Instituto Superior de
Engenharia do Porto

2023

ACME

SOFTWARE ARCHITECTURAL DOCUMENTATION

Simão Santos (1230212@isep.ipp.pt)

Rui Neto (1230211@isep.ipp.pt)

Content

Figures

Tables

Introduction

This project was undertaken as part of the curriculum for the ARQSOFT course within the master’s program in Software Engineering. The primary objective was to devise a new architectural solution for the ACME application, which is a REST-oriented system featuring endpoints for managing:

- Products
- Reviews (pertaining to a Product)
- Votes (within a Review of a Product)

Despite its functional capabilities, it is essential to recognize that this application was originally built upon a less-than-ideal architectural foundation, deviating from fundamental design principles.

In the upcoming sections of this report, we'll explore the current architectural issues and the chances for improvement they present. We'll take a close look at how these deviations affect the application's extensibility. This analysis will help us identify the key areas needing attention.

Moreover, this report will explain the practical decisions we've taken and the architectural designs we've chosen. We've employed the ADD methodology to guide our enhancements and refinements for the application. This structured approach ensures a systematic and efficient way of making improvements without unnecessary complexity.

System-as-is

System-as-is documentation is a strategic process management approach that involves the identification and evaluation of a system's existing state. This process was accomplished through reverse engineering, which entails a comprehensive analysis of the codebase. The outcomes of this reverse engineering endeavour are meticulously documented in the following diagrams.

4 + 1 Architectural View Model

Due to the complexity of software, detailing its architecture from a single perspective is challenging. Therefore, we will employ the 4 + 1 architectural view model, which allows us to describe software systems' architecture design through multiple concurrent viewpoints. The model has four views: logical, development, process, and physical. In addition, selected use cases or scenarios are utilized as the "plus one" view to show the design. [1]

- **Logical View**: The logical view is concerned with the system’s functionality as it pertains to end-users. Class diagrams and state diagrams are examples of UML diagrams that are used to depict the logical view. [1]

- **Process View**: The process view focuses on the system’s run-time behaviour and deals with the system’s dynamic elements. It explains the system processes and how they communicate. The sequence diagram and activity diagram are UML diagrams that can be used to describe a process view. [1]

- **Implementation View**: The implementation view depicts a system from the standpoint of a programmer and is concerned with software administration. The implementation view is another name for this view. It describes system components using the UML Component diagram. The Package diagram is one of the UML diagrams used to depict the development view. [1]

- **Physical View:** The physical view portrays the system from the perspective of a system engineer. The physical layer, it is concerned with the topology of software components as well as the physical connections between these components. The deployment view is another name for this view. The deployment diagram is one of the UML diagrams used to depict the physical perspective. [1]

- **Scenarios:** A small number of use cases, or scenarios, that become the fifth view, are used to illustrate the description of architecture. Sequences of interactions between objects and processes are described in the scenarios. The use case view is another name for this view. [1]

In addition to this model, we must specify in which level we are representing our views. This levels hardly depend on the granularity of the system we are specifying.

Higher the level -> higher the granularity.

It's important to keep in mind the following graph. We need to define what's the view we are documenting and in which level we are representing our system.

