UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Daichi Ando

# An approach for Designing Microservice-Based Applications using a Domain-Driven Design Approach and Clean Architecture Principles

Master's Thesis (30 ECTS)

Supervisor(s): Mohamad Gharib

Tartu 2023

I have written this master's thesis independently. All attitudes of other authors, literary sources, and data from elsewhere used for writing this paper have been referenced.

**Acknowledgment:**
I wish to extend my heartfelt appreciation to all those who provided their invaluable support throughout my time at the University of Tartu. I am particularly grateful to my supervisor, Mr. Mohamad Gharib, whose unwavering dedication and valuable guidance have been instrumental from the initial discussion of my thesis proposal to the final stages of my dissertation.

I am profoundly grateful to my family for their steadfast support and encouragement throughout my academic journey at the University of Tartu. I am especially thankful for my wonderful girlfriend, Maryam. With her enchanting presence, every challenge turned into an opportunity, and every moment became a cherished memory. Additionally, I express my sincere appreciation to all the interviewees for generously dedicating their time to participate in the research.

# Contents

# List of Figures

# List of Tables

An approach for Designing Microservice-Based Applications using a Domain-Driven Design Approach and Clean Architecture Principles

**Abstract:**

The current landscape of software services is marked by growing complexity, necessitating adaptive changes in software architecture to keep pace with rapid developments. As software architecture profoundly impacts code organization, the adoption of microservice architecture has gained popularity for breaking down systems into manageable services. However, the development and management of numerous small services present challenges. To address this issue, this master thesis introduces a novel approach that combines the principles of Clean Architecture and Domain-Driven Design to construct a microservice architecture. This methodology utilizes extensive business requirements as input and produces a code repository prototype as its output. The study provides an overview of this approach and presents a practical use case where it is implemented with actual code. Furthermore, the performance of this approach is analyzed through a comparison with the traditional software architecture paradigm, MVC (Model, View, Controller).

**Keywords:** Software Architecture, Domain-Driven Design, Clean Architecture, SOLID principles, factory design pattern.

**CERCS:** P170 Computer science, numerical analysis, systems, control

Lähenemisviis mikroteenusepõhiste rakenduste kujundamiseks, kasutades domeenipõhist disaini lähenemisviisi ja puhta arhitektuuri põhimõtteid

**Lühikokkuvõte:**

Praegust tarkvarateenuste maastikku iseloomustab kasvav keerukus, mis eeldab kiire arenguga sammu pidamiseks kohanemisvõimelisi muudatusi tarkvara arhitektuuris. Kuna tarkvaraarhitektuur põhjalikult mõjutab koodikorraldust, on populaarsust kogunud mikroteenuste arhitektuuri kasutuselevõtt süsteemide jagamiseks hallatavateks teenusteks. Kuid arvukate väikeste teenuste arendamine ja haldamine kujutab endast väljakutset. Selle probleemi lahendamiseks tutvustatakse käesolevas magistritöös uudset lähenemisviisi, mis ühendab puhta arhitektuuri ja valdkonnapõhise disaini põhimõtted mikroteenuste arhitektuuri ülesehitamiseks. See metoodika kasutab sisendina ulatuslikke ärinõudeid ja toodab väljundina koodirepositooriumi prototüübi. Uurimuses antakse ülevaade sellest lähenemisviisist ja esitatakse praktiline kasutusjuhtum, kus seda rakendatakse tegeliku koodiga. Lisaks analüüsitakse selle lähenemisviisi tulemuslikkust, võrreldes seda traditsioonilise tarkvaraarhitektuuri paradigma MVC (Mudel, Vaade, Kontroller) abil.

**Võtmesõnad**: STarkvaraarhitektuur, Valdkonnapõhine disain, Puhas arhitektuur, SOLID-põhimõtted, tehase disainimustrid.

**CERCS**: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine

# 1 Introduction

This thesis stresses the effect of proposing the creation of a software architecture process to build a microservice architecture. This introductory chapter describes the motivation for utilizing software architecture in the software engineering field. It then presents research questions, a list of contributions, and a structure of this thesis. I employed ChatGPT [1], an AI writing assistant, throughout the writing process to ensure grammar and punctuation accuracy and enhance the overall clarity of the written content. ChatGPT, developed by OpenAI, stands as a cutting-edge language model. It harnesses deep learning methods to produce text resembling human language, driven by the input it receives.

## 1.1 Motivation

Nowadays, software development, in general, and web applications in particular, become more challenging than ever as the final software product requires meeting the constantly changing business requirements. These changes in requirements put lots of stress on the development process and the software architecture. We believe that a solution to this problem should consider not only low-level coding principles but also the architecture as well as the design of the software application/system. Extending the software applications/systems with new features often becomes costly if the software applications/systems are not optimized. In addition, a poorly designed system might lead to a problematic implementation that becomes a big lump of technical debt. It is a problem for software engineers and the whole company, as solving such a problem will require extra resources and unplanned costs.

The increasing demand and complexity of new features in back-end applications may negatively influence the system quality, especially maintainability, during development Koller (2016). Solving this problem will require the software architecture to satisfy two conditions: (1) a seamless translation of the business requirements into a component of the software system; and (2) a coherent methodology to use these components to derive the code. The seamless translation is meaningful because inaccurate translation quickly causes technical debt, and other codes can be accumulated. A well-defined methodology ensures consistency, efficiency, and maintainability of the code base. It also facilitates collaboration among team members and allows for easier debugging and future enhancements.

This thesis proposes the combination of Domain-Driven Design (DDD) and Clean Architecture Principles by satisfying the above two conditions. MVX or MVC may suffice for

[1]https://openai.com/chatgpt

simple apps with little business logic Nunkesser (2021). There are also apps of medium complexity that do not need external infrastructures or do not need separate models for the layers. Those applications don't have a solid need to organize the business logic; however, the large size of the application is obligated to take care of Nunkesser (2021).

Adopting a clean architectural approach can help reduce the issue of low maintainability and significant amounts of code duplication because each layer is independent of the layers above and below it Nunkesser (2021). While the primary objective of employing the clean architecture is to orchestrate code organization Lakhai *et al.* (2022), it does not inherently address the translation of business requirements into the system's business logic. In essence, clean architecture specializes in organizing already articulated business logic. DDD emerges as the optimal choice to cater to the input of business logic. The primary advantage of DDD lies in its ability to articulate real-world incidents into a graph with ubiquitous language Bucchiarone *et al.* (2020), thereby facilitating the elicitation of essential data-centric information. Consequently, DDD complements the lacuna in Clean Architecture and furnishes a fully structured business domain for software architecture, wherein clean architecture capitalizes on its inherent strengths.

## 1.2 Research Questions

This study attempts to present a comprehensive approach to developing a micoservice architecture that enhances the development experience across various dimensions. The thesis meticulously unveils the proposed approach in a sequential manner. Following the elucidation of this approach, a case study is conducted, wherein applications are constructed using both the proposed approach and a conventional method involving the MVC (Model-View-Controller) architecture. This comparative analysis forms the core of the study and seeks to address the subsequent research inquiries:

**RQ1:** To what extent does the incorporation of Clean Architecture and Domain-Driven Design (DDD) principles enhance the understandability of the software architecture?

**RQ2:** How does the incorporation of Clean Architecture and Domain-Driven Design (DDD) principles facilitate the refactoring process of the software architecture in terms of reducing technical debt and improving code maintainability?

**RQ3:** In what manner does the incorporation of Clean Architecture and Domain-Driven Design (DDD) influence the extensibility of the software architecture over the system's lifecycle?

## 1.3 Contribution

The findings of this study aim to provide a guide to companies and industries for the improvement of large-scale software services or products. This study focuses explicitly on microservice architecture since it is helpful for the extensive complex software system. The use of these findings is not limited to software developers but gives product owners and project managers working on software services. This thesis's contributions are accomplished by providing a clear step of software architecture from the business requirements to the code implementation.

## 1.4 Structure of the Thesis

**The structure of the thesis is as follows:**

- **Chapter 2:** This chapter elucidates the fundamental principles of three pivotal concepts: Microservice architecture, Domain-Driven Design, and Clean Architecture. It delves into the motivations behind their development and outlines the merits that underpin their relevance in modern software engineering.
- **Chapter 3:** This chapter provides an intricate exposition of the suggested approach. It unveils a meticulous step-by-step journey through the approach, elucidating the rationale behind each stage while illuminating the advantages it bestows upon the development process.
- **Chapter 4:** I materialize the proposed approach into reality through a proof of concept. The narrative encompasses the in-depth implementation of code, offering readers a comprehensive understanding of its execution and logical flow.
- **Chapter 5:** This chapter validates the efficacy of the proposed approach through a rigorous comparison with traditional software architecture. It not only underscores the outcomes of this validation but also confronts potential threats to validity. Moreover, this chapter peers into the horizon, delineating future trajectories for the proposed approach's advancement.
- **Chapter 6:** I encapsulate the reflective conclusion. It encapsulates the thesis's findings, lessons learned, and the broader implications of the proposed approach. This chapter functions as a culmination that ties together the disparate threads explored throughout the thesis.

# 2 Background

This chapter introduces the conducted research about the three critical components in this paper, Domain-Driven Design (DDD), Clean Architecture, and microservice architecture, and summarizes their core feature.

## 2.1 System architecture

The system architecture was initially constructed on a single server, driven by the hardware's capabilities. However, with the advancement of technology, the approach to shaping software architecture has evolved over the years.

### 2.1.1 Monolithic architecture

A monolithic application typically comprises a user interface (UI) layer, a business logic layer, and a data access layer that interacts with the database, as shown in Figure 1 in the next page Kalske *et al.* (2018).

A monolithic architecture prevents separate module execution by encapsulating all functionality into a single program. This kind of architecture is closely coupled, and a separate process handles each piece of logic that handles a request. This enables developers to divide the application into classes, functions, and namespaces using the fundamental characteristics of the language Ponce *et al.* (2019).

This style of system architecture is effortless to develop and deploy since everything is in one place. This self-contained designed system also prevents delay due to the network, which makes the throughput higher than other system architectures Turis (2019). While adopting this kind of design to begin a project is smart because it enables you to investigate a system's complexity and the boundaries between its components Chen *et al.* (2017).

Although the application becomes increasingly intricate, the monolithic structure expands, transforming into a substantial and challenging-to-handle software component that is difficult to scale De Lauretis (2019). Moreover, when a developer exercises caution and prioritizes comprehensive documentation of the structure, the code may become tightly coupled and needlessly intricate. In cases where certain sections of code are interdependent, any required modifications can incur additional development costs Weerasinghe and Perera (2021).

In addition, exercising diligence and placing a premium on meticulous structural documentation can inadvertently lead to a codebase characterized by excessive interdependencies and unnecessary complexity. When segments of code are intricately intertwined, any subsequent adjustments can result in augmented costs to rectify the entanglements Weerasinghe and Perera (2021). The endeavor to comprehend such an architecture can also incur its own share of expenses if the documentation isn't meticulously crafted. The consolidation of all components into a singular position further deepens the imperative to install a well-structured framework. Additionally, when confronted with the necessity to rectify even the minutest fragment of code, the entire codebase necessitates deployment. For systems requiring frequent updates, this translates into numerous deployments solely for marginal alterations Ponce *et al.* (2019).

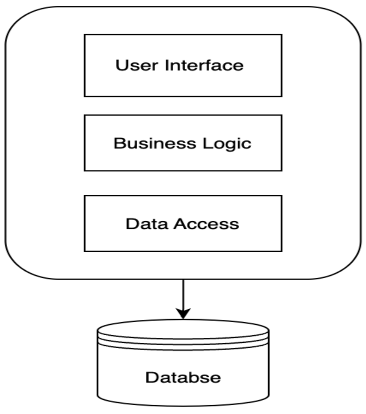


Figure 1. Monolithic architecture

### 2.1.2 Service-oriented architecture (SOA)

Enterprises use SOA to enhance agility and cost-effectiveness while reducing the burden of IT on the organization by positioning services as the primary means through which the solution logic is represented Erl (2008). SOA is not just an architecture of services seen from a technology perspective but the policies, practices, and frameworks by which we ensure the right services are provided and consumed Sprott and Wilkes (2004).
The system is organized into several services instead of being integrated into a single unit. Each service operates within its defined scope, accomplishing its specific objectives. Communication between services occurs through the network, storing their code in separate repositories, as shown in Figure 2 on the next page.

This architectural approach emerged due to the convenience of server management and deployment. Cloud technology has simplified server operations, enabling developers to create multiple repositories instead of consolidating them into one entity Weerasinghe and Perera (2021).

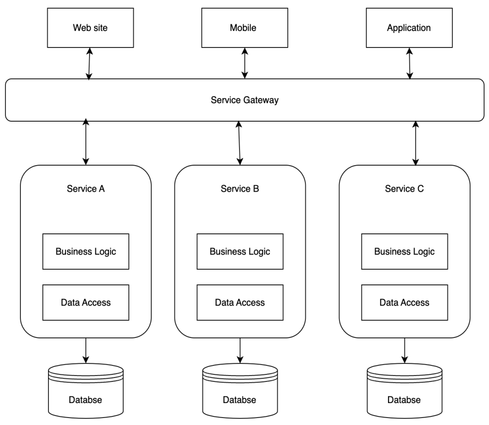


Figure 2. Service-Oriented Architecture

There are the following traits to explain the feature of SOA from Turis (2019).

