ARQSOFT

ISEP – Master's in Software Engineering

1.Introduction

The goal of this project is to improve library management systems by solving important reliability, scalability and configurability problems. Modules for manage is books, genres, authors, readers and book lending procedures are part of a library management system that was created as a REST-orientated backend service. However, the current systems are not usable for accommodating changing functional requirements, such as the ability to connect to various identity providers and database management systems (DBMS) or the ability to dynamically generate and recommend loan data based on certain configurations.

2. Add

The Attribute Driven Design (ADD) methodology is a strategy for creating software architectures in which the design process is guided by the requirements for software quality characteristics. In this project, we use the Add methodology to explain the software architecture used. ADD uses a recursive design approach and systematically decomposes a system or its components by using architectural strategies and patterns that meet its primary requirements.

Figure 1 : Add Steps

Quality Attibrutes

ID	Quality Attribute	Description
QA-1	**Maintainability**	The facility with which the system can be modified, updated or improved. A system with a high level of maintainability allows for the rapid correction of errors, adjustments and improvements with minimal risk of introducing new problems.
QA-2	**Performance**	The ability of the system to respond to tasks within an acceptable time frame under expected workload. High performance ensures the system meets user expectations for speed and efficiency.

QA-3	**Availability**	The degree to which the system is operational and accessible when needed. A highly available system minimizes downtime and ensures continuous access for users.
QA-4	**Scalability**	The system's ability to handle increased workload or user demand by adding resources. Scalability allows the system to grow in capacity as demand grows.
QA-5	**Elasticity**	The system's ability to dynamically adjust resources to match the workload demand. Elastic systems can scale up or down in real-time, optimizing resource usage.
QA-6	**Compatibility**	The ability of the system to operate within a variety of hardware, software, and network environments. Compatibility ensures the system can work with different configurations and technologies.
QA-7	**Interoperability**	The capability of the system to exchange information and work with other systems seamlessly. Interoperability facilitates communication and data sharing across different systems.
QA-8	**Releasability**	The ease and speed with which new versions of the system can be released to production. High releasability allows for frequent, reliable, and low-risk deployments.

Table 1 : Quality Attributes

Fuctional Requirements

Figure 2 : Functional Requirements

Quality Attribute Scenarios

ID	Quality Attributre	Scenario	Associated use Case	Priority(Importance,Difficu
QA-1	Usability,Interoperability	The system must provide links in the resource represation	All	(H,M)
QA-2	Security,Authentication	All authenticated request must use JWT	All	(H,H)
QA-3	Documentation,Usability	The system must provide an OpenAPI specification	All	(H,M)

QA-4	Usability,Documentation	The system must include sample requests and responses(e.g.,Postman collection)	All	(M,M)
QA-5	Automation,testing	The system must provide automated tests (e.g.,Postaman collection)	All	(M,M)
QA-6	Perfomance,Scability	Long result lists must support pagination	All	(H,M)

Table 2 : Quality Attribute Scenarios

Iteration 1

Goal : Create an overall system structure

Monolithic

This project uses a monolithic architecture, a software development paradigm in which all the components of an application are combined into a single, indivisible block. Each module, including data access, business logic and the user interface, is used to compile and function as a single unit.

Aspects Pros/Cons for this Architecture:

Aspect	Advantages	Disadvantages
1. Simplicity	- Easy to develop and implement, making it a common choice for smaller projects.	- Maintaining this simplicity becomes complicated as the system grows and new functionalities are added to the system.
2. Performance	- Direct communication between modules, without the overhead of network calls.	- Limited scalability, making it difficult to scale only parts of the application.
3. Testing and Debugging	- Testing and debug the system is easy, because everything is in the same environment.	- Requires recompilation and redistribution for the entire application after each change, reducing development agility.

Table 3 : Aspects Pros/Cons of monolithic:

Figure 3:Physical View Level 1 System-as-is

Figure 4:Logic View Level 1 System-as-is

Figure 5:Logic View Level 2 System-as-is

Figure 6:Logic View Level 3 System-as-is

Figure 7:Logic View Level 4 System-as-is

Figure 8:Implementation View Level 1 System-as-is

Figure 9 : Implementation View Level 2 System-as-is

Figure 10:Implementation View Level 3 System-as-is

Figure 11:Implementation View Level 4 System-as-is

Design Alternatives

Microservices Architecture Example

Figure 12: Microservices Architecture

Common Patterns and Use Cases

Event-Driven Microservices:

- Asynchronous communication using a message broker (e.g., Kafka, RabbitMQ, or AWS SQS) to decouple services.
- **When to Use** : Ideal for applications needing real-time, flexible workflows, such as e-commerce, IoT, or social media.