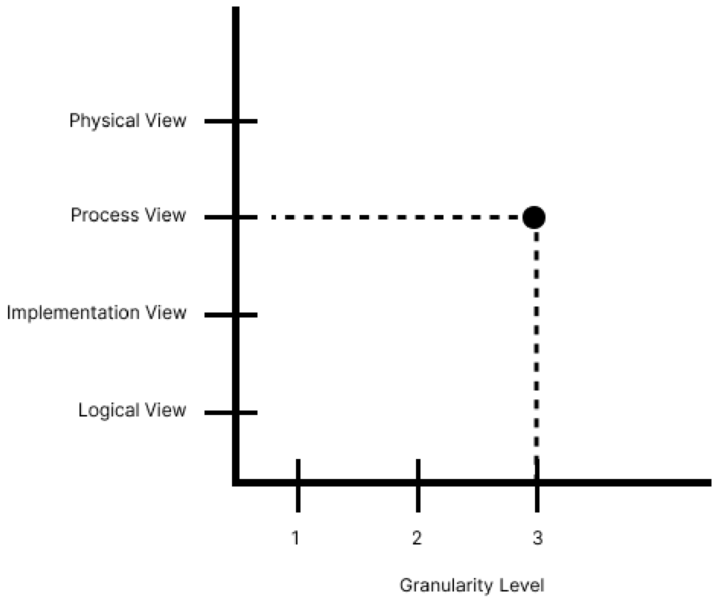


Figure 1- Graph of view and level of representation

Use Cases Diagram

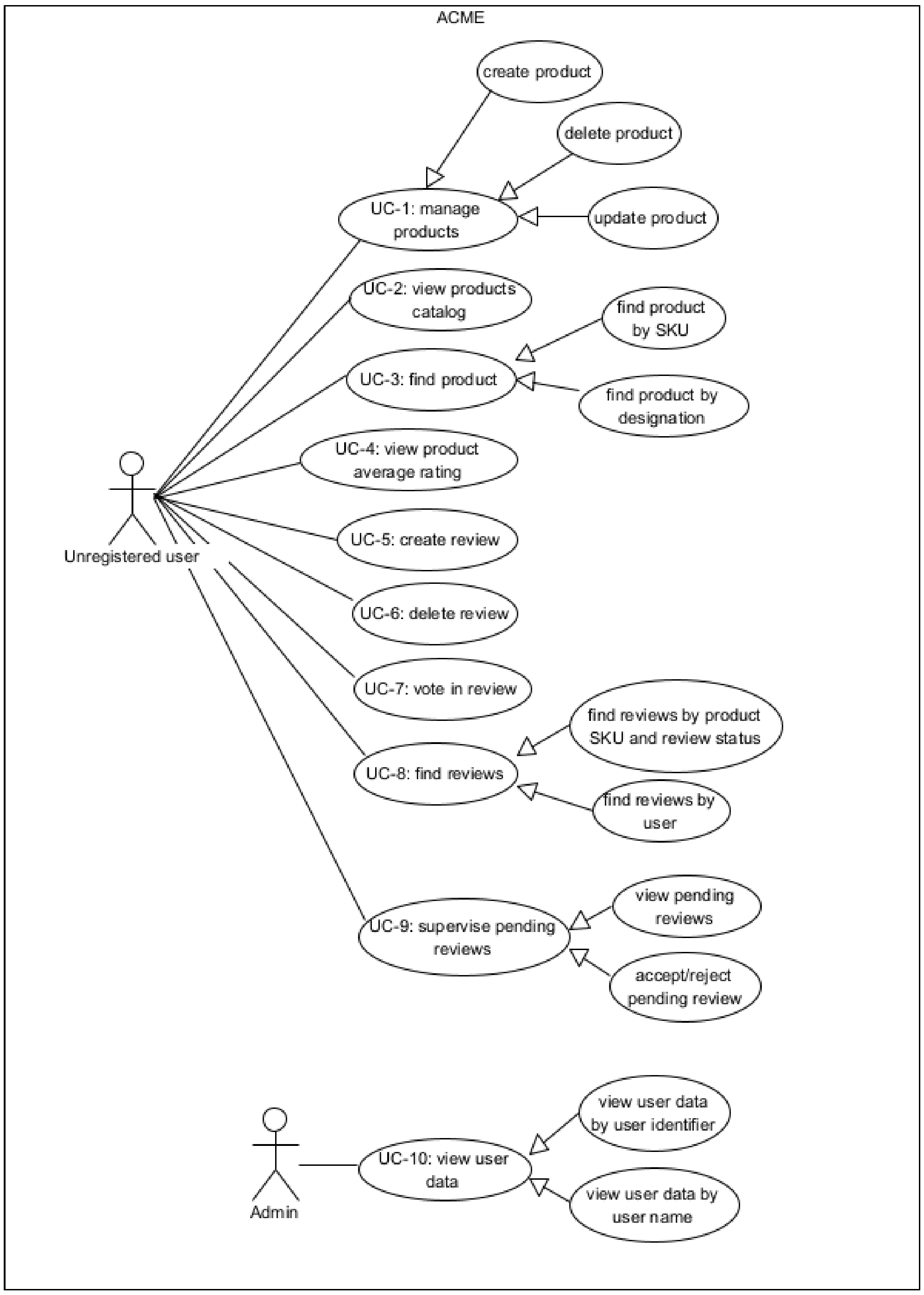


Figure 2- Use cases diagram

Logical view – level 1

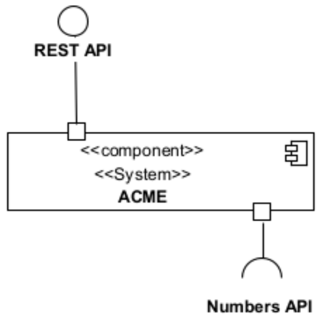


Figure 3- Logical view- level 1

Logical view – level 2

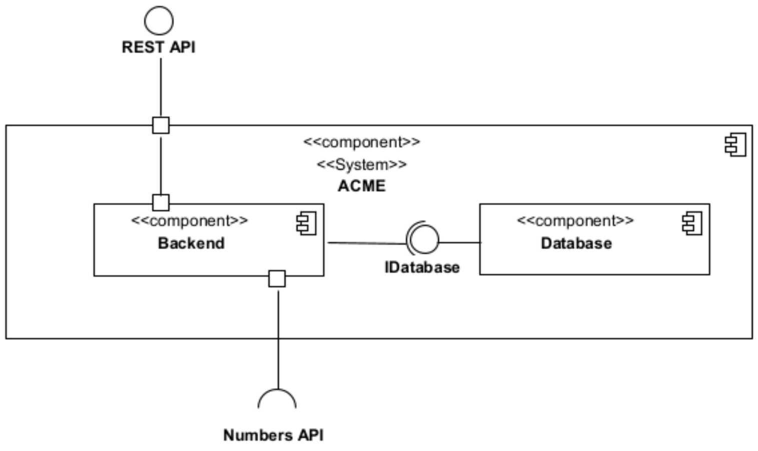


Figure 4- Logical view- level 2

Logical view – level 3

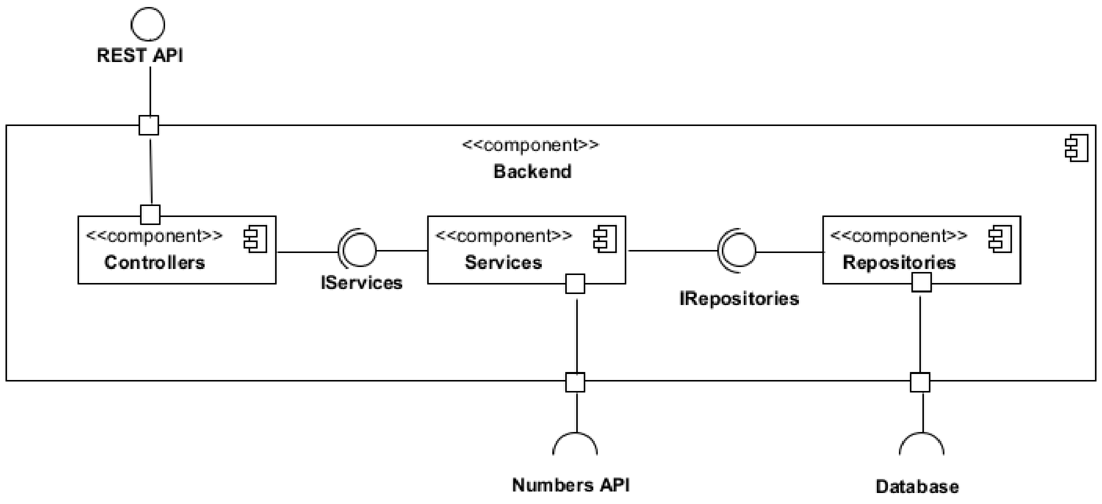


Figure 5- Logical view- level 3

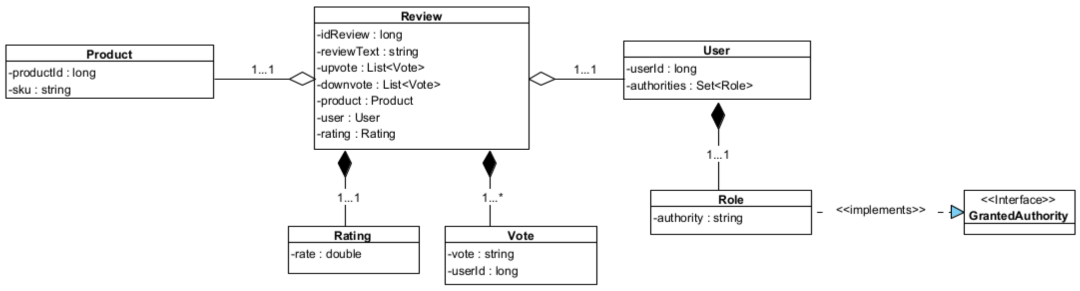


Figure 6- Logical View- level 3- class diagram

Implementation view – level 3

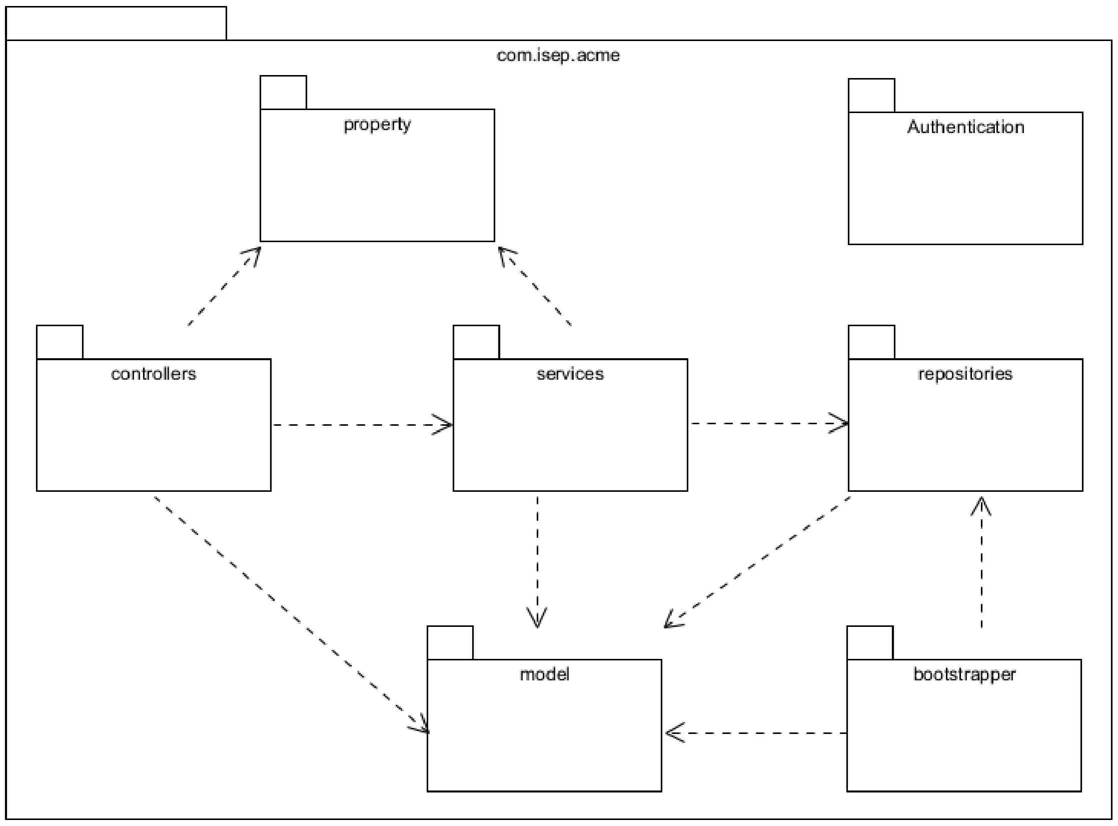


Figure 7- Implementation view- level 3

Process View - Create product

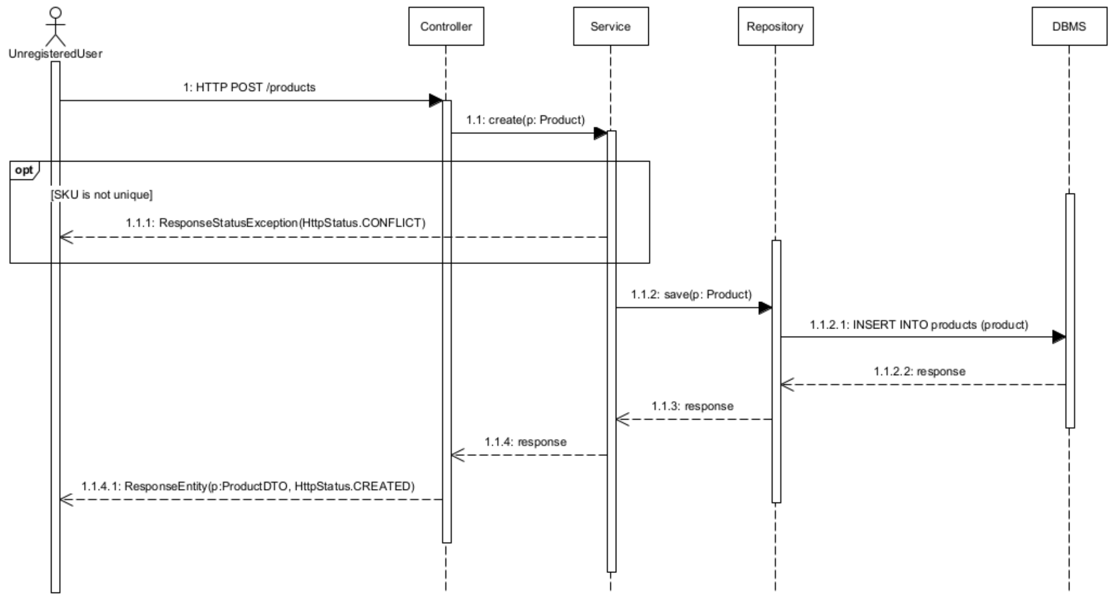


Figure 8- Process View - Create product

Process View - Create review

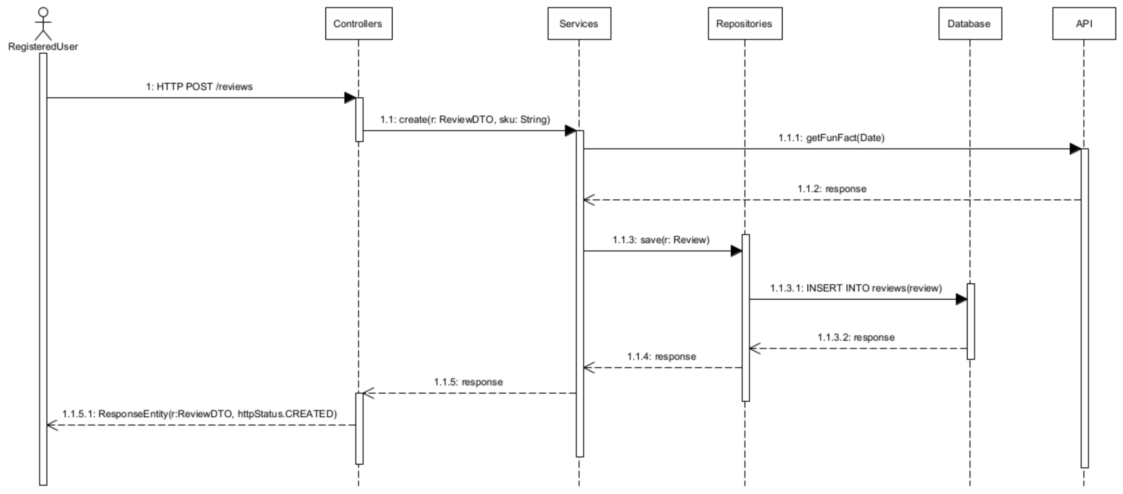


Figure 9- Process View - Create review

Process View - Create vote

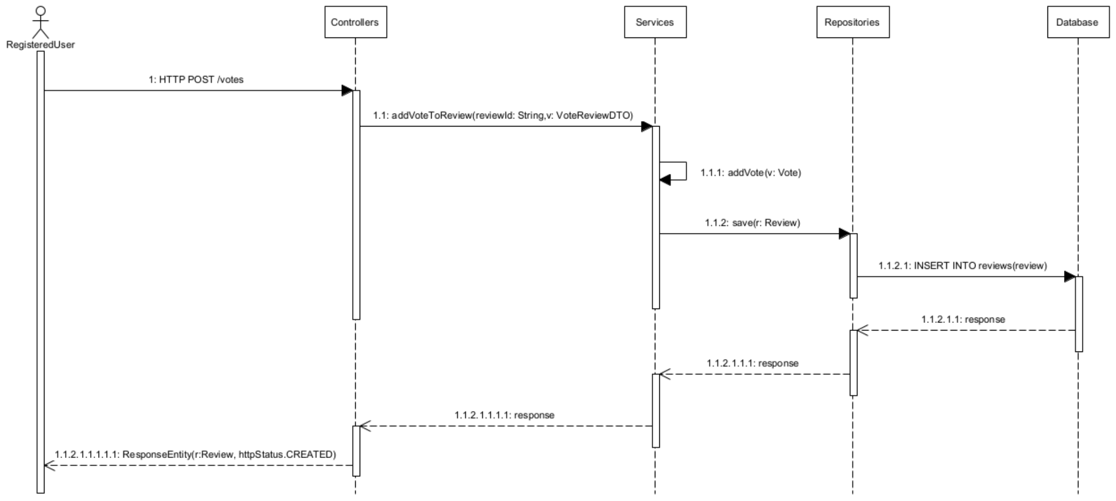


Figure 10- Process View - Create vote

Physical View

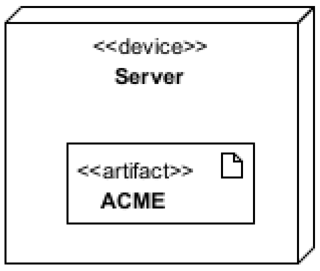


Figure 11- Physical view

Identified problems

- The software faces authorization issues due to inadequately implemented security measures, potentially leading to vulnerabilities and unauthorized access.
- The absence of a clear distinction between domain logic and persistence layers violates the Single Responsibility Principle (SRP). This integration complexity can hinder maintenance and scalability.
- Excessive coupling within components, particularly between the persistence layer and external APIs accessed via HTTP requests, restricts the system's extensibility. This tight integration makes it difficult to modify or expand functionalities without affecting other parts of the application.
- The package organization lacks coherence, with classes from diverse areas such as services, repositories, and interfaces being grouped together. This lack of structuring based on functional modules hampers code navigation and makes it challenging to locate specific components.
- Poorly chosen names for classes and packages reduce code readability and maintainability.
- The codebase exhibits poor quality, characterized by excessive commented-out code and inconsistent indentation.