1. **Loose coupling** – Service has minimal dependencies on each other. It increases the modularity so that modification of codes becomes much more manageable. For example, if a small portion of the code needs to be changed, not the whole code must be deployed, but only the service containing the code will be redeployed.

2. **Abstraction** – A service hides inner implementation, and they expose only their contract. It reduces complexity and increases readability.

3. **Reusability** – it should be possible to reuse the same service in different scenarios;

4. **Statelessness** – services do not manage the state, but they defer it to consumers;

5. **Composability**–services can be composed to create mutated services.

### 2.1.3 Micro-service architecture (MSA)

Unlike other architectural approaches, microservices are autonomous services that are smaller and easier to deal with. Microservice architectures can offer many advantages. The ability to design, develop, test, and release services with tremendous agility is crucial. Infrastructure automation makes Continuous delivery possible, which lowers the manual labor required for developing, deploying, and running microservices. MSA is especially well suited to cloud infrastructures because they considerably benefit from the elasticity and quick resource provisioning that the cloud enables Di Francesco *et al.* (2019).

MSA comprises fundamental building parts like core business services, infrastructure services, discovery techniques, and communication infrastructure Clarke *et al.* (2017). Microservices transform the entire software development process. As a result, microservices follow the evolutionary design, in which the company foresees that some functions may stop working. As conditions change, applications that can be expanded and reorganized are necessary for scalable business models, as shown in Figure 3. It is simple to alter the workflow because each microservice is a small business operation representing a discrete area of business functionality Shadija *et al.* (2017).

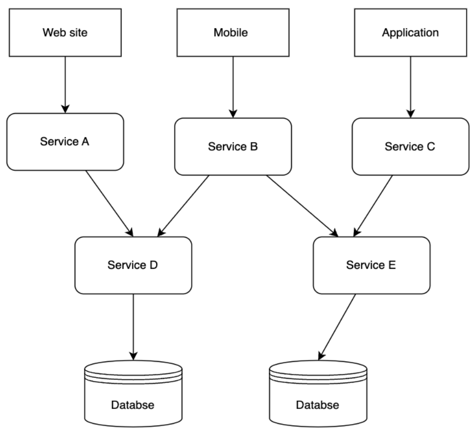


Figure 3. Microservice architecture

MSA should be utilized when the advantages outweigh the disadvantages. Unlike monolithic programs, which combine all functionality into a single process, MSAs split each functionality set into its service. Monolithic apps scale by being duplicated across several servers. On the other hand, microservices can be scaled by being distributed among servers and duplicating as necessary. In any event, microservices are not the greatest option Jamshidi *et al.* (2018).

The best benefit from the microservices is realized when use cases and conditions are carefully assessed and judgments are made. Furthermore, cloud-based platforms can be easily used in MSA development since autonomous services can be set up separately, profit from the cloud's flexibility, and provide resources quickly Pahl *et al.* (2017).

## 2.2 Clean architecture principle

The software architecture of a system is defined as “the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them” Bengtsson and Bosch (1999).

Without a clearly defined architecture, the source code will be a collection of unorganized codes that are tightly coupled and hard to maintain. For instance, if a developer wants to modify one specific component, tightly coupled components must also be adjusted. Furthermore, low-readability software architecture costs significant development costs: time, money, development speed, and an enormous amount of code. This emphasizes the vital role software architecture plays in developing software products.

This section discusses clean architecture principles and SOLID principles, whereas Clean Architecture organizes code into layers for maintainability and flexibility Martin (2017), and SOLID principles offer guidelines for clean, extensible code Madasu *et al.* (2015). Together, they promote scalable and manageable software architecture.

### 2.2.1 SOLID Principles

SOLID is an acronym encapsulating the five class diagram design principles created by Robert Martin to improve the quality of software development life-cycle processes Chebanyuk and Markov (2016).

SOLID principles guide developers to build we can build efficient, reusable, and non-fragile software, which is sustainable and maintainable for long-term needs. The major issues concerning software architecture quality are reusability, extensibility, sustainability, and maintainability. SOLID principles provide a suitable answer to develop an efficient Software architecture that can overcome the above four problems Madasu *et al.* (2015).

This principle was named by the acronym of five principles as follows:

1. **Single Responsibility Principle**: A class should have only one job to exist: “A class should have only one reason to change” Martin and Martin (2006). If a class has more than one responsibility, the code is tightly coupled, and there is a need to modify coupled codes every time. This principle helps to decouple each class to increase maintainability.

2. **Open-Closed Design Principle**: Software entities should be open for extension but closed for modification Chebanyuk and Markov (2016). In other words, it should be able to extend the behavior of an entity with minimum change for its source code. This principle helps to keep existing code stable and makes it easier to add new features without introducing bugs.

3. **Liskov Substitution Principle**: Objects of a parent class should be replaceable with objects of their child class without affecting the correctness of the program Madasu *et al.* (2015). This principle promotes the reusability of the component to create a more robust and maintainable object-oriented design.

4. **Interface Segregation Design Principle**: "The class interfaces can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of methods while others use the other" Martin and Martin (2006). Clients should only know about the methods or interfaces that interest them so they are not distracted by non-related information Madasu *et al.* (2015).

5. **Dependency Inversion Design Principle**: "High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions" Martin and Martin (2006). This principle helps to decouple components, making changing and maintaining the software easier without affecting other parts of the implementation details Madasu *et al.* (2015).

### 2.2.2 Clean Architecture

Clean Architecture is one of the software architectures that embodies the SOLID principles. The main idea behind Clean Architecture is to create software that is independent of external concerns, such as the user interface, the database, or any specific framework. Instead, the architecture should focus on the application's business logic Martin (2017).

Below Figure 4 is created by the creator of clean architecture principles, Rober C. Martin. The software architecture consists of four layers, like an onion, and each layer depends on the inner layer Martin (2017).

1. **Entities:** It encapsulates enterprise business rules. An entity can be an object with methods or data structures and functions. It doesn't matter so long as many different applications in the enterprise can use the entities Aguilar and Figueira (2020).

2. **Use cases:** The software in this layer has business rules specific to the application. It serves as the implementation of all system use cases, coordinating data flow to and from the entities and directing these entities to utilize enterprise business rules to achieve the objectives of each use case. Changes made to this layer should never impact the entities, nor should it be affected by external factors like the database, user interface, or common frameworks, as it remains isolated from such concerns. However, modifications to the application's operation will influence this layer. If the details of a use case change, certain codes in this layer will inevitably be affected Aguilar and Figueira (2020).

3. **Interface Adapters:** The software within this tier consists of adapters responsible for converting data between formats most suitable for the use cases and entities and the format preferred by external entities like the Database or the Web. The models likely serve as data structures transferred between controllers, use cases, presenters, and views to facilitate data flow Aguilar and Figueira (2020).

4. **Frameworks and Drivers:** This layer comprises frameworks and tools, including the Database, Web Framework, and others. Typically, minimal code is written directly in this layer, mainly consisting of glue code responsible for communication with the inner circle. This layer contains all the details defined in the interface adapter layers Aguilar and Figueira (2020).

On the right below the Figure 4 is an image of the data flow with clean architecture principles. Data is received by the controller in the interface adapter layers. Next, it is handed to a use case interactor in the use cases layer to process the input. Finally, the use cases layer output is handed to the presenter to format preferred by external entities.

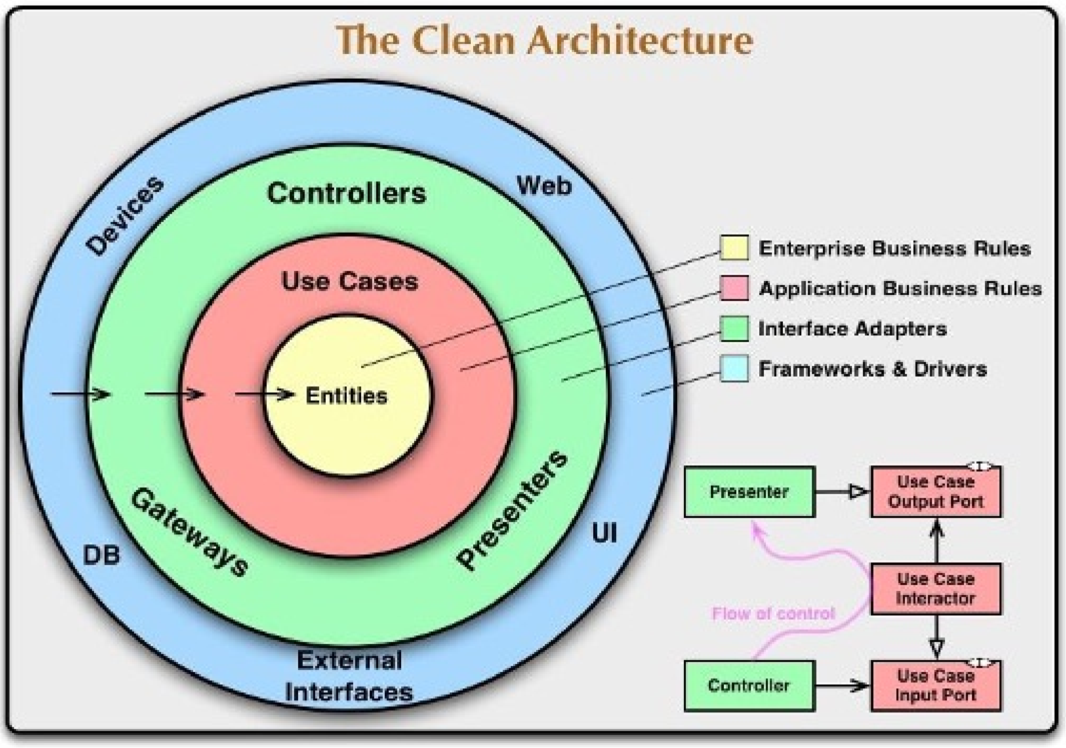


Figure 4. The “Clean Architecture” schema proposed by Robert C. Martin (2017)

## 2.3 Domain-Driven Design (DDD)

When creating a software application, developers try to understand the real-world problem the application should solve rather than just jumping into writing code. They spend time identifying the most important concepts, rules, and relationships the software needs to model. Once they understand the problem domain, developers can create a model using software design patterns and tools. This model serves as a blueprint for the actual code that will be written to implement the application.

The model-driven process known as Domain-Driven Design (DDD) captures domain information pertinent to software design. DDD promotes agile, collaborative modeling of domain experts and software developers to promote domain expertise and the accuracy of an emergent design Rademacher *et al.* (2018).

Developers work closely with domain experts, who have expertise in the problem domain, to create a model of the problem domain based on the business requirements. This model breaks down the problem domain into smaller, more manageable parts and represents the key concepts, entities, and relationships in the problem domain Merson and Yoder (2020). By creating a detailed problem domain model, developers can ensure that the software is well-aligned with the users' needs and requirements and solves real-world problems more effectively.

DDD is an excellent place to start when looking for microservices. Still, it's up for debate where to draw the line for the constrained context Xie *et al.* (2018); quite a few inconsistencies in the work of domain experts, analysts, designers, and developers led to its evolution. It became clear that a common language had to be created. Complex systems with a wide range of domain knowledge domains best suit DDD. The introduction of well-defined domain boundaries comes at a cost. Bounded contexts are the various domain constituents. Each constrained context can be treated equally as a separate microservice Vural and Koyuncu (2021).

### 2.3.1 Strategic design

Strategic patterns aim to evaluate the problem domain and decompose the problem domain of a software system into multiple sub-domains Turis (2019). Some specific terms are used in the process of evaluation of the problem domain.

1. **Bounded context**: A Bounded Context establishes a clear boundary for a specific domain model, encompassing sub-domain parts and employing a shared ubiquitous language to describe its concepts, properties, and operations. Each Bounded Context defines unique meanings for terms within its domain, allowing the same term to have different interpretations in other Bounded Contexts Turis (2019).

2. **Context map**: Determining appropriate Bounded Contexts remains a difficult task. In addressing this challenge, Context Map models and diagrams, along with context mapping as a practice and the strategic DDD patterns, play significant roles in defining the relationships between Bounded Contexts Kapferer and Zimmermann (2020).

3. **Ubiquitous language**: The language used in this context establishes the structure and meaning of domain concepts pertaining to the software being developed. It also harmonizes the terminology and specialized language to facilitate effective stakeholder communication Rademacher *et al.* (2020). As a result, both software architecture and code must adhere consistently to the terms of the ubiquitous language, mirroring the structures and relationships of domain concepts as depicted in domain models Rademacher *et al.* (2020).