CQRS:

- Separates the data modification (commands) from data retrieval (queries), allowing each to be optimized independently.
- **When to Use**: Best suited for systems with complex business logic and requirements for scalability, where read and write operations have different performance characteristics.

Key Components and Design Strategies

- Scalability: Load balancers optimize resource usage by distributing traffic across instances, while microservices architecture allows for independent scaling based on demand and distributed databases support increased data volume through horizontal scaling.
- Fault Tolerance: Secondary databases ensure data availability during primary failures, load balancing reroutes traffic to healthy instances, and service replication maintains redundancy through multiple service instanceses support increased data volume through horizontal scaling.
- Resource Pooling: Shared database and cache resources improve efficiency by allowing multiple services to access common data stores.
- Lazy Loading: Caching implementation defers data retrieval until needed, providing faster access to frequently used data.
- Broker: Services can publish events or messages to a central system, where other services can subscribe to and process those messages independently. This enables improved handling of spikes in traffic
- Domain-Driven Design (DDD): Services structured around business domains, ensuring each service encapsulates its own domain logic and data, which facilitates scalability and maintainability while promoting clearer communication and collaboration within teams.

Aspects Pros/Cons for this Architecture:

Aspect	Advantages	Disadvantages
1. Increased Modularity	- Independent microservices that can be developed, deployed, and scaled individually.- Easier maintenance and updates of specific services.	- Increased complexity in managing and coordinating multiple services.- More challenging end-to-end testing.
2. Scalability and Flexibility	- Services can scale independently based on demand.- Flexibility to use different tech stacks for each service.	- Complexity in scaling all services while maintaining performance.- Some services might require more advanced scaling strategies.
3. Fault Isolation	- Failure in one service does not necessarily bring down the whole system.- Improved system resilience.	- Requires sophisticated error handling across services to prevent cascading failures.- Network communication issues can still propagate failures across services.
4. Event-Driven Communication	- Loose coupling through message brokers.- Asynchronous	- Managing message brokers adds operational complexity.- Eventual consistency can lead to

	communication for more responsive systems.	delayed data synchronization between services.
5. Independent Data Storage	- Each service can have its own database, optimized for its specific needs.- Decentralized data management allows easier scaling and resilience.	- Cross-service data queries become harder.- Data duplication and inconsistencies might occur.- Managing distributed transactions or data consistency across services is complex.
6. Continuous Integration / CI/CD	- Faster and more frequent deployments with minimal disruption to other services.- Easier to implement modern DevOps practices such as blue-green deployments.	- Requires sophisticated automation and orchestration tools for managing pipelines for all services.- Infrastructure costs rise as services grow.
7. Service Decoupling	- Services communicate via well-defined interfaces and protocols, making them independent from each other.	- Network overhead and potential latency due to services interacting over the network.- Service discovery and inter-service communication require careful handling.
8. Fault Isolation and Resilience	- System resilience improves with fault isolation (e.g., individual service failure won't bring the whole system down).	- Requires careful design to ensure cascading failures don’t occur (e.g., circuit breakers, retries).- Debugging failures in a distributed system is harder, requiring advanced monitoring and logging tools.

Table 4 : Aspects Pros/Cons of Microservices:

CI/CD

Figure 13 : Physical View Level 1 system-to-be

External Components and Tools Used

- **Spring Boot** for backend development, facilitating the creation of RESTful APIs.
- **JPA** to facilitate data persistence, working as an intermediary between the system and the database.
- **Relational Database** as H2 to store information about books, readers, loans, etc.
- **Spring Security** for authentication and authorisation, supporting the AUTH API.

Board

Not Addressed	Partially Addressed	Addressed
UC-1	QA-3	Structure of the System
UC-2		QA-1
UC-3		QA-8
UC-4		
QA-2		
QA-4		
QA-5		
QA-6		
QA-7		

Table 5 : Board of Iteration 1

Iteration 2

Goal : Identify structures to support primary funcionality

The requirements developed in this iteration were:

- Persisting data in different data models (e.g. relational, document) and SGBD:
  - Relacional data model:SQL Server
  - Document data model (NoSQL): e.g. MongoDB,
- Adopting different IAM (Identity and Access Management) providers:
  - Google
  - Facebook

Requirement 1

1. PersisFng data in different data models (e.g. relational, document) and SGBD (e.g. MySQL, SQL Server, MongoDB, Redis).