- Lack of user inputs validation.
- Poorly handled errors, including generic error messages.
- Lack of proper validation and security checks during file uploads.

System-to-be

This section will be dedicated to the new ACME version.

Use Cases

The updated use case diagrams now explicitly incorporate two additional roles: 'Registered User' and 'Moderator,' alongside a new use case labelled 'UC-11.' In the initial ACME version, while authentication was implemented, authorization issues surfaced due to implementation errors.

These problems led to the incorrect belief that the system only had two roles, as depicted in the previous diagrams. Upon a thorough analysis of the implementation code through a reverse engineering process, it became apparent that the developers originally intended to include two more roles: 'Moderator' and 'Registered User.'

This revelation sheds light on the correct role structure within the system, resolving previous ambiguities and paving the way for improved authorization processes.

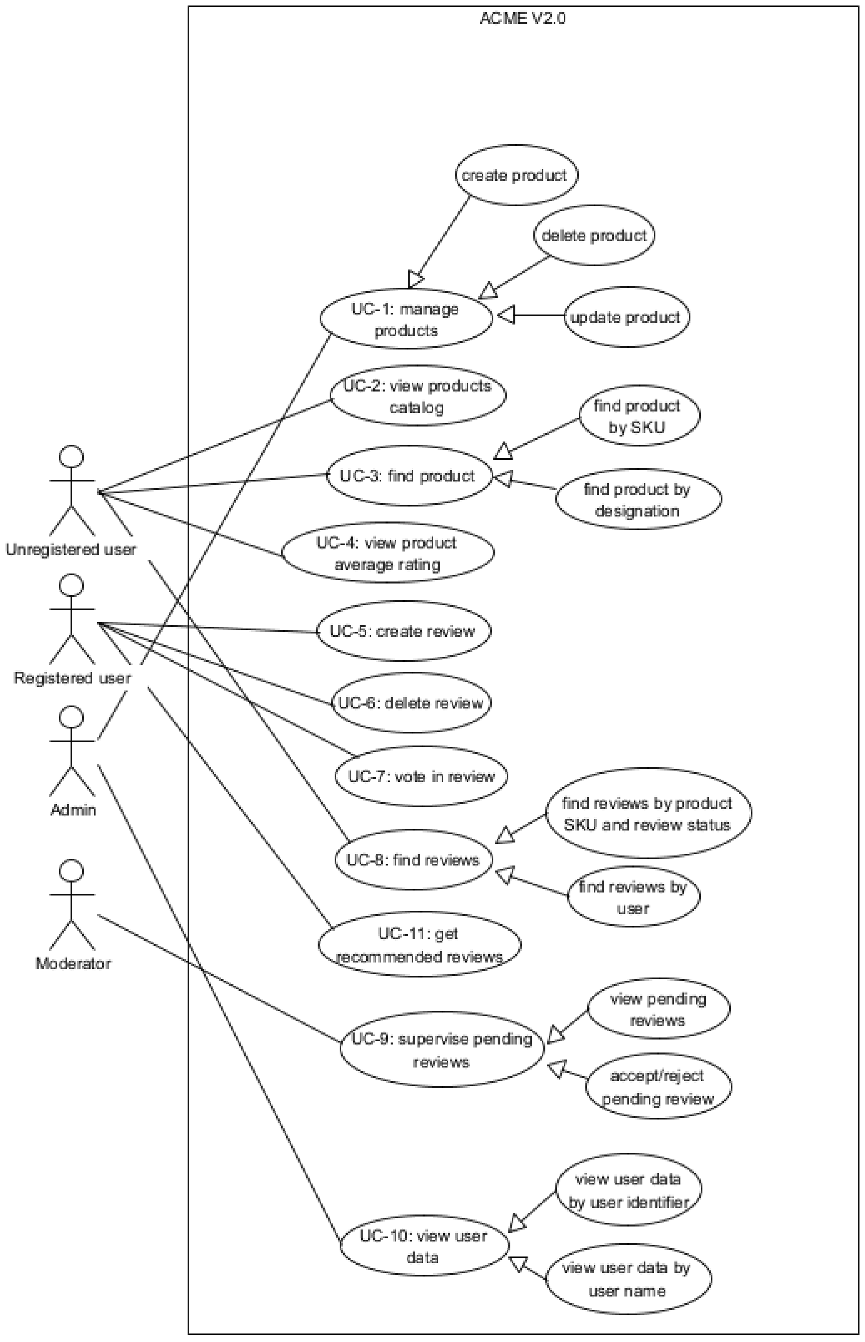


Figure 12- Use case diagram

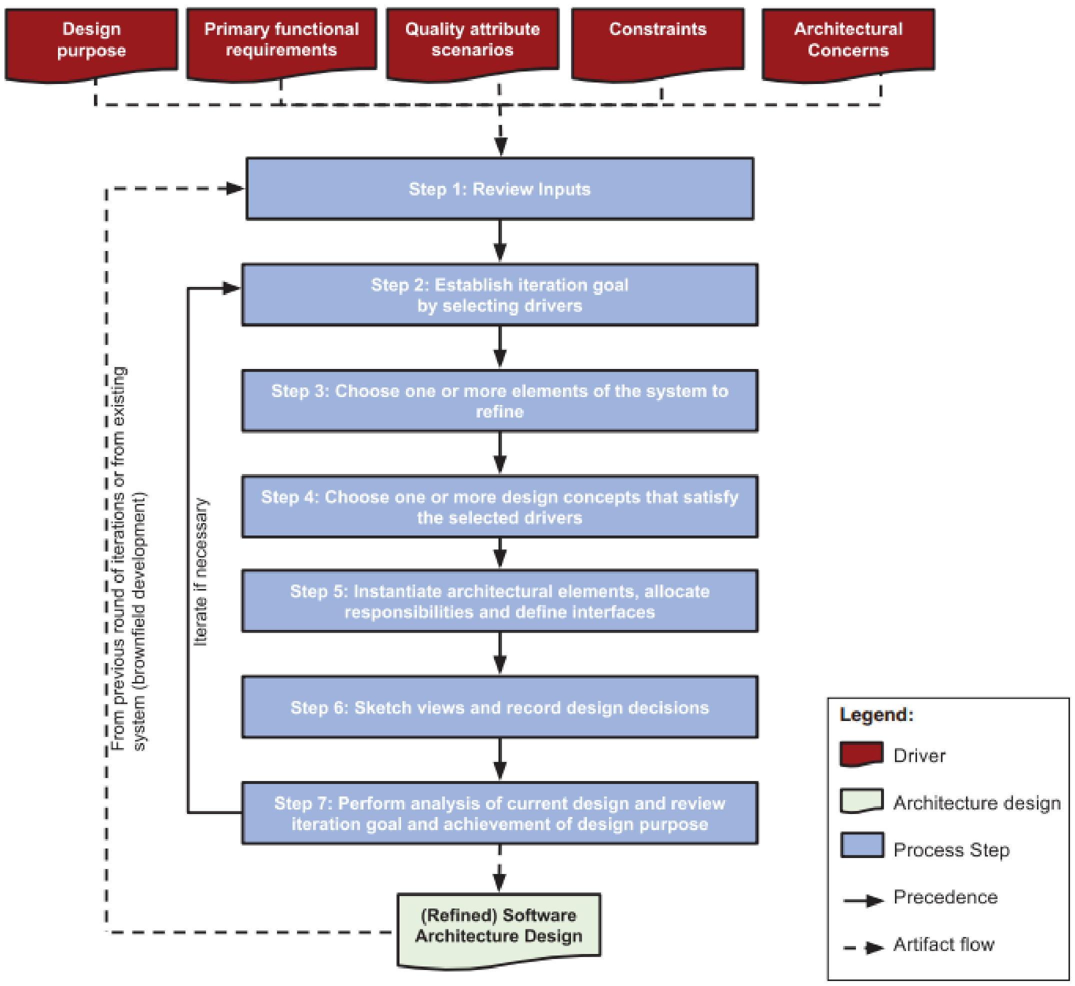
Attribute-driven design

To achieve the goals outlined in the project assignment, a comprehensive architectural design process was undertaken, employing the Attribute-Driven Design (ADD) methodology. The ADD method plays a crucial role in ensuring that all pertinent aspects are thoroughly considered, thereby facilitating the creation of a robust and fitting design. [2]

One of its most pivotal characteristics lies in its provision of meticulous, step-by-step guidance, outlining the tasks to be executed within each design iteration. This methodological approach is instrumental in guiding the design process toward comprehensive and well-informed outcomes and specifically concentrates on quality attributes achieving them by selecting various types of structures represented through different views. [2]

Additionally, the method's focus on varied structures and viewpoints enriches the design process, allowing for a comprehensive exploration of architectural possibilities to achieve optimal outcomes. [2]

The following image [2] displays the steps and artifacts provided by the ADD method.