### 2.3.2 Tactical design

Tactical patterns serve the purpose of handling complexity within the domain. They aim to represent and illustrate objects, their behavior, meaning, function, and relationships in a cohesive manner. Each pattern provides guidance on implementing an object with specific functionality and attributes to enhance the overall model's readability, maintainability, and extensibility Turis (2019).

1. **Entity**: An entity is an object with attributes and functions whose unique identity is significant. Even if certain attributes change, the object retains the same identity Turis (2019). The three main components of the entity are public attributes, public operations, and the transformation of attributes under operations Chen *et al.* (2019). The below formula is a reference of the equation 1 of the entity retrieved from Chen *et al.* (2019). In this context, properties refer to the set of attributes associated with the entity, while "id" denotes a distinct identifier. Operations encompass a collection of functions, and "H" represents the transformation of properties resulting from these operations Chen *et al.* (2019).

$$\text{Entity} = <\text{properties} = \{\text{id}, \text{p1}, \text{p2}, \ldots, \text{pn-1}\}, \text{operations} = \{\text{get}, \text{set}, \text{otherOperations}\}, H> \tag{1}$$

2. **Value Objects**: As its name suggests, a value object is represented solely by its value. It lacks identity, and any changes made to it result in a different value Turis (2019). The below formula is a reference of the equation 2 of the value object retrieved from Chen *et al.* (2019). Compared to entities, value objects express only the clear characteristics of a certain concept Chen *et al.* (2019).

$$\text{Value Object} = \langle \text{properties}, \text{operations} \rangle, \tag{2}$$

3. **Aggregates**: An aggregate represents a cohesive group of entities and value objects that form a transactional consistency boundary. The entire aggregate should maintain consistency at all times. To achieve this, an aggregate root is established as the entry point to the aggregate, and other entities and value objects are considered internal, inaccessible from outside Turis (2019). The below formula is a reference of the equation 3 of the aggregates retrieved from Chen *et al.* (2019).

$$\text{Aggregate} = \langle \text{Entities}, \text{Value Objects} \rangle \tag{3}$$

4. **Repository:** By encapsulating the logic involved in storing, retrieving, updating, and removing aggregates from a particular persistence store, a repository acts as a mechanism for persisting aggregates. A model can be built without considering infrastructure issues by removing the technical details from a store's implementation Turis (2019).

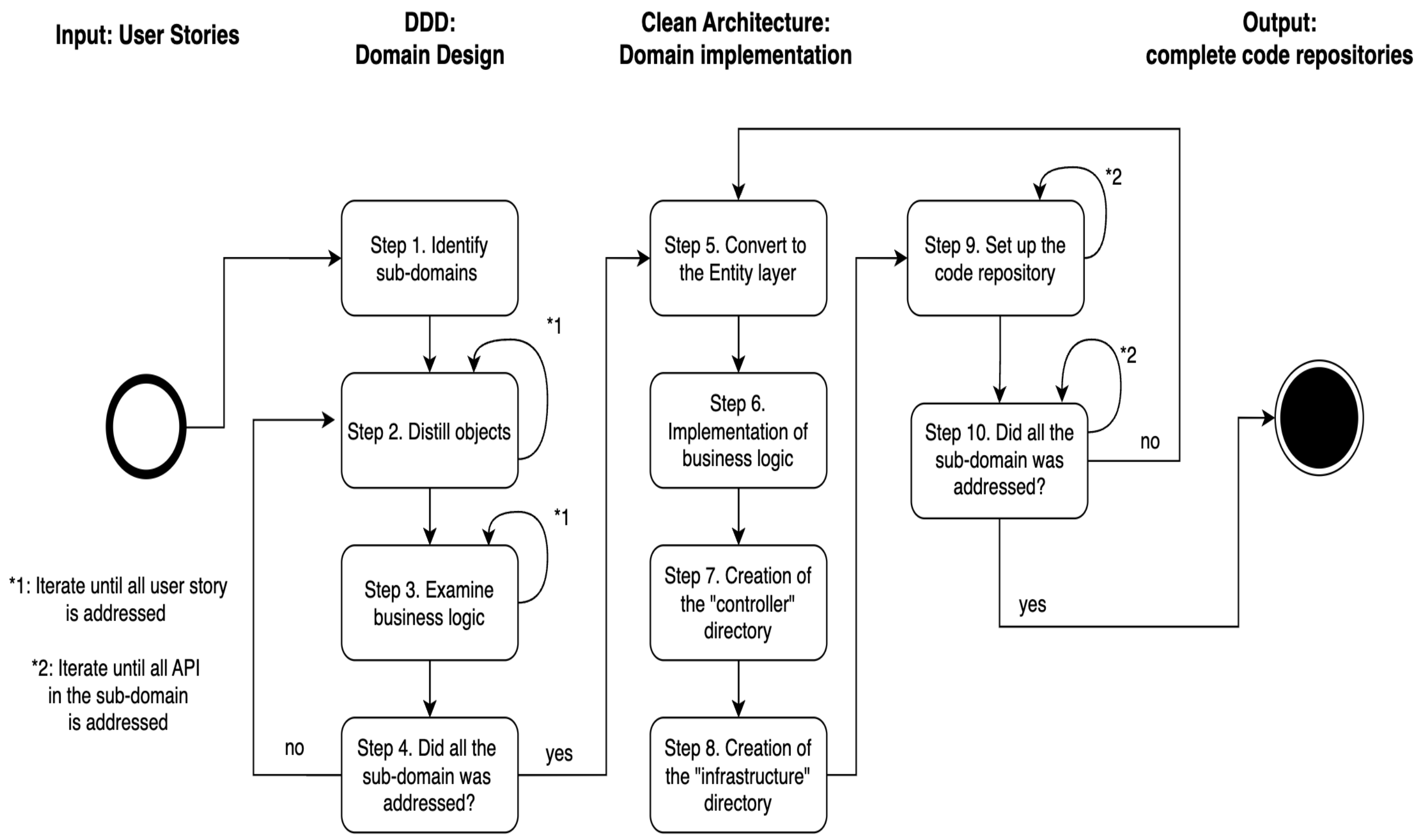
5. **Factory Design Pattern:** The book Design Patterns popularized the software design pattern known as Factory Gamma *et al.* (1995). It is in charge of producing sophisticated objects and aggregates. It is particularly helpful when creating a new aggregate involves multiple steps during which the existing aggregate is inconsistent, and the aggregate consists of numerous entities and value objects. The factory creates a fully consistent aggregate while encapsulating the creation logic. A factory can be implemented as a class or as a method on an aggregate root Turis (2019).

# 3 Proposed Approach

This section introduces the approach of using DDD and Clean Architecture to create Microservice Architecture. It explains step by step from the business requirements to the completed prototype of the software application.

The flow of the proposed approach is as follows:

1. **Input**: As input, user stories should be elicited from the business requirements. They are tailored to specific use cases, serving as cues for identifying domain logic that must be appropriately modeled within the domain model Steinegger *et al.* (2017). Without well-defined user stories, the development process risks being directionless and might lead to irrelevant solutions.
2. **Domain Design**: The DDD designs the problem domain based on the given user stories. The created domain consists of multiple domains with bounded contexts. This step is necessary to understand the domain and prevent ambiguity or confusion during development.
3. **Domain Implementation**: Clean architecture applies each domain inside the problem domain to build the code organization. This step is crucial as Clean Architecture helps avoid tight coupling between different parts of the code-base, making it easier to modify or replace components without affecting the entire system.
4. **Output**: After completing the approach, developers see the complete structure of the code of each domain and their relationships. The output also fosters better communication within the development team, making it easier for them to collaborate and make informed decisions during the development process.



Input: User Stories
DDD: Domain Design
Clean Architecture: Domain implementation
Output: complete code repositories
Step 1. Identify sub-domains
Step 2. Distill objects
Step 3. Examine business logic
Step 4. Did all the sub-domain was addressed?
Step 5. Convert to the Entity layer
Step 6. Implementation of business logic
Step 7. Creation of the "controller" directory
Step 8. Creation of the "infrastructure" directory
Step 9. Set up the code repository
Step 10. Did all the sub-domain was addressed?
*1
*2
no
yes
*1: Iterate until all user story is addressed
*2: Iterate until all API in the sub-domain is addressed

Figure 5. Overview of the proposed approach

## 3.1 Domain design

This stage uses the DDD approach to design each sub-domain structure and API documentation. It starts with a list of user stories as input and breaks down complex business problems into smaller and cohesive sub-domains. The result comprises each sub-domain structure with a context map and a list of available APIs, including their names, inputs, and outputs.

The domain design stage referenced the idea from Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications Gamma *et al.* (1995). This paper explains the whole process of DDD, from business requirements to the implementation stage. The proposed approach referenced domain design from the paper.

### 3.1.1 Step 1. Identify sub-domains based on key business processes

This step divides The problem domain into multiple sub-domains, as Figure 6 on the next page shows. Breaking a big domain into small sub-domains offers benefits such as modularity, focused development, and scalability. Smaller sub-domains encourage loose coupling and reusability while being simpler to comprehend, manage, and test. Domain experts can focus on particular sub-domains to ensure correct solutions and a more agile development process. Additionally, by drawing distinct lines between sub-domains, communication, and solutions are enhanced Kapferer and Zimmermann (2020).

Each user story is analyzed to specify the key business process involved. Those key business processes are high-level components representing significant parts of the application's functionality. They are considered sub-domains, and each user story should be clarified which sub-domain is related.

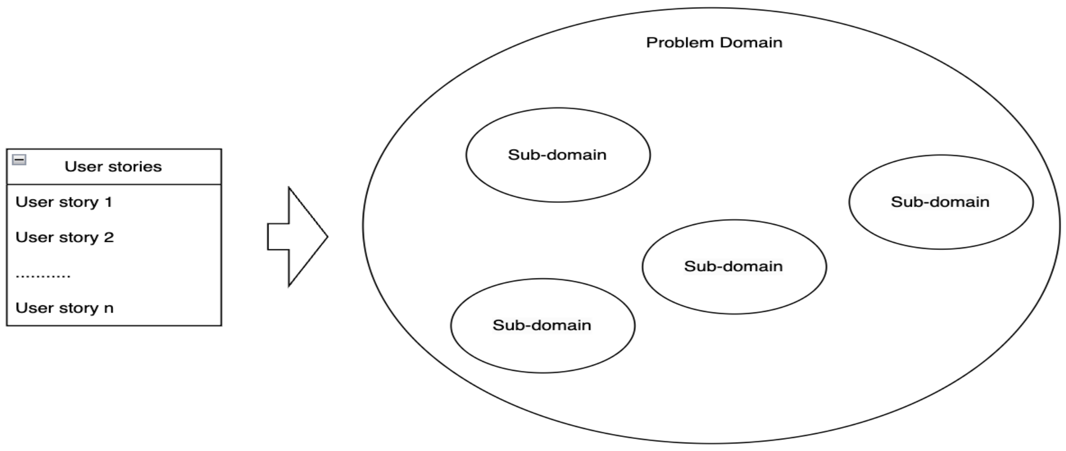


Figure 6. Sub-domain elicitation

### 3.1.2 Step 2. Distill objects from user stories

This step focuses on extracting three types of objects from the user stories that pertain to the sub-domain. Creating a consistent structure enables code reusability and aligns well with Clean Architecture principles for effective implementation. A shared ubiquitous language is also established, enhancing communication and understanding among stakeholders and the development team.

Figure 7 shows the process of detailed elicitation of objects with example objects. On the left side, start from the list of user stories; it takes one domain from the created list of sub-domains and collects the corresponding user stories. Next, iterate the list of user stories by processing the below steps. Throughout the iteration, the already defined object is updated as needed.

1. List all the required entities with necessary attributes in the user story. If the entities or aggregates already exist, move to the following entity.
2. Elicit aggregate based on newly created entities to have consistency.
3. Elicit value objects from the attributes in entities or aggregates.

After eliciting three types of objects, each object should be filled with minimum variables to suffice each user story. The variables in those objects should not contain more than necessary to avoid slowing down the implementation.

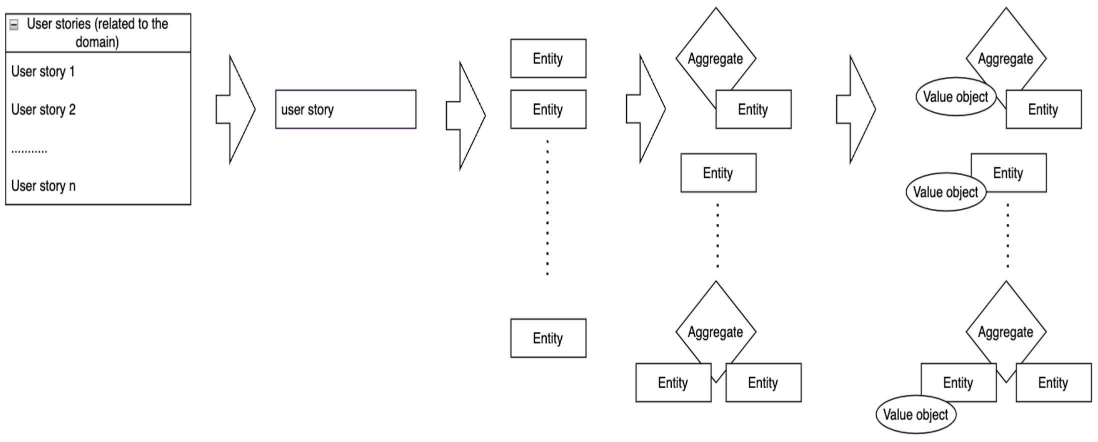


Figure 7. Objects elicitation

### 3.1.3 Step 3. Examine business logic

This step aims to examine the business logic of the user stories that relate to the domain with those defined objects from step 2. This step is necessary to refine the domain model's correctness and completeness, ensuring it accurately represents domain interactions and rules. It validates entities and aggregates against user story requirements, maintaining consistency and integrity.