Element	Statement
Stimulus	Impossibility to use other types of DBMS when needed by product owner
Stimulus source	Product owner needs to use different DBMS
Environment	Product owner has the need to use different DBMSs, which currently forces the use of different branches and increases the possibility of repeating code

Artifact	The software in particular persistence
Response	The solution should be based on configuring in such a way that only 1 branch is needed.
Response Measure	Must be realisable in 5 minutes

Table 6 : Requirement 1 PersisFng data in different data models

Requirement 2

2. Adopting different IAM (IdenFty and Access Management) providers (e.g. Google,Facebook, Azure).

Element	Statement
Stimulus	Unable to authenticate with facebook or google account to the software.
Stimulus source	Product owner wishes to facilitate the authentication process by allowing the user to authenticate through multiple providers.
Environment	Product owner wants to add support for multiple identity providers, currently it is necessary to manually implement each integration, which causes maintenance difficulties and an inconsistent experience for users.
Artifact	The authentication and authorisation module
Response	Integrate Spring Security with OAuth 2.0 and OpenID Connect to support different IAM providers, such as Google, Facebook , in a centralised and flexible way
Response Measure	Must be realisable in 5 minutes.

Table 7 : Requirement 2 AdopFng different IAM providers

Tecnical Memo

Issue	System can't support different database/IAM provider types
Issue	The current library management system does not support extensibility.This indicates that adding new features,such as integration with different types of databases/ IAM providers,requires complex changes to the base code,make the systemn difficult to evolve.
Summary of Solution	Adopt the following tacting coupling ,namely use internediary,abstract common service,and defer binding

Factors	The system needs to support the addition of new functionalities without significant structural changes.Lack of extensibility has a direct impact on the system's scalability and adaptability to future requirements.
Solution	Our solution approach uses delayed binding, common abstract services, intermediaries, and tactical coupling to ensure a modular and flexible architecture. The Dependency Inversion Principle (DIP), which states that high-level modules should rely on abstractions rather than specific implementations to allow for greater disaggregation across components, is in line with SOLID principles. Because new implementations of the system can be added without requiring changes to the old code, it also complies with the Open/Closed Principle (OCP).
Motivation	Without changing the current code, the solution includes the need to support the future inclusion of additional services and functionalities. In addition, it improves the application's flexibility.The use of various technologies, such as databases with similar interfaces, guarantees that integration is simple.
Alternatives	This solution uses a manual factory that dynamically selects the right dependencies based on the environment variables, without involving static configurations or Spring profiles. The factory reads the environment variables directly at initialisation time, determining which implementation of each service should be injected into the main service. This allows the system to be dynamically adapted to different configurations and requirements, keeping the code modular and eliminating the need to modify the core code or restart the application for each new configuration.
Pending Issues	The implementation of an effective abstraction layer to accommodate various database types and their corresponding persistence strategies is one of the Pending Issues.

Table 8 : Tecnical Memo system can't support different database/IAM provider types

Aspects Pros/Cons for our Solution:

Aspect	Advantages
High Flexibility and Modularity	Dependencies can be resolved dynamically and replaced as necessary, without changing the main code.
DIP and OCP compliance	The architecture allows the core code to depend on abstractions and remain closed to modifications, but open to extensions.

Scalability for Multiple Contexts	It's easy to add new DBMSs or IAM providers without having to touch the base code, given the use of abstract and intermediate services.

Table 9 : Advantagens for our solution

Aspect	Disadvantages
Increased Complexity	The use of intermediaries and deferred binding makes the architecture more complex and can make maintenance more difficult.
Performance Overload	Delayed binding and dependency on intermediaries can introduce slight performance overheads, especially in large-scale systems.
Increased Configuration Effort	Configuration and management of dynamic dependencies can require extensive and complex configurations, increasing development time.

Table 10 : Disadvantages for our solution

Aspects Pros/Cons for alternative Solution:

Aspect	Advantages
Reduced Complexity	The manual factory simplifies the dependency selection logic, making the code more straightforward and easier to understand.
Runtime Flexibility	Allows the system to be adapted to different implementations using environment variables, without the need to restart or change the core code.
Lower Performance Overload	By avoiding deferred binding and intermediaries, the solution tends to be more efficient in terms of processing.