Design purpose
Primary functional requirements
Quality attribute scenarios
Constraints
Architectural Concerns
Step 1: Review Inputs
Step 2: Establish iteration goal by selecting drivers
Step 3: Choose one or more elements of the system to refine
Step 4: Choose one or more design concepts that satisfy the selected drivers
Step 5: Instantiate architectural elements, allocate responsibilities and define interfaces
Step 6: Sketch views and record design decisions
Step 7: Perform analysis of current design and review iteration goal and achievement of design purpose
(Refined) Software Architecture Design
From previous round of iterations or from existing system (brownfield development)
Iterate if necessary
Legend:
Driver
Architecture design
Process Step
Precedence
Artifact flow

Figure 13- Steps and artifcats (ADD)

For practical reasons, steps 4, 5, and 6 of the ADD methodology were streamlined into a single consolidated step.

Step 6 of the ADD method serves as the phase where the informal documentation or "sketches" generated during conference discussions are transformed and recorded as formal documentation. [2]

Due to practical constraints, the presentation will focus solely on the finalized formal documentation. This comprehensive documentation will feature refined diagrams for each design concept, clearly illustrating architectural element allocation, responsibilities, and interfaces.

STEP 1: REVIEW INPUTS

Category	Details
Design purpose	This is a brownfield system in a mature domain. The purpose is to design for the next system version.
Primary functional requirements	All the use cases are defined as primary.

Category	Id	Quality attribute	Scenario	Associated use case	Importance to customer	Difficulty of implementation according to architect
Quality attribute scenarios	QA1	Extensibility	The system shall support switching between data sources (with no data migration) by just updating a configuration.	All	High	High
	QA2	Extensibility	The system shall support switching between different SKU generation mechanisms by just updating a configuration.	UC-1	High	Low
	QA3	Extensibility	The system shall support switching between different review recommendation mechanisms by just updating a configuration.	UC-11	High	Medium

Constraints	CON1: The system must support the following data sources (not simultaneously): PostgreSQL, Mongodb and Neo4J.
Architectural concerns	CRN1: The system shall be programmed using Java and Java-related technologies.
Existing architecture design	Since this is brownfield development, an additional input is the existing architecture design, which was described in the previous section.

Table 1- Review Inputs

ADD Iteration 1

SETP 2: ESTABILISH ITERATION GOAL BY SELECTING DRIVERS

In this iteration, the primary objective remains addressing the QA1 requirement, which mandates the system to enable effortless switching between data sources without requiring data migration. This seamless transition should be achievable through a straightforward configuration update. To fulfill this requirement effectively, the goal is to establish a robust abstraction for the data persistence layer. By doing so, the system can gain inherent extensibility, allowing for future enhancements and adaptability to different data sources.

STEP 3: CHOOSE ONE OR MORE ELEMENTS OF THE SYSTEM TO REFINE

In this iteration, the focus will be on refining the 'Backend' components responsible for data persistence.

STEP 4: CHOOSE ONE OR MORE DESIGN CONCEPTS THAT SATISFY THE SELECTED DRIVERS + INSTANTIATE ARCHITECTURAL ELEMENTS, ALLOCATE RESPONSABILITIES AND DEFINE INTERFACES + SKETCH VIEWS AND RECORD DECISIONS

In this iteration, the formal documentation is represented through the following diagrams. In the logic view - level 2, three new components have been allocated, each responsible for data persistence. These components employ distinct database technologies, and collectively, they provide interfaces for the system's 'Backend'.

On a higher level of granularity, logic view - level 3, there were introduced two different design concepts/approaches (concept A and concept B). While concept A aligns with the goals established for this iteration, a closer examination reveals a deviation from the Dependency Inversion Principle (DIP) of the SOLID principles [3].

Specifically, the provision of three different interfaces for the service layer, where a conditional expression could be used to select the required interface, does not adhere to the DIP. To be more precise, the 'Services' component, instead of relying on an abstraction from the persistence layer, is currently dependent on a concrete implementation. To enhance the design and uphold DIP, it's crucial to reconsider the approach to interface implementation.

A more suitable solution is presented in concept B, where a new component named 'Repositories' is introduced. This 'Repositories' component is designated to handle the Spring Data Repositories abstraction.

The interfaces provided by this 'Repositories' component are defined by the 'Services' components, adhering to the Dependency Inversion Principle (DIP). In this design, the 'Repositories' component acts as an intermediary, decoupling the service layer from the concrete implementations in the persistence layer.

Logical View - level 1

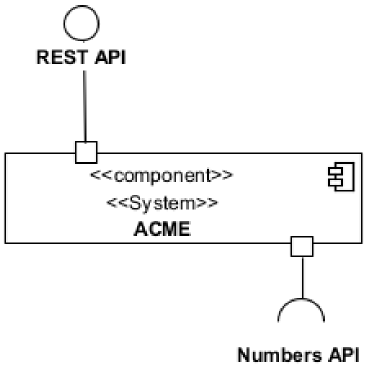


Figure 14- Logical View - level 1

Logical View - level 2

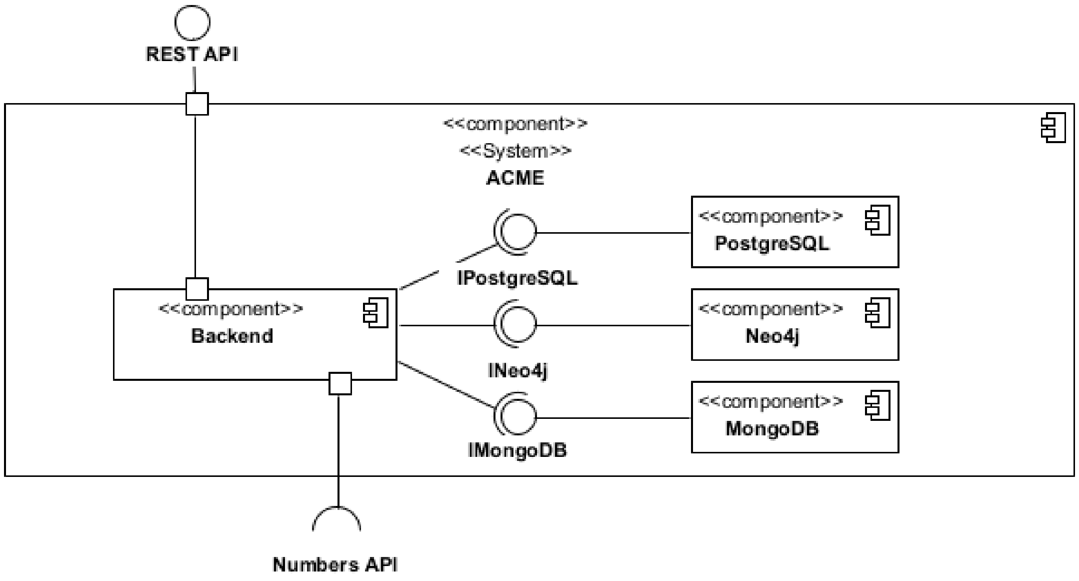


Figure 15- Logical View - level 2

Logical View - level 3 (concept A)

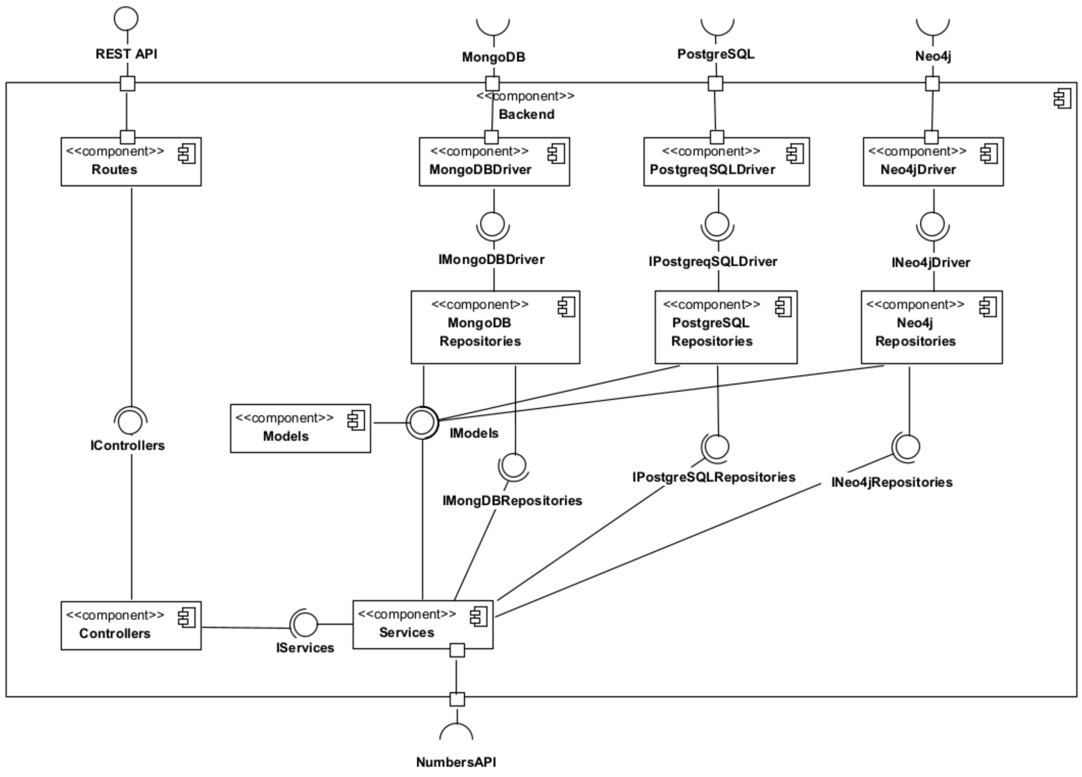


Figure 16- Logical View - level 3 (concept A)

Logical View - level 3 (concept B)

