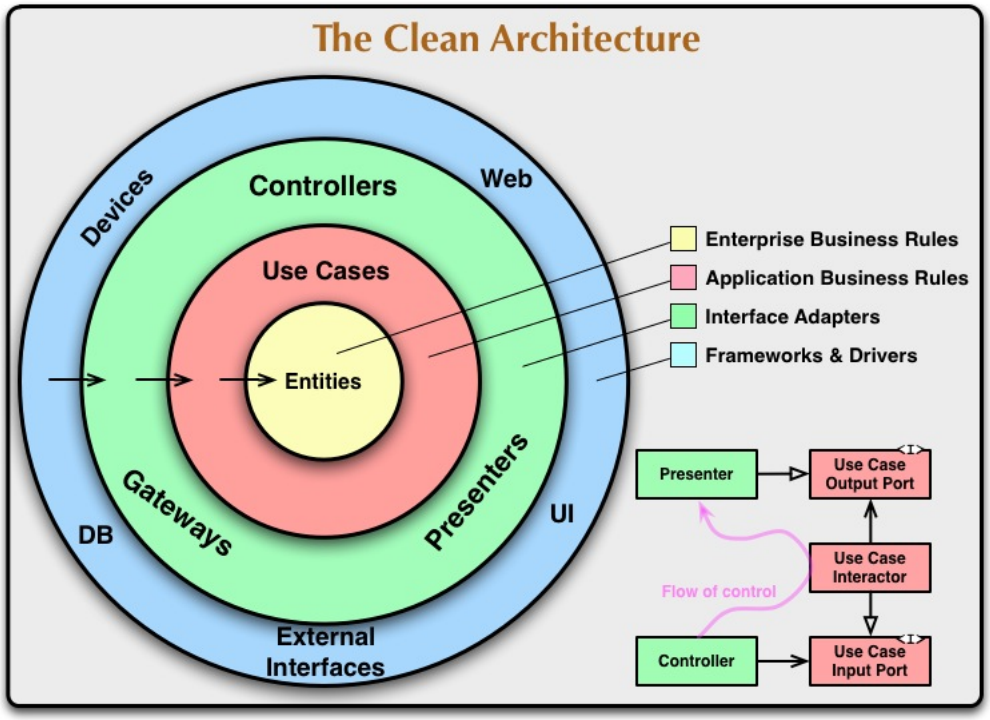

Figure 4.1: Clean Architecture

This diagram represents the actionable idea of producing systems that are[8]:

- **Independent of Frameworks**: It should be relatively easy to switch between frameworks such as Express.js or Deno, or even between REST and GraphQL.
- **Testable**: Business invariants should be testable without the UI, database, web server, or any other external element such as a currency exchange provider.
- **Independent of UI**: The UI can change easily without changing the rest of the system. It can even be replaced by a Command-Line Interface (CLI).
- **Independent of Database**: Just as it should be easy to switch between frameworks, it should also be easy to swap underlying database engines. It should be possible to swap MySQL for Oracle or even MongoDB.
- **Independent of any external agency**: Business rules should not know anything about the outside world.

In the context of order placement refactoring, a good initial step could be to introduce the following layers:

- **Domain**: with `Order` entity (potentially an Aggregate Root)
- **Application**: with `OrderPlacementUseCase` containing application logic orchestration
- **Infrastructure**: with `OrderPlacementController` connected with an external framework such as Express.js

This code snippet represents the first refactoring iteration where comments above the classes indicate potential file organization:

```
// layers/domain/Order.ts
class Order {
    private constructor(items, customerId) {}

    public static create(items, customerId) {
        if (items.length < 1) {
            throw new ValidationError('At least one item is
               required');
        }

        return new Order(items, customerId);
    }
}

// layers/application/OrderPlacementUseCase.ts
import openExchangeService from 'services/openExchangeService';
import mysqlOrm from 'db/orm';

class OrderPlacementUseCase {
    async execute(dto) {
        const {items, customer} = dto;
        const order = Order.create(items, customer.id);

        // Base item price is in EUR
        const totalEUR = items.reduce((acc, item) => acc +
           item.price, 0);

        // Exchange to customer currency
```

```
27          const rate = await openExchangeService.getRate({base:
               'EUR', quote: customer.currency});
28          const totalPrice = totalEUR * rate;
29
30          // Persist order
31          await mysqlOrm.Order.create({
32              customerId: customer.id,
33              totalPrice,
34              items,
35          });
36      }
37  }
38
39  // layers/infra/OrderPlacementController.ts
40  class OrderPlacementController {
41      async execute(req, res) {
42          // ... variables extraction logic
43          await orderPlacementUseCase.execute(dto);
44      }
45  }
46
47  // layers/infra/router.ts
48  app.post('/orders', (req, res) => {
49      return orderPlacementController.execute(req, res);
50  });
```