Table 11 : Advantagens for alternative solution

Aspect	Disadvantages
Less Extensibility and Modularity	Each new implementation requires changes to the factory's logic, making it less open to extensions compared to your current solution.
Dependence on Environment Variables	The need to define environment variables can be a limitation in some environments or complicate configuration in specific contexts.

Less DIP and OCP compliance	The solution depends on specific instances in the factory, which creates a greater coupling with the implementations and reduces abstraction in the logic for choosing dependencies.

Table 12 : Advantagens for alternative solution

Explanation for our choice

The current solution is the ideal choice for the libraryManagement project, as it fully meets the objectives of extensibility, configurability and reliability. With the use of deferred binding, abstract services and tactical coupling, the system is designed to incorporate new implementations (such as different types of databases or IAM providers) without changing the core code, guaranteeing robust extensibility. Configurability is achieved by allowing different components and dependencies to be defined dynamically at runtime, which makes it easier to adapt the system to new contexts and specific requirements without the need for manual recompilation or reconfiguration. In addition, the modular approach and the use of abstractions increase reliability, as they isolate the impact of changes and reduce the risk of failures when introducing new components, creating an architecture that remains stable and responsive in the long term.

Tactics

- **Delayed Binding**
    - **Delayed binding** allows the decision about which implementation to use to be made only at runtime, rather than being determined at the time of compilation or initial configuration. This is made possible by using profiles or environment variables to dynamically select the appropriate dependency according to the active configurations. This approach increases the flexibility of the system, as it allows implementations to be replaced or updated without the need to modify the core code, just by adjusting the configuration. Deferred binding is particularly useful in systems that need to adapt to different execution contexts or environments, such as different types of database or authentication providers.

- **Tactical Coupling**
    - **Tactical coupling** is a strategy that allows a main class, such as a core service, to rely only on abstractions, such as interfaces, to interact with specific implementations, without knowing their internal details. This tactic maintains a minimum and controlled level of coupling, ensuring that new implementations can be integrated without impacting existing code. Tactical coupling promotes modularity and ease of maintenance, while allowing dependencies to be replaced transparently, without restructuring the core logic. This is essential in systems seeking flexibility and scalability, as it allows components to be exchanged without affecting overall behaviour.

- **Use of Abstract Services**

- **The use of abstract services** is a tactic that defines contracts for specific functionalities, such as data persistence or authentication, that all concrete implementations must follow. This means that any new service that implements the abstract interface can be integrated into the system without modifying the core logic, fulfilling the Open/Closed principle. By separating the definition of the service from its implementation, this approach allows for the replacement and expansion of functionalities, keeping the system modular and prepared for changes. The use of abstract services is fundamental for systems that require interoperability and extensibility, allowing for a flexible structure that is independent of the specific technologies used in each context.

Patterns

- **Dependency Injection**
  - **Dependency Injection** is a pattern that aims to increase the modularity and testability of code by allowing the dependencies of a class to be injected instead of being created or defined directly. This pattern promotes decoupling, as the class depends on interfaces or abstractions rather than concrete implementations, making it easier to replace dependencies with other implementations at runtime or testing. In addition, Dependency Injection helps to keep control of dependencies centralised, allowing the system to automatically choose the correct implementations according to configurations.

- **Strategy Pattern**
  - **The Strategy Pattern** makes it possible to define a family of algorithms or behaviours and make them interchangeable depending on the execution context, without the client knowing which specific strategy is being used. It promotes flexibility by encapsulating different execution methods under a common interface, allowing the system to select the correct implementation at runtime. In this way, the Strategy Pattern simplifies the code by moving the decision logic away from the client, which makes it easier to extend the system to new behaviours without changing the existing code.