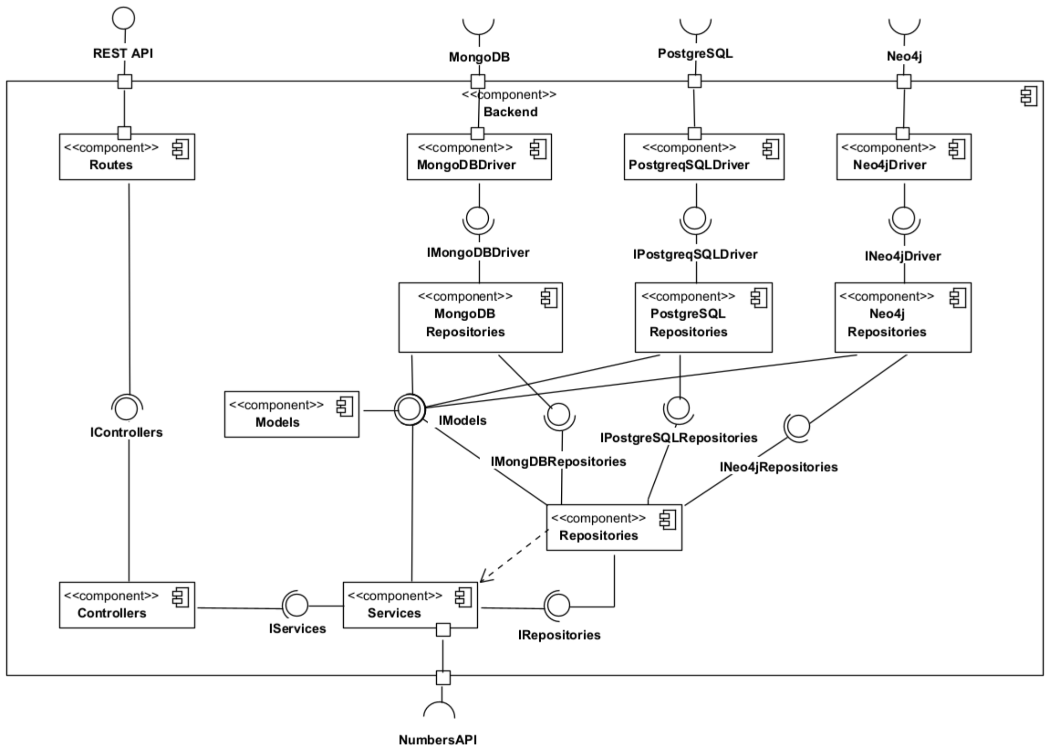


Figure 17- Logical View - level 3 (concept B)

Implementation View

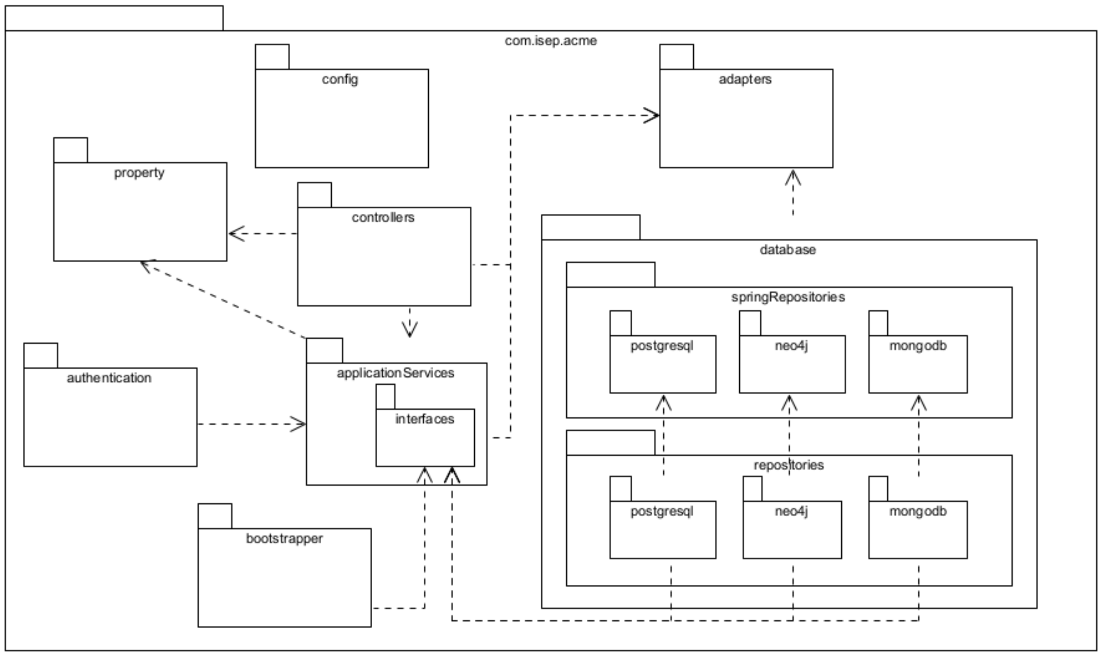


Figure 18- Implementation View

STEP 5: PERFORM ANALYSIS OF CURRENT DESIGN AND REVIEW ITERATION GOAL AND ACHIEVEMENT OF DESIGN PURPOSE

The following Kanban table summarizes the design progress and the decisions made during the iteration.

Not Addressed	Partially Addressed	Completely Addressed	Design Decisions Made During the Iteration
		QA1	Introduction of an abstraction layer that effectively decouples the service layer from the persistence layer.
		CON1	The system supports PostgreSQL, Mongodb and Neo4J.
		CRN1	The technologies that have been selected are Java related.

Table 2- design progress and the decisions made during the iteration

At the conclusion of this iteration, we have successfully achieved our goal; however, upon closer analysis, there are notable concerns regarding our current design. Specifically, certain components are not adhering to the Single Responsibility Principle (SRP). For instance, the 'Models' component is tasked with both persistence and domain logic, violating the SRP. Similarly, the 'Services' component is overly entangled with HTTP requests, expanding its responsibilities beyond what is ideal.

To enhance our design, it is imperative to address these issues. One potential solution is to refactor the 'Models' component, separating the responsibilities of persistence and domain logic into distinct modules or components. By doing so, each component can focus on a singular purpose, aligning with the SRP and ensuring a more maintainable and comprehensible codebase.

Additionally, for the 'Services' component, decoupling it from direct dependencies on HTTP requests is essential. Consider employing abstraction layers or interfaces to isolate the HTTP-related functionalities. This separation of concerns will not only enhance the modularity of our system but also pave the way for easier future modifications and expansions.

In conclusion, two new architectural concerns emerge:

CRN2: Enhance the design by segregating the duties of data persistence and domain logic into separate modules or components.

CRN3: Detach modules from direct reliance on specific HTTP implementations, promoting a more abstract and versatile architectural approach.

All this new concerns will be considered as drivers for the next iteration.

ADD iteration 2

SETP 2: ESTABILISH ITERATION GOAL BY SELECTING DRIVERS

The objective of this iteration is to enhance the existing design to accommodate the drivers that were overlooked in the previous iteration, as well as any new ones that emerge.

STEP 3: CHOOSE ONE OR MORE ELEMENTS OF THE SYSTEM TO REFINE

On this iteration the element to refine is the whole system.

STEP 4: CHOOSE ONE OR MORE DESIGN CONCEPTS THAT SATISFY THE SELECTED DRIVERS + INSTANTIATE ARCHITECTURAL ELEMENTS, ALLOCATE RESPONSABILITIES AND DEFINE INTERFACES + SKETCH VIEWS AND RECORD DECISIONS

During this design stage we will focus on externalizing infrastructural details, data storage, and user interface and pushing them to the edges of our application as opposed to building the application around those pieces. To achieve this objective, our focus will be on delving into the fundamental principles of both the Onion and Clean Architecture paradigms [4], [5]. Both approaches are rooted in the SOLID principles, emphasizing the importance of software design that is extensible, maintainable, and adaptable. Our aim is to identify the architectural model that aligns most seamlessly with our specific project requirements.

Design options

- Adopt Onion Architecture

The Onion Architecture focuses on controlling coupling in software development by organizing the codebase into distinct layers. This approach helps prevent dependencies from flowing inwards, ensuring that inner layers remain decoupled from outer layers. The central tenet of this architecture is to adhere to the Dependency Inversion Principle (DIP), which states that high-level modules should not depend on low-level modules, but both should depend on abstractions. [4]

The innermost layer typically represents the Domain, encapsulating the core business logic and entities. Surrounding the Domain layer are application services, responsible for coordinating actions within the application and acting as an interface between the domain and the external layers. [4]

On the outer side, there is the infrastructure layer, which handles external concerns such as databases, external services, and user interfaces. Crucially, the architecture emphasizes the externalization of the database layer, ensuring that the application remains independent of specific database technologies, which aligns with the outcome of the last iteration. [4]

- Adopt Clean Architecture

The Clean Architecture shares similarities with the Onion Architecture concerning dependency flow and domain centrality, both emphasizing the importance of decoupling and adhering to the 'SOLID' principles. [5] However, there are notable distinctions between the two.

In the Onion Architecture, application services are responsible for coordinating the flow of data to and from entities (domain objects). In contrast, the Clean Architecture achieves this through splitting application logic by use cases.

Reason for discarding: The rationale behind this decision was that splitting the middle layer into use cases was perceived as overengineering due to the relatively low complexity of the project.

This choice was made to strike a balance between architectural best practices and the specific needs and complexity of the given project.

- Adopt Repository pattern

In response to a concern identified during the last iteration (CRN2), which highlighted the need to decouple modules/components from direct HTTP implementations [6], the decision to adopt the repository pattern was made promoting an abstraction layer between the application and external data sources.

- Adopt Domain Driven Design

The decision to adopt Domain-Driven Design (DDD) was driven by its ability to enhance adherence to the Single Responsibility Principle (SRP) and improve the overall maintainability of the codebase.

DDD not only streamlines the organization of domain logic but also simplifies the process of discovering, reusing, and maintaining code components. [7]

In this approach, the core domain logic is encapsulated within aggregate roots. These aggregate roots act as cohesive entities, ensuring that related functionalities and data are encapsulated together.

Coupled with the repository pattern implemented in the persistence layer, which is responsible for storing and retrieving these aggregates, DDD provides a structured and efficient way to manage domain models. [7]

In the updated diagram, we present the new logical view of our system. Notably, all changes were concentrated at level 3, so there was no need to represent levels 1 and 2 again. The red lines represented on the diagram delineate the different layers of our solution.