The process of this step is similar to step 2. It takes the related user story to the domain, and continue the business logic elicitation until all the user stories in the list are addressed. As the result of this step, the documentation that contains the below attributes should be created.

- **User Story Identifier**: It tracks which user story it writes about.
- **Input**: The input for the user story should be defined. It is a list of variables or objects with their data type.
- **Business logic flow**: It is the whole process of the business logic to complete the user. This flow should be described with numbered list. It also explicitly explain what data is modified and how does it modified.
- **Output**: The output for the user story should be defined. It is a list of variables or objects with their data type.

Throughout the process, create repositories for aggregates or entities that handle any data related to their creation, updates, or deletions. Should any inconsistencies arise with the objects defined in step 2, return to step 2 to make the necessary updates. After updating the objects, revisit step 2 to ensure the overall consistency of the remaining objects

### 3.1.4 Step 4. Completion of each sub-domain

Repeat steps 2 and 3 until all the user stories in the list are addressed.

## 3.2 Domain Implementation

Once the problem domain has been defined, the subsequent step involves the application of Clean Architecture to each sub-domain. At this stage, multiple sub-domains are interconnected using a context map, and detailed API documentation for each sub-domain exists. Clean Architecture principles are then applied to each domain, creating a unified code repository for each sub-domain, as the context map dictates. This phase aims to establish multiple code repositories that align with a microservice architecture.

This phase holds essential importance within the proposed approach as it involves the actual implementation of the code. The developed sub-domains are meticulously structured and implemented, making full utilization of clean architecture principles.

The domain implementation referenced mainly Clean Architecture made by Martin (2017). Some technical features of software architecture are also referenced: input port Sanchez *et al.* (2022) and factory design pattern Gamma *et al.* (1995). Below are the main ideas of the proposed software architecture, and Figure 8 visualizes it. The layers are color-coded, aiding in their comparison with clean architecture. Each arrow in the diagram represents the dependencies between the layers.

1. The software architecture involves the distinct segregation of five layers: entity, usecase, interface, controller, and infrastructure. As depicted in Figure 8, these layers align with the core principles of clean architecture, providing equivalent functionalities. However, there is one distinction in the relationship between the usecase and interface directories. Usecase relies on the functions defined in the interface, which contain functions relevant to external devices. While this contradicts the conventional rules of dependency direction Fowler (2012), it is a logical necessity to configure it in this manner. As per the principle that details must always rely on the interface Madasu *et al.* (2015), the usecase directory, which encompasses the detailed implementation, inevitably needs to establish a dependency on the interface directory.

2. The five directories mentioned earlier will reside within the "internal" directory, focusing on the domain-specific aspects, while other files will be located outside this directory. Figure 8 illustrates that separate external files exist in the root directory, such as database-related or deployment-related elements. These files are utilized within the "cmd" directory, where the overall business logic in the "internal" directory is constructed. This architecture effectively dissociates the business domain from unrelated external files.

3. Within the "controller" directory, as illustrated in Figure 8, three distinct sub-directories exist: "inputPort", "api", and "presenter". The primary role of the "controller" directory is to receive requests, pass the data to the "usecase" directory for processing, and subsequently return the output as a response.

    (a) The "api" directory houses API handler functions responsible for handling incoming requests.

    (b) Meanwhile, the "inputPort" functions validate and transform the requests received within the "api" directory. Once validated, the requests are passed to the "usecase" directory for processing, and the resulting output is subsequently transferred to the functions in the "presenter" directory, as required.

(c) The functions in the "presenter" directory are responsible for converting the data into a format suitable for the User Interface and returning it as the response.

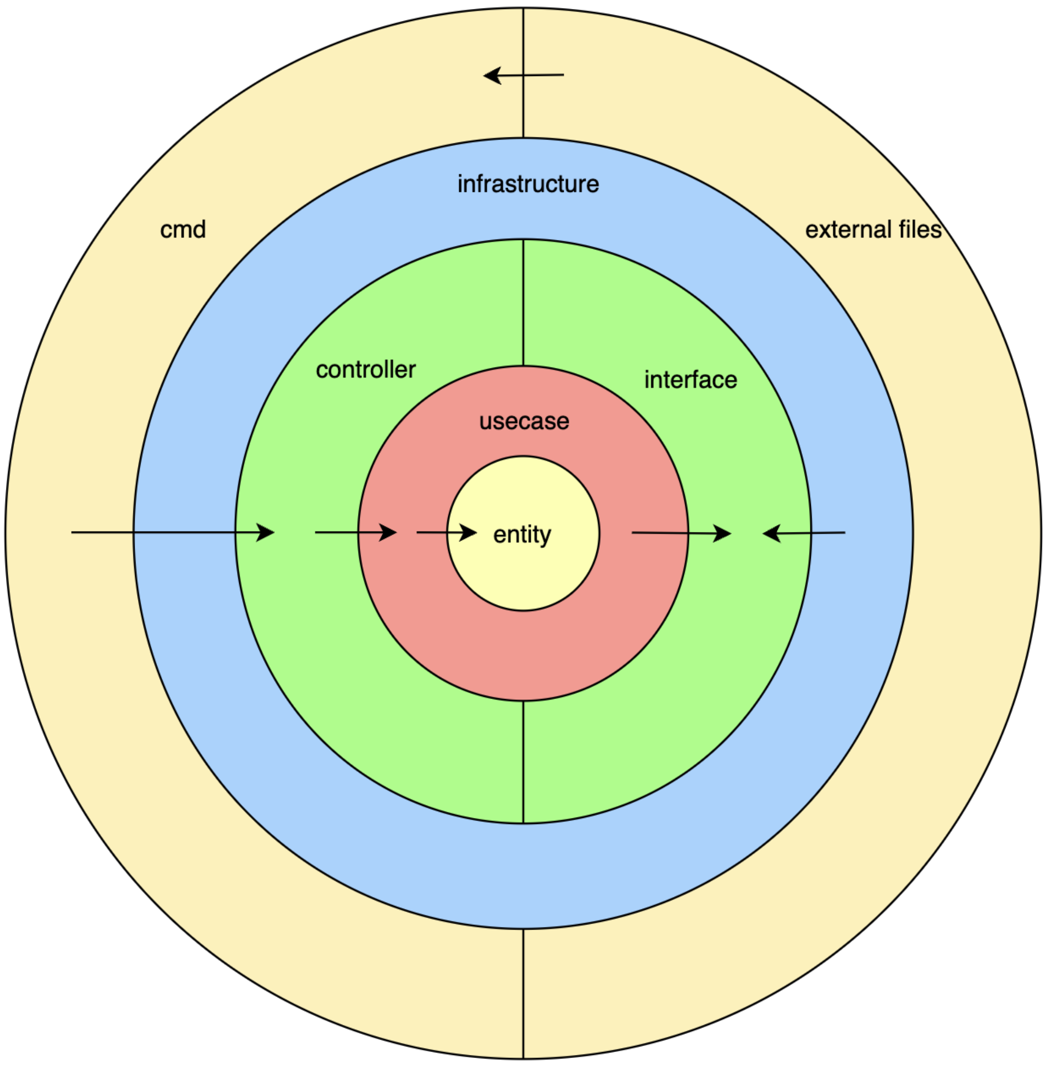


Figure 8. Proposed software architecture based on clean architecture

### 3.2.1 Step 5: Convert defined objects into classes

In this step, all the defined entities, value objects, and aggregates in the entity layer are implemented. Figure 9 illustrates the flow with some exemplary objects. The objects within the rectangle above depict the diagram created during the Domain Design stage.

Each file within the directory should contain more than one entity or aggregate labeled with the respective entity or aggregate name. In cases where similarities exist, developers have the flexibility to divide one file into multiple files based on grouping criteria. Additionally, value objects are incorporated within the file of the aggregate if they solely depend on one specific aggregate or entity. However, if value objects are shared among multiple entities or aggregates, they should be consolidated into a single file named "common." The "entity" directory currently contains several files, each accommodating the specified arrangements.

### 3.2.2 Step 6: Implementation of business logic

This stage aims to finalize the contents of the "usecase" and "interface" directories based on the API documentation developed during the Domain Design phase.

The "usecase" directory comprises files, each containing functions that implement the specified "flow" outlined in the API documentation, in other words, business logic. These functions are organized and grouped according to the most relevant class from the entity layer, with each file residing in the "usecase" directory bearing the name of the corresponding entity.

Any external functions required to support the functions in the "usecase" directory are defined in the "interface" directory. These external functions are unrelated to domain logic and include tasks such as data retrieval from databases or external APIs. To adhere to Clean Architecture principles, these functions are accessed through an interface rather than direct implementation Madasu *et al.* (2015).

Figure 10 illustrates the process of creating files within the "usecase" and "interface" directories. Starting with a list of available APIs drawn on the left side from the domain design phase, one API is selected, processed, and saved into the corresponding directories. This iterative process continues until all APIs have been addressed. The detailed flow of the process is as follows. During the below process, developers can optimize already defined function signatures since some can be redundant or unnecessary.

1. Developers individually examine an API to assess the required interactions with the classes defined in the entity layer. If necessary, they create helper functions within the same file.

2. Developers implement a function to fulfill the API's purpose, adhering to the specified "flow" detailed in the API documentation.

3. While implementing, if any essential function necessitates communication with external devices, developers add the corresponding function to the appropriate interface within the "interface" layer.

4. Ensure that each function stored in the "usecase" directory is appropriately grouped according to the file defined in the "entity" directory.