How the system looks after implementation

Figure 14 : Logic View Level 2 System-to-be

Figure 15 : Logic View Level 3 System-to-be

Figure 16 : Logic View Level 4 System-to-be

External Components and Tools Used

- **Relational Database** SQL Server

- **Document Database** MongoDB
- **OAuth 2.0 Providers** Google,Facebook

Board

Not Addressed	Partially Addressed	Addressed
UC-3	QA-3	Structure of the System
UC-4		QA-1
QA-2		QA-8
QA-5		QA-4
QA-7		QA-6
		UC-1
		UC-2

Table 13 : Board of Iteration 2

Iteration 3

Goal : Refining previously created structures to fully address the remaining drivers

The requirements developed in this iteration were:

- 3.Generating Lending and Authors ID in different formats according to the following specifications:
    - 24 hexadecimal characters
    - 20 carateres alfanuméricos como hash de id de negócio da entidade de negócio
- 4. Recommending Lendings according to following specifications:
    - X books most lent from the Y most lent genre (student 1)
    - Based on the age of the reader: (student 2)

Requirement 3

3. GeneraFng Lending and Authors ID in different formats according to varying specificaFons

Element	Statement
Stimulus	Inability to generate IDs for business owners and authors in different formats as requested by the product owner
Stimulus source	Product owner needs to generate personalized IDs for Loans and Authors, following specific specifications

Environment	Product owner needs variable loan and author IDs in different formats to help search for these same loans or authors.
Artifact	The ID generation and management module (Loans and Authors).
Response	Implement a service for generating IDs of different formats according to the given specifications. This service must be capable of generating IDs with variable formats, dynamically defined through a configuration file or specific rules.
Response Measure	The new ID format must be configured and used in the system within 5 minutes

Table 14 : Requirement 3 GeneraFng Lending and Authors ID in different formats according to varying specificaFons

Requirement 4

4. Recommending Lendings according to varying specificaFons

Element	Statement
Stimulus	Unable to recommend book lendings based on custom criteria provided by the product owner.
Stimulus source	Product owner wants to improve user experience by recommending books that are relevant, based on different specifications
Environment	Product owner needs to customize book recommendations that vary according to the criteria for choosing books read by users, and currently it is necessary to manually implement these logics, which results in complexity and lack of scalability.
Artifact	Loan recommendation system.
Response	Develop a recommendation system that allows the definition of criteria for recommendations, using an analysis of users' reading history, in addition to rules that can be configured through configuration files
Response Measure	Should be doable in 5 minutes.

Table 15 : Requirement 4 Recommending Lendings according to varying specificaFons

Tecnical memo

Aspect	Details
Issue	System lacks flexibility in ID generation and personalized lending recommendations based on reader demographics.

Detailed Issue	The current library management system faces two limitations: it cannot generate unique IDs for lendings and authors in various formats, and it lacks functionality for tailored lending recommendations based on specific criteria, such as genre popularity and reader age. These constraints reduce the system's adaptability and hinder its ability to meet diverse requirements without core code modifications.
Summary of Solution	To address these issues, we propose implementing an extensible ID generation mechanism and a flexible recommendation service. The ID generation will use a domain-level interface combined with the Factory Pattern, while the recommendation service will leverage the Strategy Pattern for configurable lending suggestions.
Factors	- **ID Generation**: The system must generate unique IDs for lendings and authors in multiple formats (e.g., UUID, sequential) without requiring code changes for each format. The current lack of configurability limits the system's adaptability to evolving requirements. - **Lending Recommendations**: The system should provide tailored recommendations based on reader demographics, such as age and genre popularity, within configurable age ranges. This flexibility is essential for enhancing user experience and engagement.
Solutions	- **ID Generation**: Create a general interface, `AlgorithmId`, within the domain layer to define methods for generating lending and author IDs. Use the Factory Pattern to dynamically load the appropriate ID generator based on external configurations. This approach enables new ID formats to be added as independent implementations, adhering to the Open/Closed Principle (OCP) and Dependency Inversion Principle (DIP). - **Lending Recommendations**: Develop a common interface, `Recomentation`, which outlines core methods for recommending lendings. Implement distinct recommendation strategy classes that adapt based on age-specific genres and popularity metrics. These strategies will be selected dynamically through external configurations, ensuring modularity and ease of adaptation to changing business rules.
Motivation	This combined solution provides a modular approach to both ID generation and lending recommendations, allowing the system to meet changing requirements without code rewrites. It simplifies maintenance, minimizes testing overhead, and facilitates rapid adaptation to future needs.
Alternatives	A static ID generation method in the service layer would require code changes for each new ID format, increasing maintenance complexity. Implementing a static recommendation method would necessitate

	code changes for each new recommendation criterion or adjustment in age groups, reducing system flexibility.
Pending Issues	Validate that external configuration supports all required ID formats and test the Factory Pattern implementation for compatibility with new ID generation types. Assess the efficiency of each recommendation strategy in retrieving and processing necessary data in the repository, ensuring that configuration changes reflect accurately in recommendation results.