Logical View - level 3

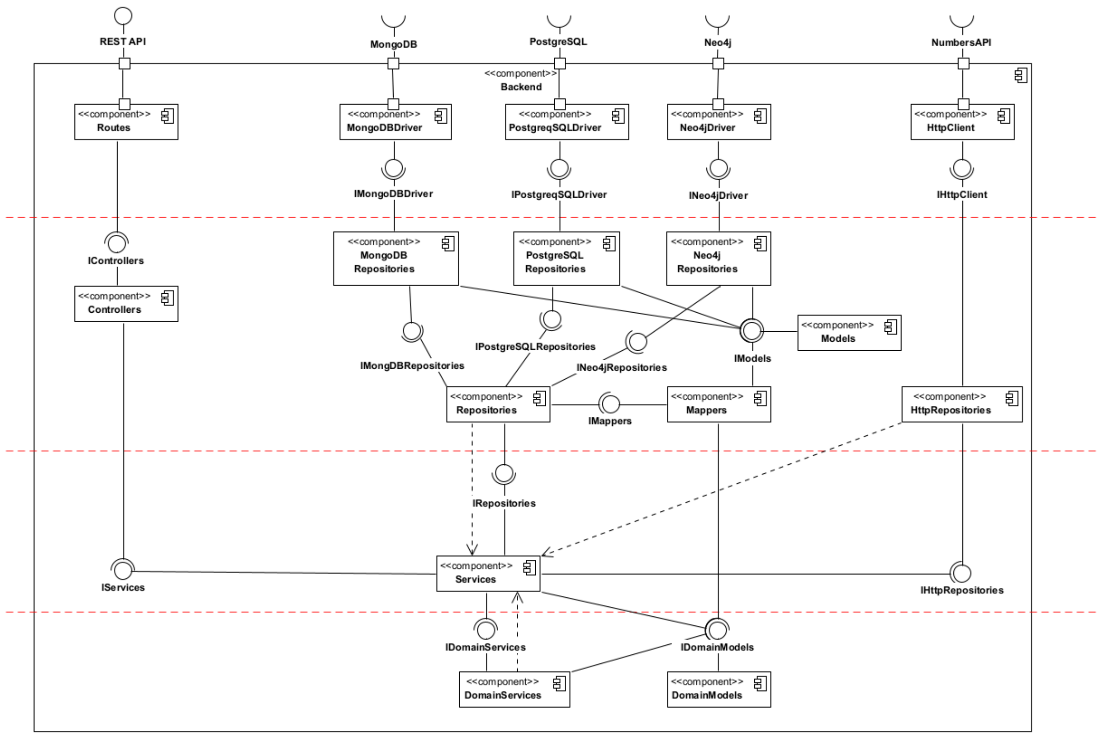


Figure 19- Logical view- level 3

The following class diagram provides a detailed overview of the 'DomainModels' component, showcasing the implementation of Domain-Driven Design (DDD) in our domain models. Within this diagram, three distinct contexts are delineated by the pink circles. Each context houses an aggregate root, representing a core entity encapsulating related functionalities and data. These aggregates are named 'Product,' 'Review,' and 'User'.

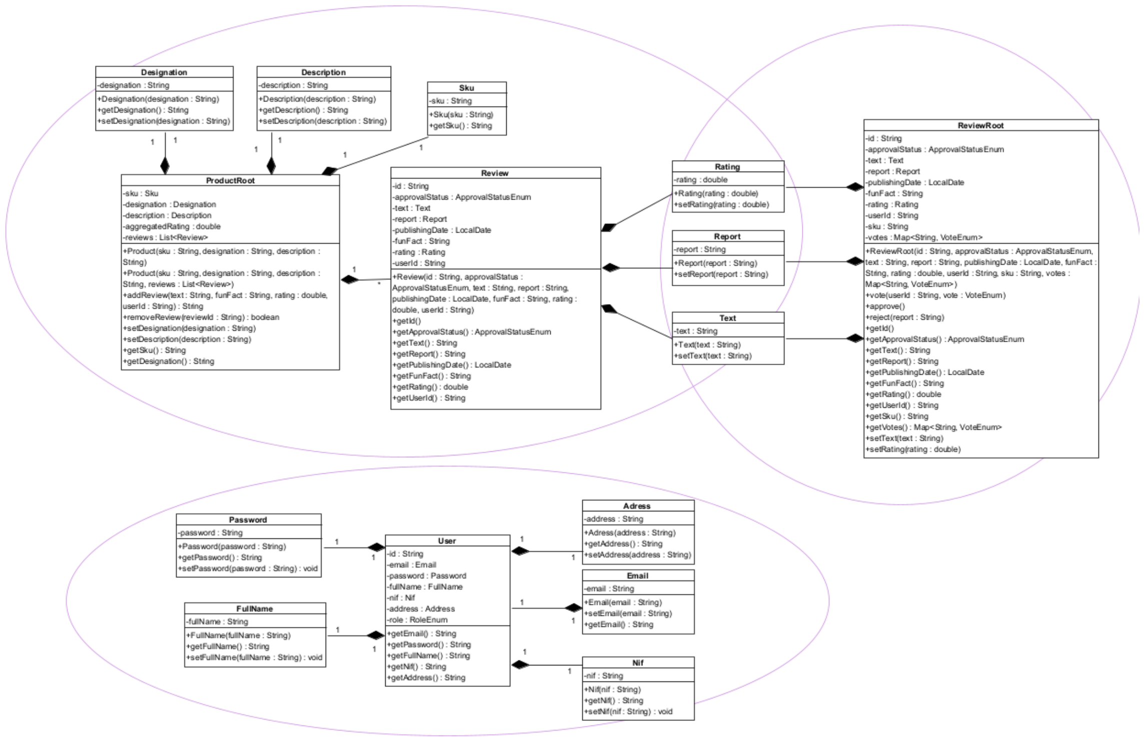


Figure 20- Domain Models Component- contexts represented

In this diagram, we delve into a specific component: the 'DomainServices.' This class diagram provides a detailed perspective on this essential element of our system. Notably, it exemplifies our adherence to the Liskov Substitution Principle (LSP) from the SOLID principles [3]. Through the whole application, we ensured that every method was appropriately implemented, eliminating any possibility of a method being left unsubscribed.

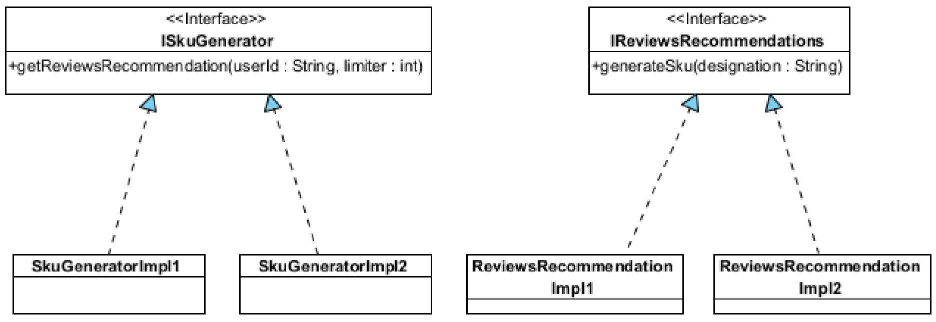

<<Interface>>
ISkuGenerator
+getReviewsRecommendation(userId : String, limiter : int)
<<Interface>>
IReviewsRecommendations
+generateSku(designation : String)
SkuGeneratorImpl1
SkuGeneratorImpl2
ReviewsRecommendation Impl1
ReviewsRecommendation Impl2