Listing 4.2: Refactoring - Step 1

A domain layer consists of an Order entity, where the factory pattern is used to enforce business invariants related to item validation. Next, the application layer has a defined Use Case whose name clearly expresses its intention. This class still contains most of the request handler function logic; however, remaining problems will be addressed in

the subsequent refactoring steps. Finally, the infrastructure layer contains a Controller definition which is responsible for handling a request, invoking appropriate Use Case(s), and returning an adequate HTTP response.

4.4.2 Step 2 - Loose Coupling

Dependency Inversion is a pattern that allows components to be decoupled, resulting in loose coupling.

Currently, the flow of dependencies can be represented in the following way:

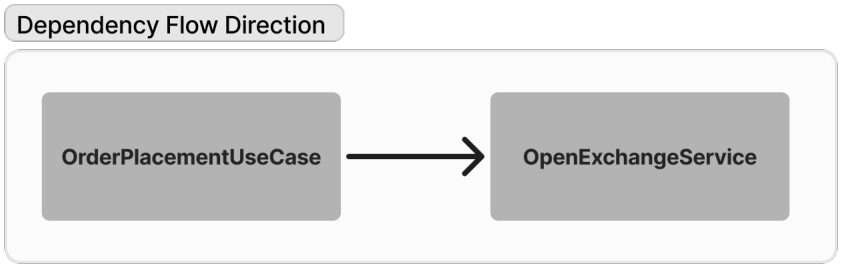

Figure 4.2: Dependency Flow

An *OrderPlacementUseCase* relies on the *OpenExchangeService*. Introducing an *ExchangeService* interface between the two components decouples them and inverts the flow of dependencies.

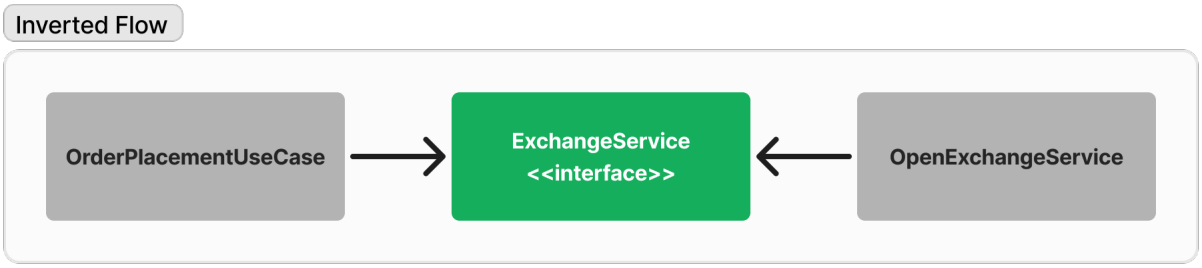

Figure 4.3: Inverted Flow

This results in an architectural boundary between *OrderPlacementUseCase* and *OpenExchangeService*.