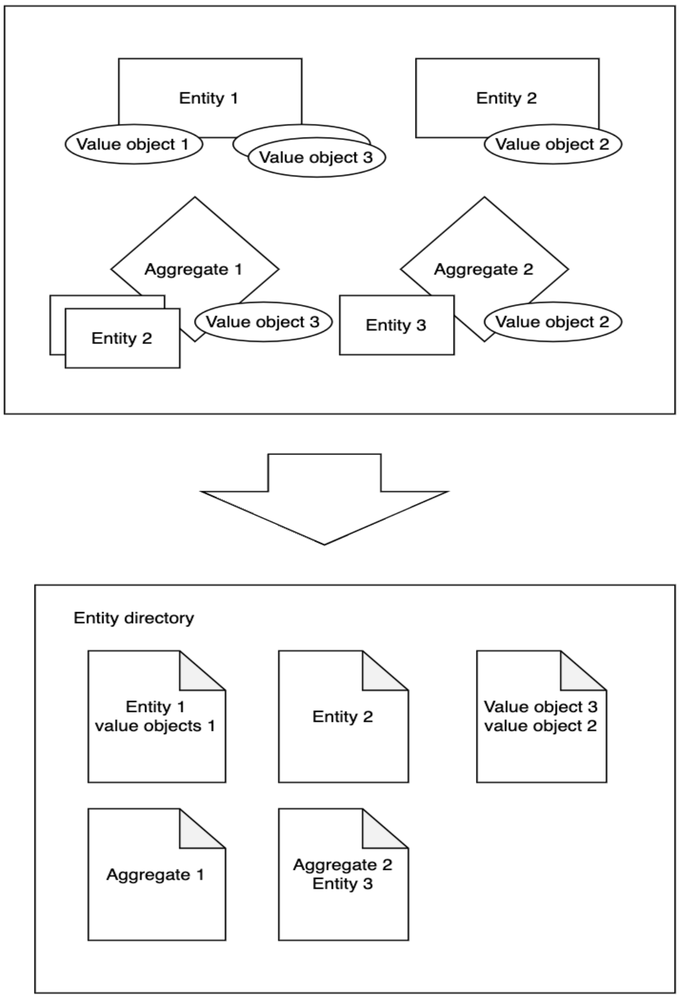


Figure 9. Entity layer creation

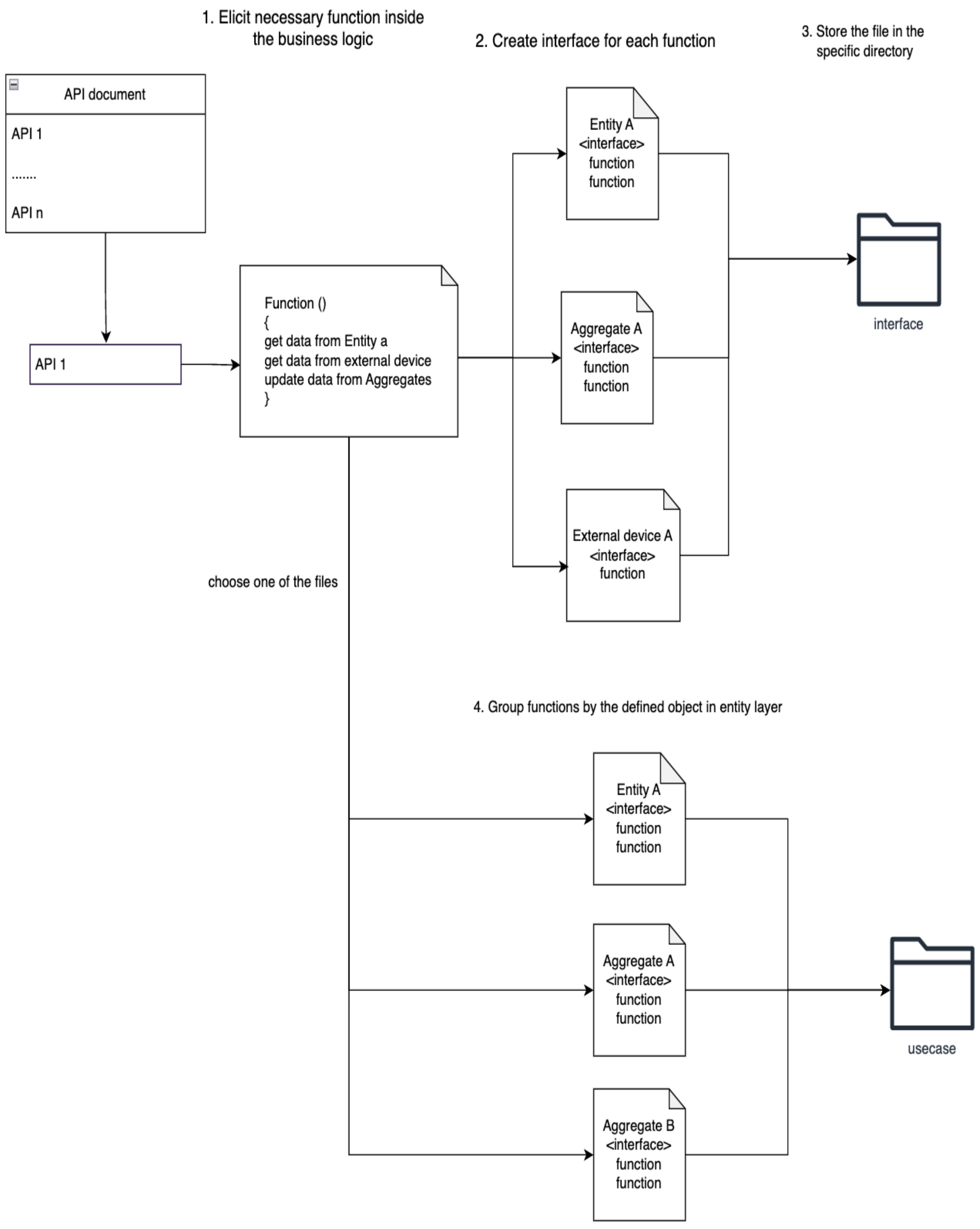


Figure 10. "usecase" and "interface" directory creation

### 3.2.3 Step 7: Build controller directory

This stage aims to finalize the contents of the "controller" directories based on the API documentation developed during the Domain Design phase and "usecase" directory created in the previous step.

Figure 11 provides an illumination of the DDD components that align with the interface adapter layers in the Clean Architecture principles. Positioned in the center of the figure are the three interface adapter components: API handler functions, Boundary, and Interfaces.

- API handler functions are responsible for handling APIs, where their primary objective is to relay data to the bounded context and return the resulting output.
- The long vertical line serves as the boundary between the bounded context and the external environment. To ensure consistency within the bounded context, the "inputPort" directory and "presenter" directory adapt and convert the data structure as necessary to facilitate the exchange of data between the bounded context and the external environment.
- The lower half of Figure 5 demonstrates that interactions with external devices are achieved through interfaces. As DDD focuses solely on the bounded context, any essential interactions involving external devices should always be designed to be interchangeable and substitutable.

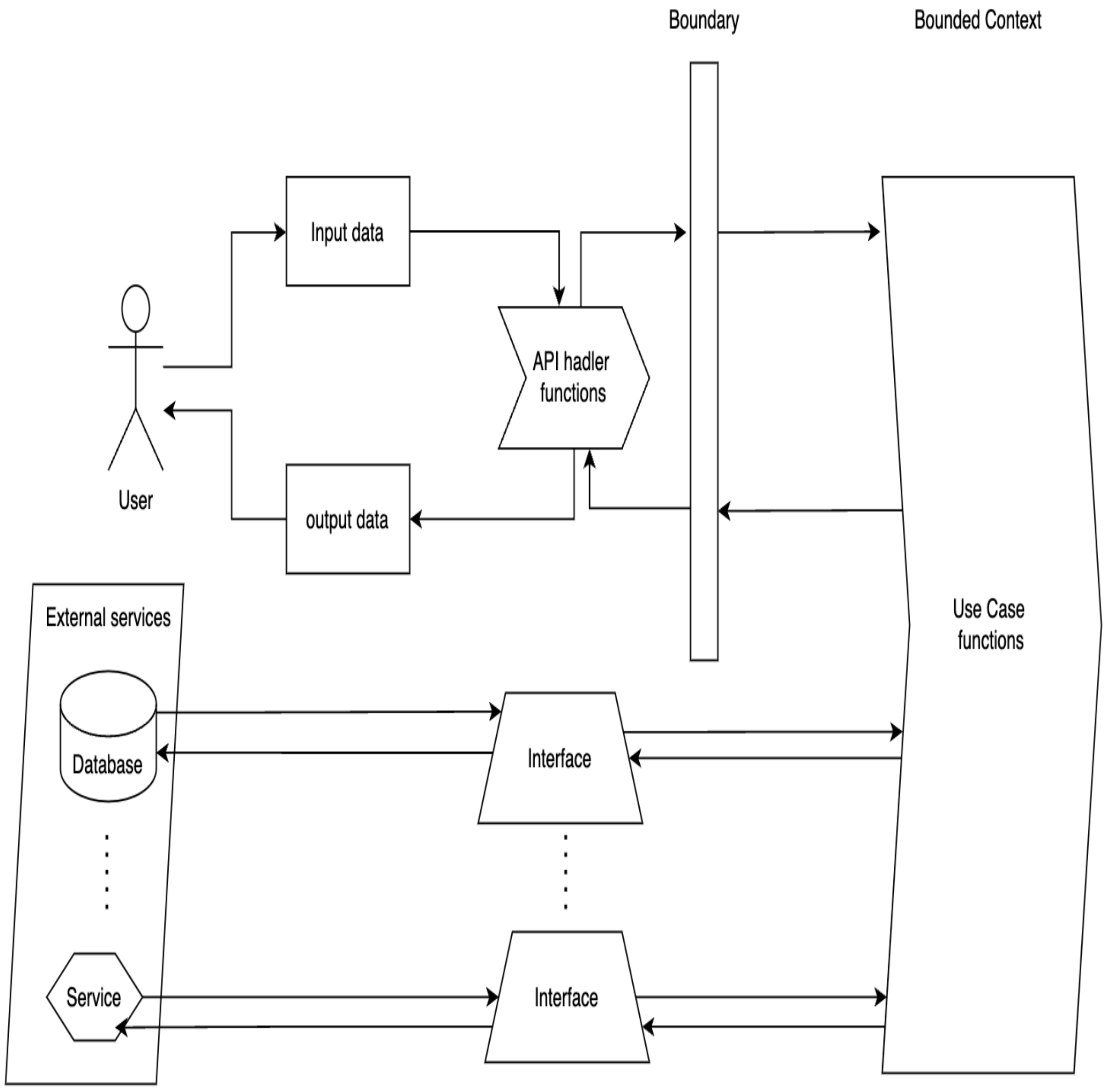


Figure 11. Controller layer creation

A new directory called “controllers” is created, with three sub-directories: "api", "inputPorts", and "presenters". Developers take one user story and create new files in those three directories as required.

Starting with a list of available APIs drawn on the left side from the domain design phase, one API is selected, processed, and saved into the corresponding directories.This iterative process continues until all APIs have been addressed. The detailed flow of the process is as follows.

1. Develop a function within the "api" directory, following the guidelines specified in the API documentation from the Domain Design phase.
2. During the creation of the above function, if input validation is necessary, generate a function in the "inputPort" directory. This function will be responsible for validating incoming requests received within the "api" directory.
3. If any function related to middleware usage is required, it should be created as a subdirectory named "middleware" within the "controller" directory. Upon passing through the function defined in the inputPorts directory, the input proceeds to a function defined in the use case directory, where the input undergoes processing.
4. The function defined in the use case layer returns the output, which should subsequently be passed to a function defined in the "presenter" directory. This function transforms the output into the desired format.

### 3.2.4 Step 8: Creation of infrastructure layer to write the implementation

In the process outlined in step 6, developers establish an "infrastructure" directory, which comprises multiple sub-directories, each corresponding to an external service identified during the earlier steps. In step 6, function signatures were created, specifically tailored to each external service, and these functions are subsequently implemented within this directory. Additionally, each sub-directory for an external service should adhere to the factory design pattern, creating instances to handle the respective functions.

### 3.2.5 Step 9: Create a cmd directory to build a server

During this step, a server is constructed to enable the functioning of the sub-domain. The "cmd" directory is generated along with other essential files required to run the server. The process for this step unfolds as follows.

1. Put all the created directories and files from Step 5 to Step 9 inside the new directory called “internal.” The "internal" directory is under the root directory.

2. In the root directory, create the necessary files and directories to facilitate the server building process. These files and directories may include configuration files, build scripts, server initialization scripts, and any other components required to set up and run the server effectively.

3. Under the root directory, a "cmd" directory is established to streamline the process of building the server. This directory retains essential files necessary for server construction and execution. Additionally, these files are responsible for reading external files or directories under the root directory, such as configuration files, to aid in the server-building process. The factory design pattern, implemented within both the "infrastructure" and "usecase" directories, harnesses its advantages, such as dependency injection, encapsulation of business logic, and concealment of complex object initialization Chen *et al.* (2019).

### 3.2.6 Step 10. Completion of code repository for each sub-domain

Repeat steps 5 to 9 until all the sub-domains are processed.

# 4 Implementation

In this section, the proposed approach is implemented to showcase the process and provide evidence of its viability in real-life use cases. The implementation aims to demonstrate the approach's proof of concept and practical application.

The proposed approach is implemented within the context of the existing software architecture. An existing code repository, available at https://github.com/gieart87/gotoko, had initially been developed as a small e-commerce business following the MVC (Model View Controller) software architecture.

This implementation is done by the Golang [2] programming language created by Google. Clean Architecture emphasizes separation of concerns, maintainability, and modular design Aguilar and Figueira (2020), and Golang's features align with the following principles MALINA (2016):

- **Simplicity:** Go's concise syntax, and minimalistic design encourage clean and readable code. Clean Architecture values clarity, making Go's simplicity a natural fit for creating well-organized and understandable architecture layers.
- **Package Management:** Go's built-in package management simplifies dependency management, enabling clear separation of external dependencies for each layer of Clean Architecture. This promotes encapsulation and reduces the likelihood of tight coupling.
- **Strong Type System:** Go's strong typing ensures data flows between architectural layers with clear types, reducing ambiguity and preventing unintended data mixing. This aligns with Clean Architecture's emphasis on strict boundaries and encapsulation.

The step for the implementation process of this section is as follows.

1. Elicit user stories from the existing code repository.
2. With the gained user stories, the proposed approach is then applied to create a new code repository. This new repository is developed following the principles and guidelines of the proposed approach, demonstrating how it addresses the same user stories but with a different software architecture. This new repository is now called the "new" software architecture.

[2] https://go.dev/

To ensure a fair comparison between the two software architectures, the code repository undergoes a refactoring process and is enhanced with additional functionalities until it reaches a reasonable size. This refactored code repository is now called the "traditional" software architecture. In the next Validation section, "traditional" and "new" architectures are compared to validate the proposed approach.