Figure 21- DomainServices Component

Implementation View - level 1

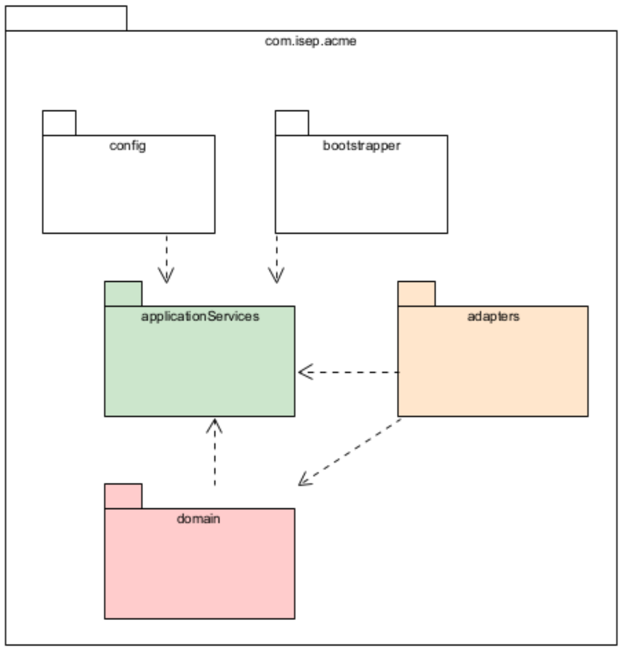


Figure 22- Implementation View - level 1

Implementation View - level 2

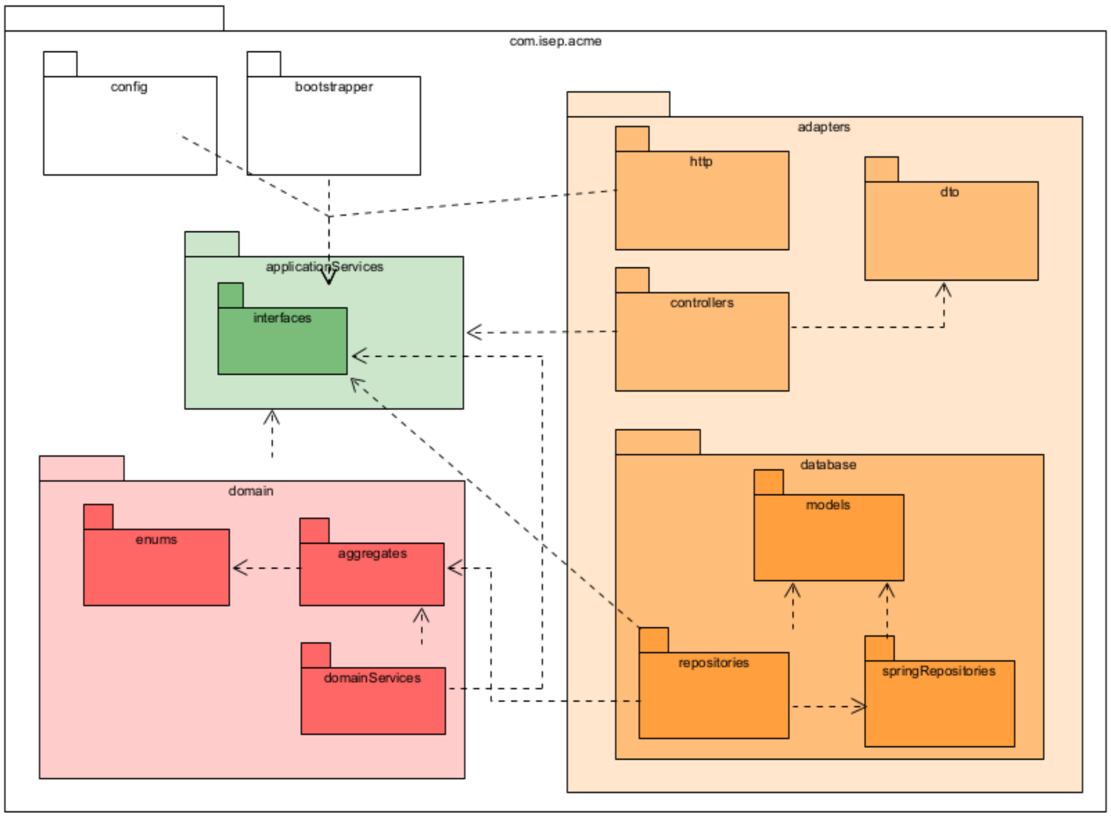


Figure 23- Implementation View - level 2

Implementation View - level 3

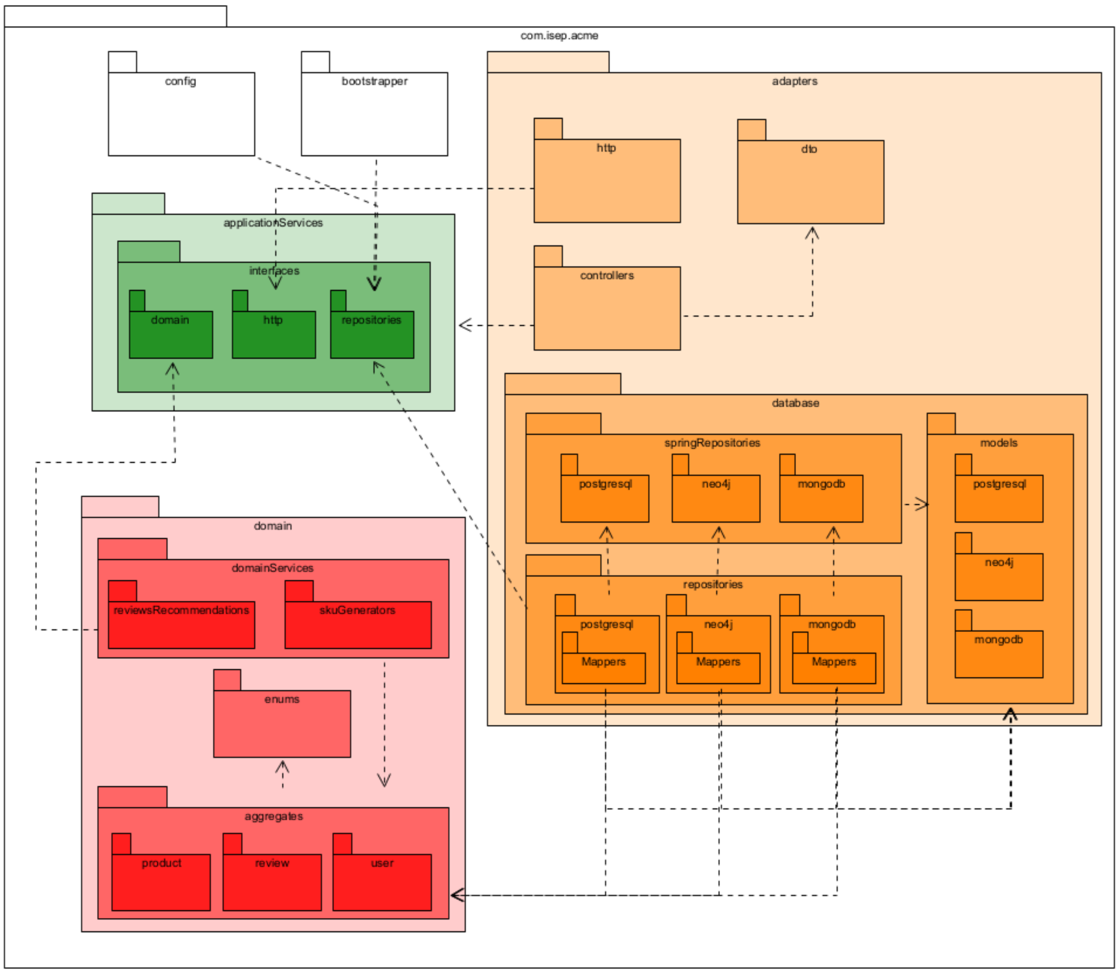


Figure 24- Implementation View - level 3

Mapping from Logical view to Implementation view

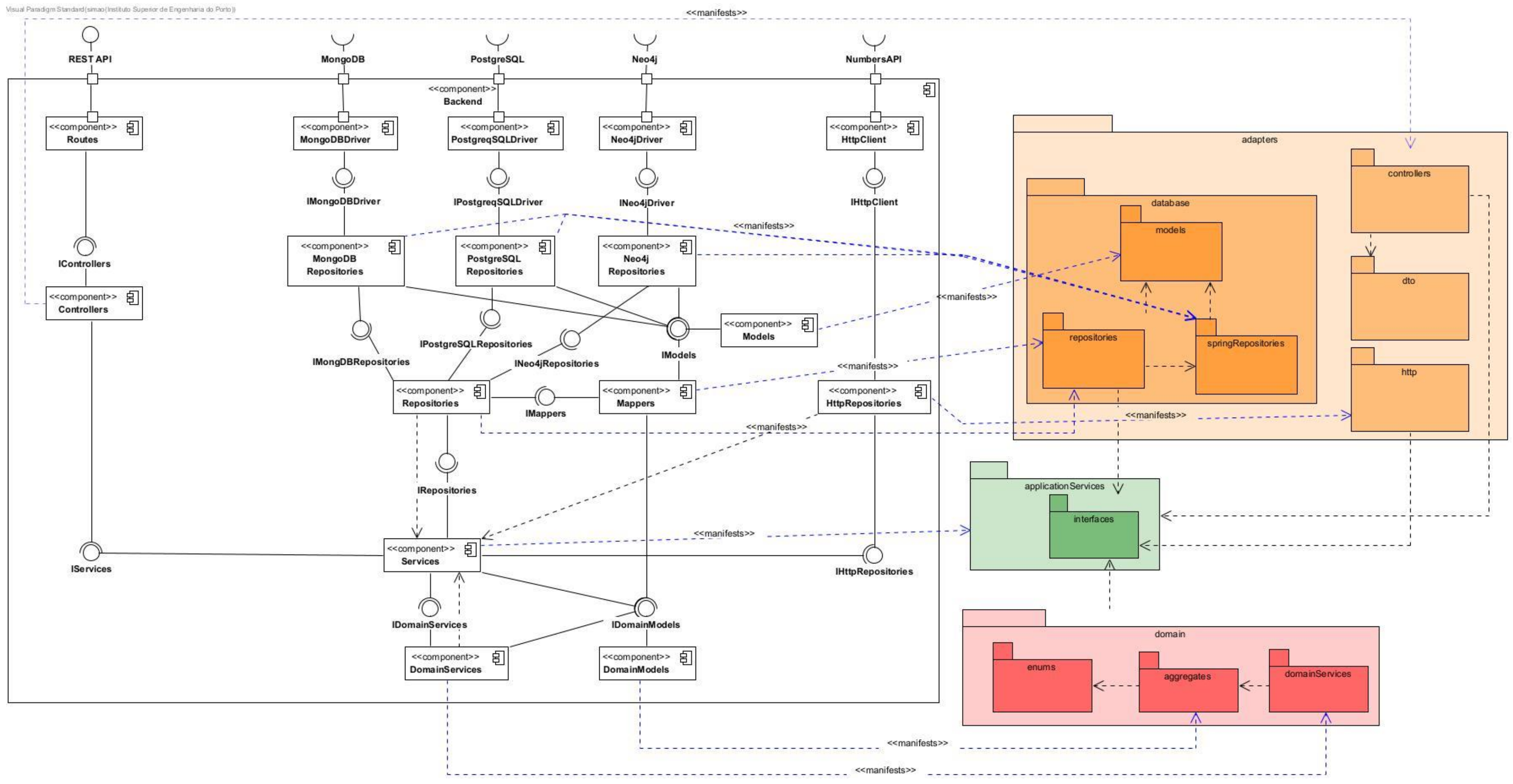


Figure 25- Mapping from Logical view to Implementation view

Process View - Create product

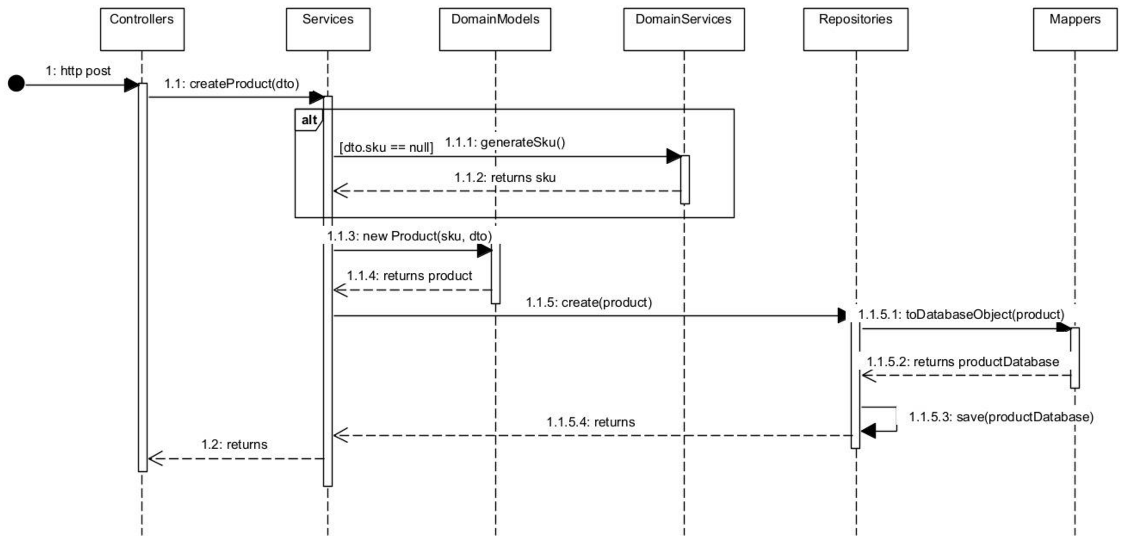
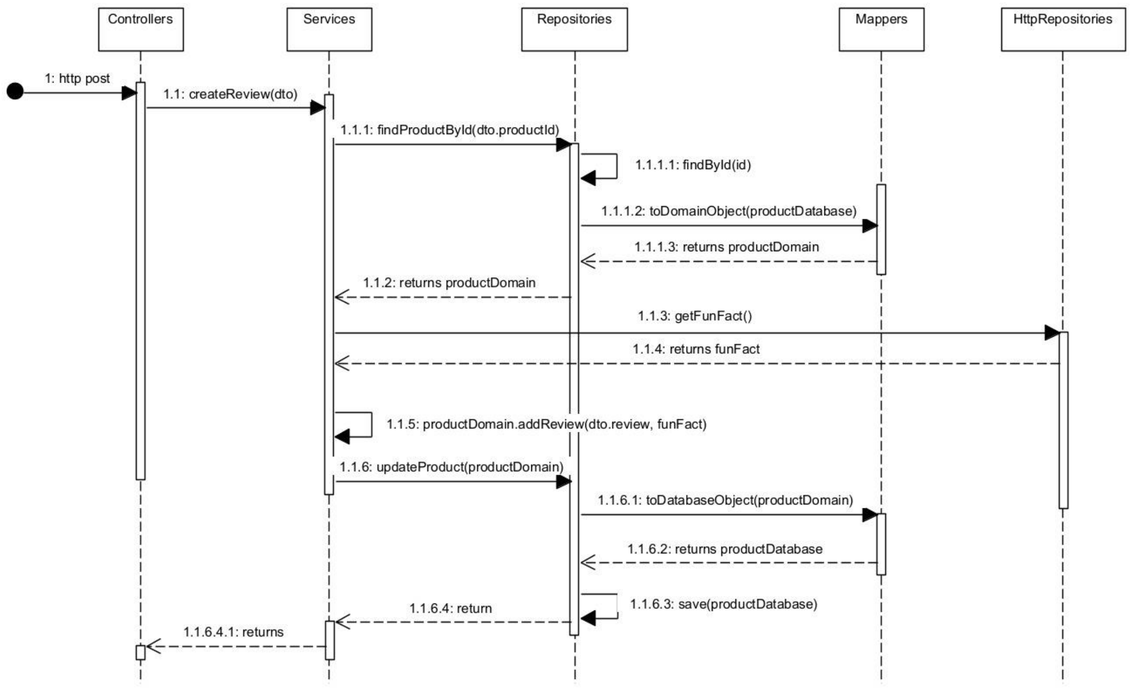


Figure 26- Process View - Create product

Process View - Create review



Controllers
Services
Repositories
Mappers
HttpRepositories
1: http post
1.1: createReview(dto)
1.1.1: findProductById(dto.productId)
1.1.1.1: findById(id)
1.1.1.2: toDomainObject(productDatabase)
1.1.1.3: returns productDomain
1.1.2: returns productDomain
1.1.3: getFunFact()
1.1.4: returns funFact
1.1.5: productDomain.addReview(dto.review, funFact)
1.1.6: updateProduct(productDomain)
1.1.6.1: toDatabaseObject(productDomain)
1.1.6.2: returns productDatabase
1.1.6.3: save(productDatabase)
1.1.6.4: return
1.1.6.4.1: returns

Figure 27- Process View - Create review

Process View - Create vote

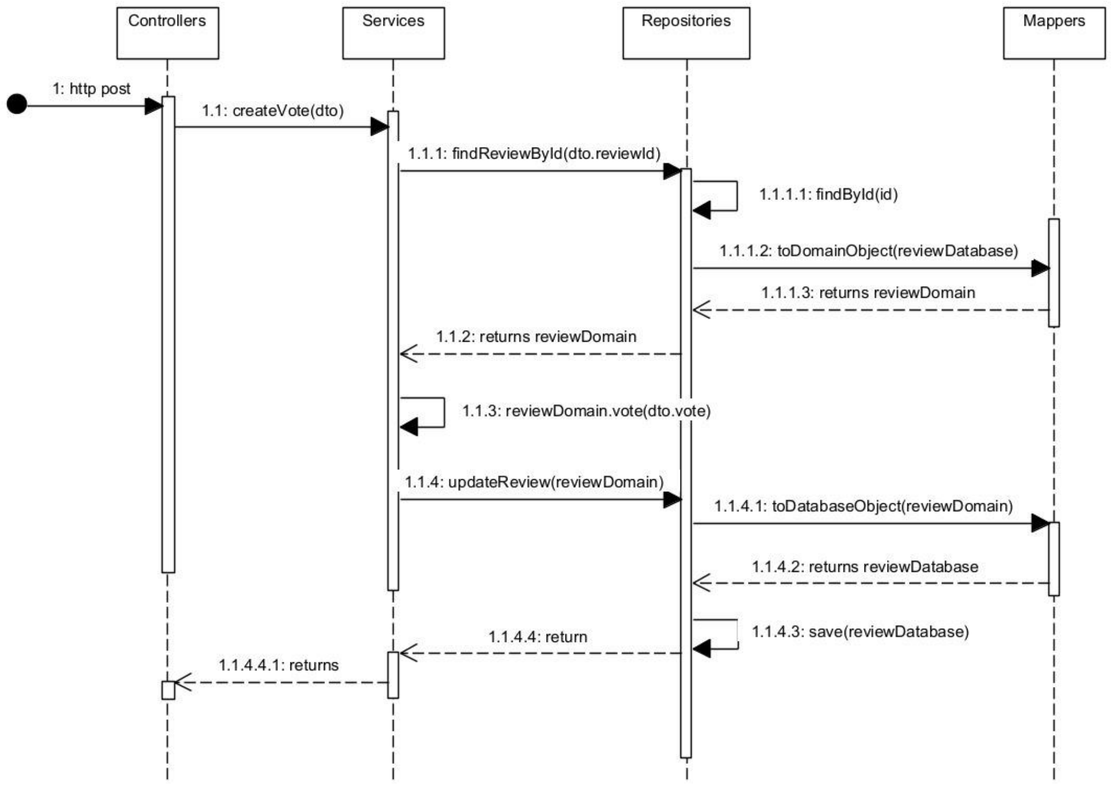

Controllers
Services
Repositories
Mappers
1: http post
1.1: createVote(dto)
1.1.1: findReviewById(dto.reviewId)
1.1.1.1: findById(id)
1.1.1.2: toDomainObject(reviewDatabase)
1.1.1.3: returns reviewDomain
1.1.2: returns reviewDomain
1.1.3: reviewDomain.vote(dto.vote)
1.1.4: updateReview(reviewDomain)
1.1.4.1: toDatabaseObject(reviewDomain)
1.1.4.2: returns reviewDatabase
1.1.4.3: save(reviewDatabase)
1.1.4.4: return
1.1.4.4.1: returns

Figure 28- Process View - Create vote

Physical View

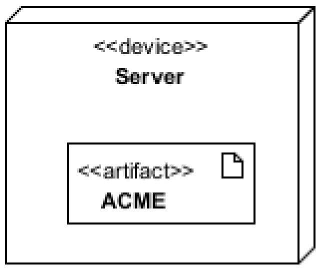


Figure 29- Physical View

STEP 5: PERFORM ANALYSIS OF CURRENT DESIGN AND REVIEW ITERATION GOAL AND ACHIEVEMENT OF DESIGN PURPOSE

The following Kanban table summarizes the design progress and the decisions made during the iteration.

Not Addressed	Partially Addressed	Completely Addressed	Design Decisions Made During the Iteration
		QA2	Introduction of a new component specialized in managing domain services logic in adherence to the Dependency Inversion Principle (DIP).
		QA3	Introduction of a new component specialized in managing domain services logic in adherence to the Dependency Inversion Principle (DIP).
		CRN2	Introduction of a new component dedicated to handling domain logic in alignment with Domain-Driven Design (DDD) principles.
		CRN3	Adoption of the repository pattern for external API requests.

Table 3- Design progress and the decisions made during the iteration

Tests

Integration Testing