Electrical Appliances Analogy

High-level modules such as appliances (lamps or toasters) should not depend on low-level modules such as specific wall sockets. Instead, both should depend on an abstraction, which is the standardized electrical socket interface. This allows them to function with different sockets as long as both (appliances and sockets) conform to the standardized interface. In short, the primary objective is "*plug-and-play*" functionality. Switching between a phone charger and a hair dryer does not require replacing a wall socket, so switching between various exchange rate providers should not require modification of *OrderPlacementUseCase*.

```
class OpenExchangeService implements ExchangeService{
    async getRate(base, quote): Promise<number> {...}
}
class XeExchangeService implements ExchangeService{
    async getRate(base, quote): Promise<number> {...}
}

interface ExchangeService {
    getRate(base: string, quote: string): Promise<number>;
}

// layers/application/OrderPlacementUseCase.ts
import mysqlOrm from 'db/orm';

class OrderPlacementUseCase {
    constructor(
        private exchangeService: ExchangeService,
    ) {}

    async execute(dto) {
        const {items, customer} = dto;
        const order = Order.create(items, customer.id);

        // Base item price is in EUR
        const totalEUR = items.reduce((acc, item) => acc +
            item.price, 0);

        // Exchange to customer currency
        const rate = await this.exchangeService.getRate('EUR',
            customer.currency);
```

```
29          const totalPrice = totalEUR * rate;
30
31          // Persist order
32          await mysqlOrm.Order.create({
33              customerId: customer.id,
34              totalPrice,
35              items,
36          });
37      }
38  }
39
40  const openExchangeService = new OpenExchangeService();
41  const xeExchangeService = new XeExchangeService();
42
43  const useCase = new OrderPlacementUseCase(openExchangeService);
```

Listing 4.3: Refactoring - Step 2

This second refactoring snippet depicts how Dependency Inversion allows developers to easily switch between Open Exchange Rates and XE providers.

4.4.3 Step 3 - Repository Pattern

Chapter 3 explores fundamental DDD building blocks, including the Repository pattern. In this refactoring step, a practical application of the Repository will be presented, demonstrating how this layer of abstraction decouples application logic from the underlying persistence storage.

Currently, a concrete database connection is imported within the *OrderPlacementUseCase*. The main drawback of this approach was already addressed in a previous refactoring step: "*high-level modules should depend on abstractions*". Thus, the next natural step is to introduce a Repository interface and invert the flow of dependencies.

A repository should hide the complexity of the Order persistence mechanism and, through simple methods, establish a contract unrelated to a specific database engine. At this point, the goal is to simply save the Order and retrieve it later if needed. This operation is abstract and can be implemented using MongoDB, MySQL, Oracle, or even a filesystem database.

The following snippet demonstrates how a Repository, together with Dependency Inversion, abstracts the underlying persistence solution:

```
class MongoDBOrderRepo implements OrderRepo{
    async save(order): Promise<void> {...}
}

class MySQLOrderRepo implements OrderRepo{
    async save(order): Promise<void> {...}
}

interface OrderRepo {
    save(order: Order): Promise<void>;
}

// layers/application/OrderPlacementUseCase.ts
```

```
class OrderPlacementUseCase {
    constructor(
        private exchangeService: ExchangeService,
        private orderRepo: OrderRepo,
    ) {}

    async execute(dto) {
        const {items, customer} = dto;
        const order = Order.create(items, customer.id);

        // Base item price is in EUR
        const totalEUR = items.reduce((acc, item) => acc +
            item.price, 0);

        // Exchange to customer currency
        const rate = await this.exchangeService.getRate('EUR',
            customer.currency);
        const totalPrice = totalEUR * rate;

        // Persist order
        await this.orderRepo.save(order);
    }
}

const mysqlRepo = new MySQLOrderRepo();
const mongoRepo = new MongoDBOrderRepo();

const useCase = new OrderPlacementUseCase(xeExchangeService,
   mysqlRepo);
```

Listing 4.4: Refactoring - Step 3

Similar to the *ExchangeService*, the *OrderRepo* is injected into the *OrderPlacementUseCase*, and the *save* method call effectively replaces MySQL-specific INSERT operation.

In the next step, the remaining business logic will be encapsulated within the Order entity.

4.4.4 Step 4 - Business Logic Encapsulation

Although a rich domain model is emerging, the domain logic is still leaking into the use case. A significant portion of the main feature relates to monetary-related operations; the system must constantly handle money, various currencies, and their exchange. This is not immediately noticable because the concept of "Money" is not incorporated into the domain layer and the vocabulary used within the software.