## 4.1 User Story Elicitation

The short description and user stories retrieved from the existing code repository are as follows:

**Description**: The application is an e-commerce platform that allows users to browse, purchase products, and make payments. It provides a user login feature, enabling users to access their purchase history and view the available products. After completing the payment process, users can access their order details. Below is the summary of the user story in Table 1.

| **User story** | **Explanation** |
|---|---|
| US1 | As a user, I can create a user as a customer. |
| US2 | As a user, I can log in. |
| US3 | As a customer user, I can list products with a specific page number to see particular products. |
| US4 | As a customer user, I can get a specific product identified by a unique product identifier. |
| US5 | As a customer user, I can see the information about the items in the shopping cart. |
| US6 | As a customer user, I can add one product type to the shopping cart. |
| US7 | As a customer user, I can remove one product type from the cart. |
| US8 | As a customer user, I can update items in the cart so that multiple types of products can be added or removed. |
| US9 | As a customer user, I can pay (cash or credit card) to complete the shopping. |
| US10 | As a customer user, I can see a list of orders I made. |
| US11 | As a customer user, I can see a history of purchased products. |

Table 1. User stories

## 4.2 Domain Design

### 4.2.1 Step 1. Identify sub-domains based on key business processes

Based on the gathered user stories, multiple sub-domain is elicited: Product management, Order management, User management, and main service for users. The sub-domains are interconnected in a manner similar to a microservice architecture.

- Product management: Handling product information, categories, and inventory, such as listing available products.
- Order management: Handling customer orders, payments, and product shipping, such as getting the order information.
- Customer management: Handling user registration, authentication, and profiles.
- Main service: It uses APIs created in the above sub-domains to serve a wide range of usecase applications for end-users.

During this implementation, due to time constraints, the sub-domain will be limited to the Main Service, which will be responsible for handling other sub-domains. As a result, this Main Service will function as if it manages all tasks independently, relying on simpler procedures instead of making API calls to other sub-domains.

### 4.2.2 Step 2. Distill objects from user stories

Utilizing the obtained user stories, Table 2 showcases the derived entities, value objects, and aggregates. In the "Entity" column, the names of potential entities are displayed. The "Aggregate" column illustrates the entity's relationship within the aggregate, denoted by "many" or "one" to signify the relationship type. Additionally, the "Value Object" column specifies the particular values that require storage.

| **User story** | **Entity** | **Aggregate** | **Value Objects** |
|---|---|---|---|
| US1 | session, user | - | - |
| US2 | user | - | - |
| US3 | products | - | - |
| US4 | products | - | - |
| US5 | cart, cart_item, product | cart has: many cart_item | cart_item has: one product shopping_status 0: in progress 1: completed |
| US6 | Same as above | | |
| US7 | Same as above | | |
| US8 | Same as above | | |
| US9 | order, payment | Order one payment one cart (payment) method 0: cash 1: card | - |
| US10 | - | order | - |
| US11 | cart_item | - | - |

Table 2. List of distilled objects

Upon extracting the three categories of objects detailed in Table 2, these objects have been filled with the essential variables to satisfy the user stories, as depicted in Figure 12. Below the diagram, a comprehensive overview of these objects is provided. For variables, their respective data types are indicated within the right-side parenthesis. The relationships with associated objects are denoted: "one" indicates possession of a singular object, while "many" signifies an array of objects.

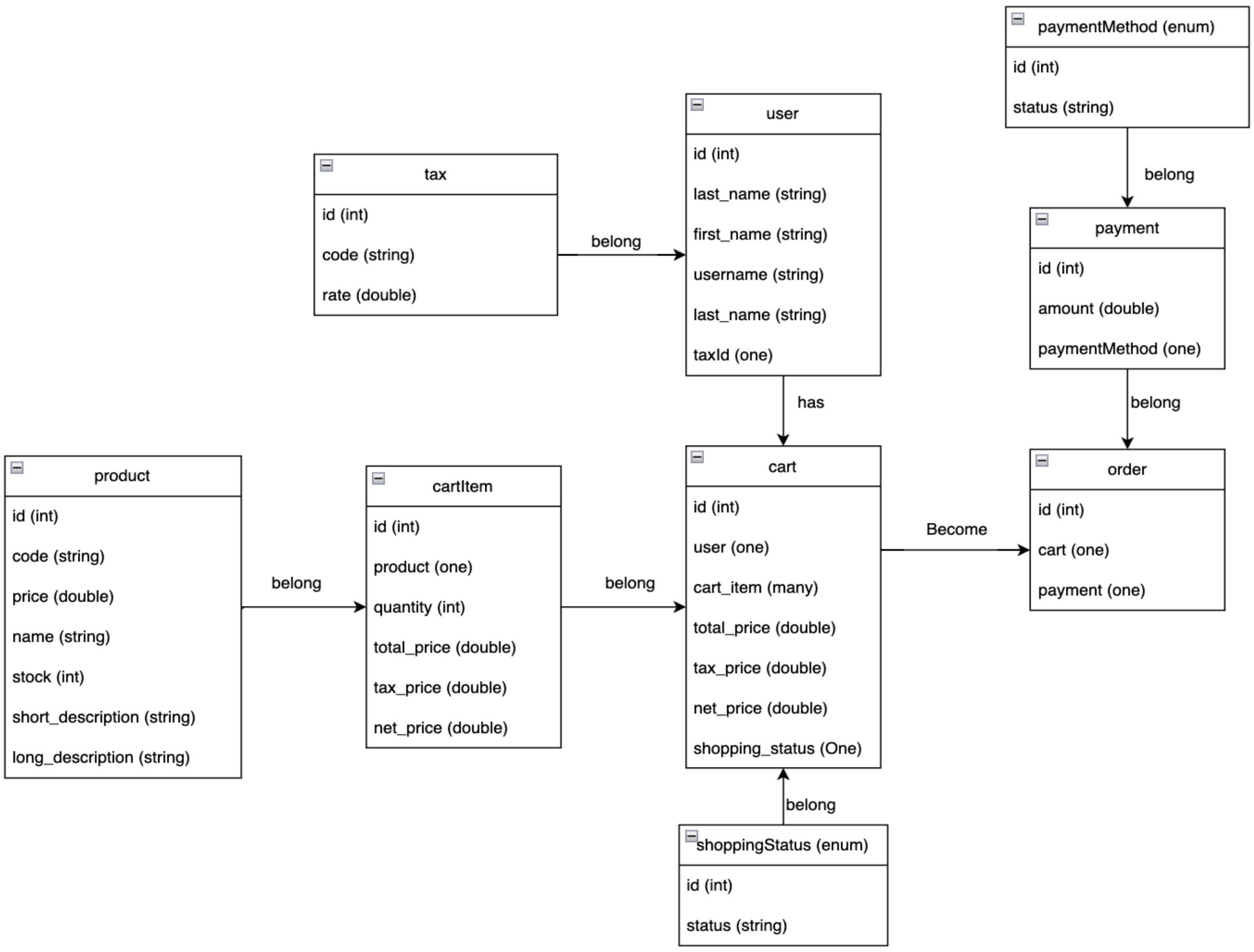


Figure 12. Graph of defined objects

### 4.2.3 Step 3. Examine business logic

This phase is intended to generate the business logic for each user story. Essentially, the business logic aligns with the intention of each API. Hence, Table 3 presents the list of business logic flows for each user story, constituting the API documentation. By default, the output consistently includes an error code.

| **User story** | **Input** | **Flow** | **Output** |
|---|---|---|---|
| US1 | username, password, firstName, lastName | Validate username if not taken | |
| US2 | username, password | 1.Get user object based on the username 2.Compare password. 3. Create session key, and put it in a session object, then store it. | sessionKey |
| US3 | productPageNum | List products based on the productPageNum | (array) product |
| US4 | code | Get product based on the code (product code) | product |
| US5 | sessionKey | 1.Validate session 2.Get the cart object based on the userId from session object. The state of this cart object needs to be “in progress”. 3. Get cartItem object based on the cartId and put them in the cart object. 4. Return the cart object with items in it. | cart |
| US6 | sessionKey, quantity, productId | 1.Validate session 2. Validate product ID 3. Get product based on the product ID 4. Get cart based on the user ID in session. 5. Check if the product is already added. a.If yes, take it into consideration to update the cart object. 6. Update the cart object and store it. | |
| US7 | sessionKey, productId | 1.Validate session 2.Get cart object based on the userId from session object. a.Return if cart doesn’t exist 3. Delete cartItem object. 4. Update cart object based on the deleting product(s). | |

Table 3. API documentation (Page 1)

| User story | Input | Flow | Output |
|---|---|---|---|
| US8 | sessionKey, productInfo (array) [ quantity productId] | 1.Validate session 2.Get cart object based on the userId from session object. a.If doesn't exist, create one 3. Delete all the items in the cart. 4. Loop cartItem based on the productInfo to add each cartItem object. While looping, calculate the costs for the cart object. 5.Update the cart object and store it. | |
| US9 | sessionKey, paymentMethod | 1.Validate session, 2.Get cart object based on the userId from session object and status is in progress. a.If doesn't exist, return error that cart object doesn't exist. 3.Create payment object and store it. 4.Update cart status to "comeplted", 5. Create order object and store it | order |
| US10 | sessionKey, orderId | 1.Validate session, 2.Get order object based on the userId from session object. a. If doesn't exist, return error that order object doesn't exist 3.Return order object that has payment and cart object in it. | order |
| US11 | sessionKey | 1.Validate session, 2.Get multiple cart objects based on the userId from session object. The status of those cart should be "completed". 3.Get cartItems based on the above car object IDs. 4.Put product object in the cartItem. 5.Return the array of cartItem. | (array) [cartItem] |

Table 4. API documentation (Page 2)

### 4.2.4 Step 4. Completion of each sub-domain

Since the implementation focuses only on a "Main Service," all the sub-domains are created.

## 4.3 Domain Implementation

### 4.3.1 Step 5: Convert defined objects into classes

Utilizing the information gained in Figure 12, which contains all the variables, the ensuing step involves distributing these classes into individual files. If a value object is specific to a single class, it should be included within the same file. Each file should have class definitions or constants relevant to the corresponding class. On the next page, Figure 13 shows the file structure after completing this step.

```
/
└─entity/
   ├─auth.go
   ├─cart.go
   ├─cartItem.go
   ├─order.go
   ├─payment.go
   ├─product.go
   ├─tax.go
   └─user.go
```

Figure 13. Entity directory

### 4.3.2 Step 6: Implementation of business logic

This stage defines the business logic. On the next page, Listing 1 shows the whole codes in the product.go file under the "usecase" directory. This file archives user story number 3 and 4: getting one specific product and listing all the products based on the page number. In the "usecase" directory, every file employs the factory design pattern to manage functions. Specifically, for product-related functions, an instance called "productService" is generated using the "newProductService" approach. Moreover, "newProductService" provides an interface, allowing the recipient object to encapsulate the associated business logic.

Below, two functions are detailed, both associated with the "productService." Each of these functions is designed to fulfill a specific user story. Within these functions, certain methods are invoked from external devices. In Listing 1, some functions are related to the database, so those functions are stored within the "interface" directory under "repository" layer, as demonstrated in Listing 2. If other external devices are used, other name of the directory should be created under the "interface" directory.

Listing 1. **Usecase directory**: usecase/product.go

```go
package usecase

import (
    "clean_architecture_with_ddd/internal/entity"
    "clean_architecture_with_ddd/internal/interface/repository"
)

type productService struct {
    repo repository.Repository
}

func NewProductService(repo repository.Repository) ProductUsecase {
    return &productService{
        repo: repo,
    }
}

type ProductUsecase interface {
    GetProduct(productID string) (*entity.Product, error)
    ListProductsByPage(page int) ([]*entity.Product, int, error)
}

func (s *productService) GetProduct(productID string) (*entity.Product, error) {
    product, err := s.repo.GetProductByCode(productID)
    if err != nil {
        return nil, err
    }
    return product, nil
}

func (s *productService) ListProductsByPage(page int) ([]*entity.Product, int, error) {
    products, err := s.repo.ListProductsByPageNum(page, entity.PerPage)
    if err != nil {
        return nil, -1, err
    }

    count, err := s.repo.GetProductCount()
    if err != nil {
        return nil, -1, err
    }

    return products, count, nil
}
```

Listing 2. **Interface directory**: interface/product.go

```go
package repository

import "clean_architecture_with_ddd/internal/entity"

type ProductRepository interface {
    GetProductByCode(code string) (*entity.Product, error)
    GetProductCount() (int, error)
    ListProductsByPageNum(pageNum int, perPage int) ([]*entity.Product, error)
}
```

After completing this step, the "usecase" and "interface" directories are completed as shown in Figure 14.

### 4.3.3 Step 7: Build controller directory

In this phase, APIs for input and output data are handled using the functions crafted within the "usecase" directory. Listing 3 serves as an illustrative example to guide the progression of this step. It portrays an operation where a user's current shopping cart is updated. This file is positioned within the "api" directory, housing the central functions dedicated to API management. The below list explains Listing 3.