Table 16 : Tecnical Memo System lacks flexibility in ID generation and personalized lending recommendations

Tactics

To accomplish this, we adopted the following tactics:

Modular Design

- **Modular Design**: By defining a common interface and using separate strategy classes for different algorithms, the system promotes modularity. This allows for independent development and testing.

Dependency Injection

- **Dependency Injection**: Leveraging ApplicationContext in the factory enables us to inject specific implementations based on configurations. This supports loose coupling and enables unit testing by injecting mock or alternative implementations as needed.

Configuration-Driven

- **Configuration-Driven**: By making algorithms selection configurable, the system allows business requirements to dictate ID formats without requiring additional code. This approach improves adaptability and meets client-specific needs by simply changing configuration files.

Patterns

The following design patterns were used to support this flexible and adaptable ID generation solution:

Factory Pattern

- **Factory Pattern**: By centralizing the instantiation logic, it allows for easy integration of new algorithms without altering existing code. This not only promotes a cleaner architecture but also enhances the system's ability to quickly adapt to changing requirements and business rules.

Strategy Pattern

- **Strategy Pattern**: By defining a common interface and implementing different strategies, the system can cater to diverse user needs without compromising the

integrity of the overall design. This not only enhances the user experience but also future-proofs the system against changing requirements.

Open/Closed Principle (SOLID)

- **Open/Closed Principle (SOLID)**: By defining interfaces and implementing each algorithm as a separate class, we achieve an open-for-extension yet closed-for-modification structure, allowing the system to scale to accommodate new ID formats without altering existing code.

Aspects Pros/Cons for this Solution:

Aspect	Pros	Cons
Flexibility and Adaptability	- Factory Pattern allows easy addition of new ID formats without code changes.- Strategy Pattern enables configurable recommendation criteria (age, genre, etc.).	- Adds complexity due to multiple patterns and configuration-driven selection.
Testing and Extensibility	- Dependency Injection enables mock implementations for testing - Easily scalable with new ID formats or recommendation criteria.	- Compatibility and performance testing across configurations require additional resources.
Configuration-Driven Customization	- Business rules and client needs are adaptable via external configuration without changing code.- Supports flexible, user-specific ID formats and recommendations.	- Requires careful configuration management; misconfigurations can lead to unexpected behaviors.

Table 17 : Aspects Pros/Cons for this Solution

Aspects Pros/Cons for Alternate Solution:

Aspect	Pros	Cons
Simplicity	- Easier to implement initially, as each ID generation or recommendation logic is directly in the service layer.	- Lack of modularity means that new formats or criteria require changes in the core service code, violating the Open/Closed Principle (OCP).
Performance	- Reduces the overhead of using patterns like Factory or Strategy, which may improve performance in small	- Increased complexity in large systems due to an accumulation of logic within the service layer,

	applications with limited requirements.	making the code harder to understand and maintain.
Adaptability and Flexibility	- Suitable for applications with fixed, unchanging requirements.	- Requires code changes for each new ID format or recommendation criterion, leading to higher maintenance and reduced flexibility to adapt to new business needs.

Table 18 : Aspects Pros/Cons for this Alternate Solution

Explanation for our choice

The solution of making the algorithms in the business layer is ideal because it offers a more modular, flexible and SOLID-orientated architecture, whereas the alternative limits adaptability and makes long-term maintenance difficult. By implementing the logic for generating IDs and recommendations in the business layer, we use standards such as Factory and Strategy, allowing new criteria or formats to be added without the need to change the base code. This makes it easier to extend the system and reduces the impact of changes in business requirements, since the business logic is encapsulated and separated from the service layers. In addition, external configuration and dependency injection make it possible to customise the application to specific needs without changing the code, while the alternative solution, by centralising the logic in the service layer, creates a rigid and more error-prone system.

How the system looks after implementation

Figure 19 : Logic View Level 4 System-to-be

Board

Not Addressed	Partially Addressed	Addressed
	QA-3	Structure of the System
	QA-5	QA-1
		QA-8
		QA-4
		QA-6
		UC-1
		UC-2
		UC-3
		UC-4

		QA-2
		QA-7

Figure 20 : Board 2 Iteration