Let's introduce a new Value Object that represents monetary value:

```
1  // layers/domain/Money.ts
2  class Money {
3      private constructor(
4          public readonly amount: number,
5          public readonly currency: string,
6      ) {}
7
8      public exchangeTo(currency: string, rate: number): Money {
9          if(this.currency === currency) {
10             // Example of extra validation: Error or possibly
                  return of the same amount
11             throw new Error('Cannot exchange to the same
                  currency');
12         }
13
14         return Money.create(this.amount * rate, currency);
15     }
```

```
16
17      public static create(amount: number, currency: string):
            Money {
18          // ... Validation logic
19
20          return new Money(amount, currency);
21      }
22  }
```

Listing 4.5: Refactoring - Step 4 (Money)

This Value Object not only ensures that all money-related invariants are validated but also enriches the software itself; developers can now operate on an intuitive object, and the addition of the *exchangeTo* method further facilitates application logic orchestration.

The next step is to address the domain logic related to items and total order price calculation. An Item certainly deserves to become an Entity managed by the Aggregate Root - Order. In this example, a simplified implementation of Order is presented:

```
1   // layers/domain/Order.ts
2   class Order {
3       totalPrice: Money;
4
5       public requiresExchange(customerCurrency: string): boolean {
6           return this.totalPrice.currency !== customerCurrency;
7       }
8
9       public exchangeTotalPrice(currency: string, rate: number) {
10          this.totalPrice = this.totalPrice.exchangeTo(currency,
                rate);
11      }
12
13      public static create(items, customerId): Order {
```

```
14          const total = items.reduce((acc, item) => acc +
                item.price, 0);
15
16          return new Order(...);
17      }
18  }
```

Listing 4.6: Refactoring - Step 4 (Order)

Immediately, it is noticeable that the `Money` Value Object naturally finds its place within the Aggregate representing *totalPrice*, and the remaining business operations are reflected through appropriate methods (verbs).

Finally, at the highest abstraction layer, a refactored `OrderPlacementUseCase` is not only more readable but has become a true application logic orchestrator:

```
// useCases/order/OrderPlacementUseCase.ts
class OrderPlacementUseCase {
    constructor(
        private exchangeService: ExchangeService,
        private orderRepo: OrderRepo,
    ) {}

    async execute(dto) {
        const {items, customer} = dto;
        const order = Order.create(items, customer.id);

        // Exchange to customer currency
        if(order.requiresExchange(customer.currency)) {
            const rate = await
               this.exchangeService.getRate('EUR',
               customer.currency);
            order.exchangeTotalPrice(customer.currency, rate);
        }

        // Persist order
        await this.orderRepo.save(order);
    }
}
```

Listing 4.7: Refactoring - Step 4 (Use Case)

4.4.5 Step 5 - Unit Testing

The final step is not code refactoring but code addition to improve software quality. Unit tests are a vital part of any robust solution.

The domain layer lives at the core of the software and should not depend on the higher layers. If this rule is correctly implemented, domain objects do not have external dependencies and should be easily testable.

> "Test-Driven Development (TDD) is a technique for building software that guides software development by writing tests. It was developed by Kent Beck in the late 1990's as part of Extreme Programming." [5]

TDD is a good programming practice that results in "*Self-Testing Code*" and forces developers to first think about the interface to the code. Thinking about actual class usage and how exposed methods interact with other modules helps to separate the interface from the implementation.

The following snippet shows an example of a Money Value Object test:

```
describe('Money', () => {
    it('Should convert EUR to USD', () => {
        const m1 = Money.create(5, 'EUR');
        const m2 = Money.exchangeTo('USD', 1.25);
        expect(m2.amount).toBe(6.25);
    })
});
```

Listing 4.8: Refactoring - Step 5 (Unit Tests)

In this particular case, the test focuses on currency exchange functionality. Given €5 and a EUR/USD rate of 1.25, the exchange outcome should be $6.25. It is worth noting that one of the Value Object's characteristics is immutability. Instead of modifying the internal state of the first object, a new instance is returned. This allows sharing of value objects safely.