- API layer also conducts the factory pattern that takes a specific interface from the "usecase" directory on Line 10.
- Since the existing code repository was using the Echo library [3], this repository also uses it on Line 24.
- Below is the flow of the data stream
  1. Retrieves data from the request body on Line 28.
  2. Since this user needed to be authenticated, the middleware function validates the user on Line 33. Any middleware functions are defined in the "middleware" directory under the "controller" directory.

```
│  /
│  └─ usecase/
│      ├─ auth.go
│      ├─ cart.go
│      ├─ cartItem.go
│      ├─ order.go
│      ├─ payment.go
│      ├─ product.go
│      └─ user.go
└─ /
   └─ interface/
       └─ repository/
           ├─ auth.go
           ├─ cart.go
           ├─ cartItem.go
           ├─ order.go
           ├─ payment.go
           ├─ product.go
           └─ user.go
```

Figure 14. Entity directory

3. Validate the input by passing a function defined in the "inputPort" directory on Line 40.
4. Pass the validated input to the function in the "usecase" directory to process on Line 47.
5. If the output needs to be modified, a function in the "presenter" directory should be implemented before returning a response on Line 50.

Listing 3. **controller directory**: controller/api/cartItem.go

```go
1 package api
2
3 import ...
4
5 type cartItemHandler struct {
6     usecase usecase.CartItemUsecase
7     auth    middleware.Auth
8 }
```

[3]https://github.com/labstack/echo

```go

func NewCartItemHandler(u usecase.CartItemUsecase, auth
    middleware.Auth) CartItemHandler {
    return &cartItemHandler{
        usecase: u,
        auth:    auth,
    }
}

type CartItemHandler interface {
    AddItemToCart(c echo.Context) error
    RemoveItemFromCart(c echo.Context) error
    UpdateCart(c echo.Context) error
    GetPurchasedProducts(c echo.Context) error
}

func (cih *cartItemHandler) UpdateCart(c echo.Context) error {

    // Retrieve input
    var body request.ListCartItem
    if err := c.Bind(&body); err != nil {
        return c.JSON(http.StatusBadRequest, "failed to bind
            the struct with the request body: "+err.Error())
    }

    // middleware validation
    userId, err := cih.auth.ValidateSession(c)
    if err != nil {
        return c.JSON(http.StatusBadRequest, err)
    }


    // validate input
    if err = inputPort.ListCartItem(body); err != nil {
        return c.JSON(http.StatusBadRequest, err)
    }

    // pass to usecase
    err = cih.usecase.UpdateItemsInCart(userId, body)
    if err != nil {
        return c.JSON(http.StatusInternalServerError, err)
    }

    return c.JSON(http.StatusOK, "cart is updated")
}
```

### 4.3.4 Step 8: Creation of infrastructure layer to write the implementation

The detailed implementation files for all functions outlined in the "interface" directory are generated in this phase. For the purpose of this demonstration, as only the database serves as an external device, solely a "repository" directory exists within the "infrastructure" directory. In instances where additional external devices are required, such as utilizing a third-party library, the interface directory must incorporate interfaces for the functions, while their implementations should be documented under the infrastructure directory.

### 4.3.5 Step 9: Create a cmd directory to build a server

This stage constructs all the instances that have been created with the factory pattern in the previous steps. Listing 4 is a core function of building a server.

1. The server creates an instance based on the external devices in Line 4. Since this project only deals with the database, only one instance is created.
2. Those external instances are passed to each instance for the "usecase" layer from lines 7 to 13. Output is an interface for the encapsulation of business logic
3. With a middleware instance, the created interfaces are passed to API handler instances from lines 19 to 25. They also return an interface for the encapsulation of business logic.

Listing 4. **cmd directory**: cmd/cmd.go

```
1  func buildServer(_ *config.Config, db *sqlx.DB)
      (api.AuthHandler, api.CartHandler, api.CartItemHandler,
      api.OrderHandler, api.PaymentHandler, api.ProductHandler,
      api.UserHandler) {
2
3      // Build external devices
4      repo := repository.NewRepo(db)
5
6      // Put them in each usecase instances
7      authUsecase := usecase.NewAuthService(repo)
8      cartUsecase := usecase.NewCartService(repo)
9      cartItemUsecase := usecase.NewCartItemService(repo)
10     orderUsecase := usecase.NewOrderService(repo)
11     paymentUsecase := usecase.NewPaymentService(repo)
12     productUsecase := usecase.NewProductService(repo)
13     userUsecase := usecase.NewUserService(repo)
14
```

```
    // controller middleware instances
    middlewareAuthUsecase := middleware.NewAuth(repo)

    // Create handlers for APIs
    authHandler := api.NewAuthHandler(authUsecase)
    cartHandler := api.NewCartHandler(cartUsecase, middlewareAuthUsecase)
    cartItemHandler := api.NewCartItemHandler(cartItemUsecase, middlewareAuthUsecase)
    orderHandler := api.NewOrderHandler(orderUsecase, middlewareAuthUsecase)
    paymentHandler := api.NewPaymentHandler(paymentUsecase, middlewareAuthUsecase)
    productHandler := api.NewProductHandler(productUsecase)
    userHandler := api.NewUserHandler(userUsecase)

    return authHandler, cartHandler, cartItemHandler, orderHandler, paymentHandler, productHandler, userHandler
}
```

### 4.3.6 Step 10. Completion of code repository for each sub-domain

Since there is only one sub-domain for this project, the process for the proposed architecture is completed. The below Figure 15 shows the final directory of the file structure.

```
/
├── cmd.go/
├── config/
├── database/
└── internal
    ├── controller/
    │   ├── api/
    │   ├── entity/
    │   ├── inputPort/
    │   ├── middleware/
    │   └── presenter/
    ├── entity
    ├── infrastructure
    ├── interface
    └── usecase
```

Figure 15. Final file structure

# 5 Validation

Up to this point, this paper has introduced and elaborated upon the proposed approach, detailing the systematic procedure for developing microservice architecture software. Each step of the approach has been comprehensively outlined, emphasizing its significance and underlying factors. This section serves to validate the proposed approach by answering the research questions.

## 5.1 Methodology of the validation

Following the creation of the application using the proposed approach, I explored diverse data collection methods to address the research questions. An online survey emerged as an optimal choice to gather feedback, as it offers subjective insights into the actual development experience through implementing code. Consequently, I opted to conduct a comparative analysis of two applications: one crafted through traditional software architecture using the MVC pattern and the other constructed employing the proposed approach.

**Participant selection:**
Among my current and previous colleagues, suitable participants were chosen from those qualified for the requirements, a back-end development in Golang for over two years. This criterion ensures that participants possess substantial expertise to evaluate the software architecture effectively.

The survey is structured around three distinct questions. Each of these questions comprises three tasks (ST) designed to meticulously assess various facets of the participants' experiences and perceptions:

**ST1: Code Explanation or Pseudo Code**
The first task serves to measure the participants' grasp of the question's intent and the accuracy of their responses. Prompting participants to explain or provide pseudo-code effectively measures their understanding and ability to interpret the question correctly. This understanding serves as a fundamental prerequisite, as it influences the validity of the subsequent tasks.

**ST2: Development Experience Narrative**
The second task searches into the participants' development experiences, offering a valuable comparative analysis between the two architectural approaches. This task stands at the heart of the validation process, providing nuanced insights that directly address the research inquiries.

**ST3: Time Investment Measurement**
The third task introduces an additional dimension to the development experience assessment by quantifying the time dedicated to each implementation or explanation. This quantitative perspective offers an alternative lens through which to evaluate the research questions.

Survey questions (SQ) are based on research questions (RQ). **SQ1**, **SQ2**, **SQ3** are derived from **RQ1**, **RQ2**, and **RQ3** respectively.

**Research questions:**
**RQ1:** To what extent does the incorporation of Clean Architecture and Domain-Driven Design (DDD) principles enhance the understandability of the software architecture?

**RQ2:** How does the incorporation of Clean Architecture and Domain-Driven Design (DDD) principles facilitate the refactoring process of the software architecture in terms of reducing technical debt and improving code maintainability?

**RQ3:** In what manner does the incorporation of Clean Architecture and Domain-Driven Design (DDD) influence the extensibility of the software architecture over the system's lifecycle?

Below is the description and motivation of the survey questions.

- **SQ1**: The first question is an open-ended inquiry that involves describing the data flow of an API, aiming to evaluate its readability and comprehensibility. Participants are presented with specific application processes from an end-user viewpoint and are then requested to describe the sequence from a developer's perspective.

- **SQ2**: The subsequent question prompts participants to implement a pseudo-code focused on refactoring existing code based on additional user stories designed to assess the performance of the refactoring process.
- **SQ3**: The third question also prompts participants to implement a pseudo-code to extend a feature by integrating a third-party library, aiming to gauge the extensibility aspects. Participants are provided with a clear description of the function to be incorporated from the third-party library and its implementation.

Google Forms [4], a digital tool provided by Google that enables users to create customized surveys and questionnaires for gathering information and feedback, was used to conduct this survey. All responses obtained from the survey are transcribed in order to prepare them for subsequent analysis. Once this transcription is complete, the original responses will be securely deleted to ensure data privacy and confidentiality. The flow of the interview is as follows, and the survey content is displayed in Appendix 1.

1. Survey form is sent to each participant.
2. Participants read the consent and proceeded with the survey as agreed. The form of the consent is displayed in Appendix 1.
3. Participants read the description of the survey, including API documentation and User Stories.
4. Participants read a question and ask questions if any clarification is needed. Then, start measuring time and answering the question.
5. After answering all three questions, submit the answers.

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

## 5.2 Analysis of the Response

**Data analysis process:**
The responses collected from the survey have been transcribed to enable thorough analysis. Participants' responses to each question are carefully examined to extract key features.

- **ST1 (Trustfulness):** A trustfulness (**ST1**) is assigned to each response. A lower value suggests that the participant might not have fully grasped the question, reducing their response's significance in addressing research inquiries. This score is determined based on factors such as correct function name mentions for **SQ1** or accurate placement of pseudo-code in the intended file for **SQ2** and **SQ3**.
- **ST2 (Key insight):** Extracting meaningful insights from responses for each research question is facilitated through **ST2**. This involves identifying and formulating succinct sentences containing key terms that contribute to addressing each research question.
- **ST3 (Time spent):** The effort required to answer each question is quantified using **ST3**. If the ratio of time spent on a question for each architecture is approximately 1, it indicates that both architectures necessitate similar efforts to complete the task.

## 5.3 Results

This section presents the answers to the research questions based on the survey results.

**RQ1:**
Upon analyzing the responses for RQ1, Table 5 reveals that all participants provided correct answers according to **ST1**. Additionally, **ST3** demonstrates that both architectures demanded an equal amount of effort to complete the respective question. The insights derived from **ST2** indicate that the proposed architecture introduces complexities in logic when compared to the traditional architecture. However, it is noteworthy that the clear separation of layers was cited as contributing to better comprehension of the structure. The cumulative response implies that the proposed architecture enhances understandability through the distinct separation of logic, although this might simultaneously lead to structural complexity and reduced overall understandability.

| **Participant ID** | **Truthfulness** | **Key insight** | **Time spent (traditional, new)** |
|---|---|---|---|
| P1 | 100% | Both were equally easy to understand. Proposed architecture has clear separation of logic, and traditional one has simple structure. | 5min, 6min |
| P2 | 100% | Because of too many layer separations in the proposed architecture, traditional architecture was easier to understand. | 5min, 5min |
| P3 | 100% | Both architectures had the same understanding. | 3min, 3min |
| P4 | 100% | Due to many interface separations in the proposed architecture, traditional architecture was easier to understand. | 5min, 5min |

Table 5. Response from **SQ1**

**RQ2:**

Evaluation of the responses for RQ2 is outlined in Table 6. Here, most participants provided accurate answers based on **ST1**. Moreover, **ST3** indicates that traditional architectures required slightly less effort to address this question. Insights gathered from **ST2** underline that the proposed architecture introduces complexities in logic, primarily due to increased usage of interfaces. Notably, the proposed architecture contains more extensive code, thereby demanding more effort. However, some responses also highlight that the proposed architecture's benefit lies in clear logic separation, especially for larger projects. The overall response suggests that traditional architecture boasts slightly superior refactoring features, though this result could vary based on project scale.

| **Participant ID** | **Truthfulness** | **Key insight** | **Time spent (traditional, new)** |
|---|---|---|---|
| P1 | 90% | This size of application prefers traditional architecture since the amount of code is less. | 15min, 16min |
| P2 | 90% | Both architectures had the same easiness of refactoring. | 14min, 16min |
| P3 | 100% | Both architectures had the same ease of refactoring, but the separation of layers felt easier to refactor if the project size was larger. | 12min, 14min |
| P4 | 100% | Proposed architecture has too many interfaces that it took much more time for implementation. | 12min, 13min |

Table 6. Response from **SQ2**

**RQ3:**

For RQ3, the survey responses are examined in Table 7. Notably, some participants did not accurately answer according to **ST1**, potentially due to limited understanding of the proposed architecture. Particularly, participant P2 misplaced the pseudo-code within the wrong layer. **ST3** indicates that traditional architectures required significantly less effort to tackle this question. Analysis of **ST2** reveals that the proposed architecture results in logic complexities compared to the traditional counterpart, attributed to the presence of numerous directories. Conversely, several comments indicated that the proposed architecture benefits extensibility by facilitating clear separation of logic, especially in handling third-party libraries. The overall response indicates that traditional architecture showcases superior extensibility, primarily due to project scale, while the proposed architecture benefits from enhanced separation of third-party libraries for better extensibility through improved understanding of each layer's functionality.

| **Participant ID** | **Truthfulness** | **Key insight** | **Time spent (traditional, new)** |
|---|---|---|---|
| P1 | 90% | Traditional architecture was easier to implement, but old architecture would be more difficult to maintain as the number third-party library increases. | 21min, 25min |
| P2 | 50% | Traditional architecture was easier to implement because of less interfaces. | 22min, 23min |
| P3 | 100% | Proposed architecture seemed more organized since it has the separation of third-party library in. But considering easiness of implementation, old architecture was easier. | 15 min, 18 min |
| P4 | 90% | Traditional architecture was easier to implement since I can put the logic in the controller layer. | 23 min, 18 min |

Table 7. Response from **SQ3**

## 5.4 Threats to validity

The evaluation conveyed above is potentially affected by the following threats to validity:

1. **Limitation of Participant Count**: The survey was conducted with a limited participant pool consisting of four individuals. It's crucial to acknowledge that the outcomes and validity of the results could exhibit variations with an increase in the survey's participant count. To counteract this limitation, the survey was meticulously designed to focus on practical implementation. This approach aimed to gather detailed data on time invested and development experiences, thereby enhancing the depth and reliability of the feedback obtained.