During implementation, we used a practical testing approach. We utilized the 'testcontainers' framework, which allowed us to quickly test our application logic with different databases like MongoDB, PostgreSQL, and Neo4j. The beauty of this method was its simplicity: we could switch between databases effortlessly, making our testing process much faster.

One key benefit was that our tests were designed on the service layer, which stayed separate from the complex database details. This meant that once we wrote the tests, we didn't have to tweak them for each database. It saved us time and ensured our tests worked reliably across all databases.

Unit Testing

Unit testing is a crucial practice in software development that ensures the reliability and correctness of individual components or functions. In this case, we tested the functionalities responsible for generating SKUs (Stock Keeping Units) for products.

By testing the SKU generators individually, we could confirm that it generates correct SKUs according to our business logic.

Conclusion

In summary, the iterative architectural enhancements implemented have not only effectively addressed existing issues within the ACME application but have also future-proofed its foundation. This refined architectural solution now aligns seamlessly with essential software design principles, including the Open/Closed principle (enabling extensibility and configurability) and other vital SOLID principles such as Single Responsibility, Dependency Inversion, and Interface Segregation.

The resulting system showcases a meticulously structured, modular, and decoupled architecture, ensuring its adaptability to future modifications and expansions. By adhering to industry best practices and fundamental principles, the ACME application stands ready for continuous growth and evolution within the dynamic landscape of software development.

In conclusion, all assigned architectural challenges were successfully addressed.

Bibliography

[1] P. D. Jayawardene, “4+1 Architectural view model in Software - Javarevisited - Medium,” https://medium.com/javarevisited/4-1-architectural-view-model-in-software-ec407bf27258.

[2] H. Cervantes and R. Kazman, *Designing Software Architectures: A Practical Approach*. 2016.

[3] J. R. Da Paixão, “O que é SOLID: O guia completo para você entender os 5 princípios da POO,” https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530.

[4] J. Palermo, “The Onion Architecture : part 1,” https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/.

[5] R. C. Martin, *Clean architecture*. Pearson Professional, 2018.

[6] P.-E. Bergman, “Repository Design Pattern - Per-Erik Bergman - Medium,” https://medium.com/@pererikbergman/repository-design-pattern-e28c0f3e4a30.

[7] Kexugit, “Best Practice - An Introduction To Domain-Driven Design,” https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design.