This next example of a unit test is more complex and depicts how an entire use case can be tested and its application orchestration logic:

```
class MockRepo implements OrderRepo {
    public getLastSavedOrder(): Order {...}

    async save(order): Promise<void> {...}
}
class MockExchangeService implements ExchangeService{
    async getRate(base, quote): Promise<number> {...}
}

describe('OrderPlacementUseCase', () => {
    it('Should place order', async () => {
        const useCase = new
           OrderPlacementUseCase(mockExchangeService, mockRepo);

        const customer = {id: 1, currency: 'PLN'};

        await useCase.execute({items, customer});

        expect(mockRepo.orders).toHaveLength(1);
        expect(mockRepo.getLastSavedOrder().totalPrice.currency)
            .toBe(customer.currency);
        expect(mockRepo.getLastSavedOrder().totalPrice.amount)
            .toBe(expectedAmount);

        // ...
    })
});
```

Listing 4.9: Refactoring - Step 5 (Unit Tests)

When a new order is placed, it is expected that the number of orders within the repo increases and that the persisted order matches what was requested. In this example, in-memory Repo and ExchangeService implementations are used. This not only allows the addition of convenience methods such as *getLastSavedOrder*, but also increases test execution speed since use case execution is not bound to real I/O.

4.5 Summary

This chapter showed a practical refactoring and the transformation of an "***Anemic Domain Model***" into a "***Rich Domain Model***".

Starting with a hypothetical order placement RESTful Web API, key issues such as lack of layered architecture, tight coupling, and leaking domain logic were identified.

Through a series of refactoring steps, those problems were addressed and fixed. This refactoring process not only improved the software design but also demonstrated how complex challenges can be tackled with different DDD building blocks.

The next and final chapter contains the conclusion and insights gained throughout this work.

Chapter 5

Conclusion

In my professional career working as a Lead Backend Engineer, I faced numerous complex technical challenges and led several projects from distinct business domains.

This practical experience shows that simply accomplishing functional requirements is usually not enough. Most large projects must not only deliver the necessary features but also adhere to high-quality standards, where functionality, maintainability, and software longevity are fundamental attributes of a successful product.

This work aims to reflect on those challenges that I have encountered and provide a set of tools and methodologies used to tackle them. Each project is different and requires different considerations and tooling; however, the selected concepts presented in this work are mostly versatile and can be adapted to various use cases.

In this work, Domain-Driven Design and recognized programming patterns were comprehensively explored as a robust approach to creating modular, scalable, and maintainable software.

Practical examples and literature references have demonstrated that the "***Anemic Domain Model***" can become a significant obstacle in complex software projects, leading to counterproductive outcomes within a team.

A more complete refactoring example presented in Chapter 4 demonstrated that implementing a "***Rich Domain Model***" by using selected building blocks and appropriate programming techniques can not only enhance code readability but also, through effective

encapsulation of invariants, increase software longevity.

Ultimately, this work presented many complex concepts and, due to its limited scope, barely scratched the surface of some of them. Looking ahead, a good continuation of this work would be the inclusion of more advanced techniques such as Event Sourcing or CQRS, as well as leveraging emerging DevOps strategies and cloud environments, which can further boost the team's productivity.

Bibliography

[1] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. ISBN: 978-0321125217. Addison-Wesley Professional, 2003.

[2] Martin Fowler. *Anemic Domain Model*. 2003. URL: https://martinfowler.com/bliki/AnemicDomainModel.html.

[3] Martin Fowler. *CQRS*. 2011. URL: https://martinfowler.com/bliki/CQRS.html.

[4] Martin Fowler. *Domain Driven Design*. 2020. URL: https://martinfowler.com/bliki/DomainDrivenDesign.html.

[5] Martin Fowler. *Test Driven Development*. 2023. URL: https://martinfowler.com/bliki/TestDrivenDevelopment.html.

[6] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. ISBN: 978-0134494166. Pearson, 2017.

[7] Robert C. Martin. *Design Principles and Design Patterns*. 2000. URL: https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf.

[8] Robert C. Martin. *The Clean Architecture*. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html.

[9] Oxford University Press. *Definition of domain noun from the Oxford Advanced Learner's Dictionary*. URL: https://www.oxfordlearnersdictionaries.com/definition/english/domain.

[10] Carl Sagan. *Cosmos*. ISBN: 978-0-345-53943-4. New York, NY: Random House, 1980.

[11] Khalil Stemmler. *Anemic Domain Model*. URL: `https://khalilstemmler.com/img/wiki/anemic/chart.svg`.

[12] Vaughn Vernon. *Implementing Domain-Driven Design*. ISBN: 978-0321834577. Addison-Wesley Professional, 2013.