2. **Scale of Application and Perceived Benefits**: A noteworthy observation from the survey responses underscores the challenge posed by the small-scale nature of the application. As one respondent articulated, the application's limited complexity hindered the discernment of certain benefits associated with the proposed approach. It is important to emphasize that the application was conceptualized to serve as a foundation for a large-scale software service utilizing microservice architecture. Consequently, certain advantages, such as the prominent utilization of interfaces, might not have been fully realized due to the inherent simplicity of the application's scope.

By acknowledging these limitations and utilizing a combination of meticulous survey design and an understanding of the application's intended scale, efforts were made to obtain valuable insights that contribute to a more comprehensive evaluation of the proposed approach's effectiveness.

## 5.5 Future work

For future possibilities, several ideas present themselves for extending and enhancing the approach proposed in this study.

**Scaling Up the Project for Validation**: A key consideration lies in augmenting the scale of the project, as acknowledged in the 5.4 section. A forthcoming step entails conducting another implementation encompassing more than three sub-domains. The queries posed to participants will be refined and expanded to strengthen the validation process. For instance, respondents will be prompted to perform refactoring for an API that integrates multiple APIs. This augmentation seeks to yield a more diverse array of feedback, fostering a heightened precision in the validation process.

**Integration with Test-Driven Development (TDD)**: The proposed approach stands hovered to harmonize with and leverage the potency of Test-Driven Development (TDD). The inherent decoupling of each layer offers a seamless integration opportunity. TDD operates as a valuable asset, minimizing uncertainty through expedited debugging, providing clearer insights into implementation progress from test coverage and success ratios, and instilling confidence in the comprehensiveness of covered functions Martin (2007).

By executing these factors, the proposed approach has the potential to evolve into a more versatile and robust software architecture process capable of accommodating larger and more intricate software projects while concurrently capitalizing on the gains offered by Test-Driven Development.

# 6 Conclusion

The primary objective of this thesis was to present an innovative method for constructing a microservice architecture through the integration of clean architecture principles and Domain-Driven Design (DDD). The journey began by elucidating essential aspects of relevant concepts, including a comparative exploration of microservice architecture against other system architectures, a comprehensive understanding of the pivotal attributes of clean architecture principles, and an in-depth grasp of Domain-Driven Design principles.

Subsequently, the proposed approach was unveiled, characterized by the distilled essential elements derived from clean architecture principles and Domain-Driven Design. Each step of this approach was thoroughly outlined, traversing the entire process from start to completion. This comprehensive exposition served to guide developers through the sequential stages of creating a microservice architecture.

Furthermore, to manifest the viability of the proposed approach, a proof of concept was meticulously actualized, with each step exhaustively described. The developed software service underwent validation by means of a comparative analysis with traditional software architecture. The validation outcomes illuminated the advantages of the proposed architecture and its viable applications and potential use cases.

In conclusion, this thesis contributes a pragmatic approach that empowers developers to translate a set of user stories into a functional software service. The approach not only serves as a valuable tool for crafting new services but also serves as a valuable reference for restructuring existing systems, enhancing the organization of business logic, and refining file structures. By combining clean architecture principles and Domain-Driven Design, this thesis offers a comprehensive framework that resonates across various software development scenarios.

## References

R. C. Martin, *Clean architecture*, Prentice Hall, 2017.

H. G. Koller, *M.Sc. thesis*, University of Oslo, 2016.

R. Nunkesser, *Hamm-Lippstadt University of Applied Sciences*, 2021.

V. Lakhai, O. Kuzmych and M. Seniv, 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT), 2022, pp. 474–477.

A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera and A. Sadovykh, *Science and Engineering. Springer*, 2020.

M. Kalske, N. Mäkitalo and T. Mikkonen, Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17, 2018, pp. 32–47.

F. Ponce, G. Márquez and H. Astudillo, 2019 38th International Conference of the Chilean Computer Science Society (SCCC), 2019, pp. 1–7.

M. Turis, *Ph.D. thesis*, Masarykova univerzita, Fakulta informatiky, 2019.

R. Chen, S. Li and Z. Li, 2017 24th Asia-Pacific Software Engineering Conference (APSEC), 2017, pp. 466–475.

L. De Lauretis, 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019, pp. 93–96.

L. Weerasinghe and I. Perera, 2021 International Research Conference on Smart Computing and Systems Engineering (SCSE), 2021, pp. 137–144.

T. Erl, *SOA design patterns (paperback)*, Pearson Education, 2008.

D. Sprott and L. Wilkes, *The Architecture Journal*, 2004, **1**, 10–17.

P. Di Francesco, P. Lago and I. Malavolta, *Journal of Systems and Software*, 2019, **150**, 77–97.

P. Clarke, R. O'Connor and P. Elger, *J. Softw. Evol. Proc. 2017, e1866*, 2017.

D. Shadija, M. Rezai and R. Hill, 2017 23rd International Conference on Automation and Computing (ICAC), 2017, pp. 1–6.

P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov, *IEEE Software*, 2018, **35**, 24–35.

C. Pahl, A. Brogi, J. Soldani and P. Jamshidi, *IEEE Transactions on Cloud Computing*, 2017, **7**, 677–692.

P. Bengtsson and J. Bosch, Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090), 1999, pp. 139–147.

V. K. Madasu, T. V. S. N. Venna and T. Eltaeib, *Journal of Multidisciplinary Engineering Science and Technology (JMEST)*, 2015, **2**, 3159–0040.

E. Chebanyuk and K. Markov, 2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2016, pp. 435–441.

R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C# (Robert C. Martin)*, Prentice Hall PTR, 2006.

P. Aguilar and L. Figueira, I Workshop de Tecnologia da Fatec Ribeirão Preto, 2020.

F. Rademacher, J. Sorgalla and S. Sachweh, *IEEE Software*, 2018, **35**, 36–43.

P. Merson and J. Yoder, 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), 2020, pp. 7–8.

Y. Xie, X. Zhou, H. Xie, G. Li and Y. Tao, 2018 IEEE 3rd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), 2018, pp. 887–893.

H. Vural and M. Koyuncu, *IEEE Access*, 2021, **9**, 32721–32733.

S. Kapferer and O. Zimmermann, MODELSWARD, 2020, pp. 299–306.

F. Rademacher, S. Sachweh and A. Zündorf, 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 229–236.

C. Chen, C. Dong, J. Cai and X. Cheng, Journal of Physics: Conference Series, 2019, p. 052028.

E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Pearson Deutschland GmbH, 1995.

R. H. Steinegger, P. Giessler, B. Hippchen and S. Abeck, The Third Int. Conf. on Advances and Trends in Software Engineering, 2017.

D. Sanchez, A. E. Rojas and H. Florez, *IAENG International Journal of Computer Science*, 2022, **49**, 270–278.

M. Fowler, *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*, Addison-Wesley, 2012.

P. Aguilar and L. Figueira, I Workshop de Tecnologia da Fatec Ribeirão Preto, 2020.

P. MALINA, *M.Sc. thesis*, BRNO UNIVERSITY OF TECHNOLOGY, 2016.

R. C. Martin, *Ieee Software*, 2007.

# 7 Appendix

## 7.1 Survey Consent

**Study title:** An approach for Designing Microservice-Based Applications using a Domain-Driven Design Approach and Clean Architecture Principles
**Researcher:** Daichi Ando
**Supervisor:** Mohamad Gharib

This study is being conducted by Daichi Ando from the University of Tartu for the master thesis in computer science department. It aims to validate the two software architectures: proposed architecture and traditional architectures. This validation is utilized to answer research questions in this master thesis.

**Participation requirements:** To be eligible to participate, a person has to be eighteen or older and should have over two years of back-end development experience in Golang programming language.

**Voluntary Participation:** Participating in this study may not benefit you directly, but it will help us providing some insights for the master thesis.

**Inquires about the Study:** If you have any comments or concerns about the study in anytime, please contact the Researcher (daichi@ut.ee).

**Privacy:** To keep the privacy confidential for the participants' identities, following procedure is carried out. Your response will be transcribed to formed information to answer research questions, and eventually deleted after the completion of thesis.

**By completing this survey, you are consenting to participate in this study.**

## 7.2 Survey Content

This is a survey to see the performance of the new and traditional software architecture. Some questions ask you to explain in text or write pseudo-code. For implementation, please fork the repository, create a new branch for each implementation question, and implement. Because this questionnaire asks you for the URL of the each branch you implemented. Kindly record the time taken for each question's response (they serve as a data source for comparing the two software architectures)

**Survey description**

This project provides APIs for e-commerce applications/websites. Traditional software architecture uses MVC (Model View Controller), but it doesn't have a View component since this project only focuses on backend development. New software architecture is based on clean architecture.

This is the link [5] that you can see the user stories and brief API documentation

Traditional software architecture (MVC) [6]
New software architecture [7]
Prerequisite:

- You have installed Go version 1.20
- You have installed MySQL locally or you run MySQL docker container.

**Q1. Understandability and Readability**
For each traditional repository [6] and new architecture repository [7], please answer the following three questions. Note: keep in mind that the third question requires you to measure the time to complete the first question.

1. Explain the flow of completing shopping. Completing shopping meaning:

   1. User adds or updates his shopping cart. Such as added a Product code P001 for 2 quantities.
   2. The user makes a payment by choosing "cash" or "creditCard".
   3. Shopping is completed. Now, user's shopping cart is empty.

Please list which functions the input is passed and what are those functions intend to do.

2. Explain your experience of understanding the code. Which one was easier to understand and why?

3. Write how long did it take for you to complete this task. (Example: traditional one for m minutes and new one for n minutes)

**Q2. Refactoring Feasibility**
For each traditional repository [6] and new architecture repository [7], please answer the following three questions. Note: keep in mind that the third question requires you to

[5] attached as an extra file named `api_documentation_with_user_stories`
[6] attached as a extra folder named `traditional_architecture`
[7] attached as the extra folder named `proposed_architecture`

measure the time to complete the first question.

1. There are some changes in business requirements; code needs to be refactored. Please refactor the code by writing pseudo-code to satisfy the blow user story. (Later questionnaire will ask you for the URL for the repository)

- As a user, I can create an account as a normal user so that I can see normal products.
- As a user, I can update my profile as a special user so that I can see exclusive products.

2. Explain your experience of refactoring the code. Which one was easier to refactor and why?

3. Write how long did it take for you to complete this task. (Example: traditional one for m minutes and new one for n minutes)

**Q3. Extensibility**
For each traditional repository [6] and new architecture repository [7], please answer the following three questions. Note: keep in mind that the third question requires you to measure the time to complete the first question.

1. A project manager wants to integrate a third-party library to store customer information. Please write pseudo-code using library api-go-wrapper [8] so that the system can store customer data (last name and first name) in the remote database when a new user is created.

How to use "api-go-wrapper" library to store new customer data:

- Create a demo account from this link [9]
- A client code, username, and password are required to establish a client [10].
- Use a 'SaveCustomerBulk' function [11] from CustomerManager to create a customer in the remote database.

[8]https://github.com/erply/api-go-wrapper

[9]https://login.erply.com/demo-account

[10]https://github.com/erply/api-go-wrapper/blob/bc89d13e22c4fffbd2d17bf8c174e93daf89d5bf/examples/example.goL20

[11]https://github.com/erply/api-go-wrapper/blob/bc89d13e22c4fffbd2d17bf8c174e93daf89d5bf/examples/customers/main.goL1
L133C22

2. Explain your experience of extending the code. Which one was easier to extend and why? If you add other external libraries, which software architecture is more maintainable?

3. Write how long did it take for you to complete this task. (Example: traditional one for m minutes and new one for n minutes)

# License

## Non-exclusive licence to reproduce thesis and make thesis public