Iteration #1

Step1 : Review Inputs

Analyze the drivers acquired from the Stakeholders and inquires to the PL teacher to better understand the requirements for the project.

Step2 : Iteration Goals

On this iteration we decided to address:

- domain model for the application
- constraints related with the infrastructure
- quality attributes related with the infrastructure

2.1 Goal

- CRN1: Establishing an overall initial system structure
- CRN2: Some team elements inexperience with a Spring-based systems
- CRN3: The teams reduced size
- CRN4: Allocate the tasks to the members of the team
- CRN5: Achiving the goal for the quality standards in a short amount of time
- CON1: The system is developed using Open-Source Technologies
- CON2: The application should be available in the near four weeks
- CON3: The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer
- CON4: The API is to be then accessible through a single page application (SPA)
- CON5: The application must use Spring Technology
- CON6: The system must ensure 99% of unauthorized login attempts are detected
- QA1, CON7: The application must run on several browsers and devices.
- QA2 : Usage of Domain Primitives
- QA3 : The application must be suitable for future modification
- QA4 : The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer

2.1.1 Importance to the Customer and Difficulty of Implementation according to the Architect

Scenario ID	Importance to the Customer	Difficulty of Implementation according to the Architect
CRN1	Low	Medium
CRN2	Medium	High
CRN3	Low	High
CRN4	Low	Medium
CRN5	Medium	High

Scenario ID	Importance to the Customer	Difficulty of Implementation according to the Architect
CON1	High	Low
CON2	High	High
CON3	Low	Medium
CON4	High	Low
CON5	High	High
CON6	High	High
QA1,CON7	Low	High
QA2	Low	Medium
QA3	Low	High
QA4	High	High

Step3: What to Refine

Since this is the first iteration, there are none elements to refine.

Step4: Design concepts that satisfys the selected drivers

To satisfy the given drivers we ended up choosing the three layer architecture, this means that we are going to build:

- SPA
- Backend onion architecture
- DB

Design Decisions and Location	Rationale	Alternatives
Backend	The team experience regarding Spring Boot applications is growing, therefore we decide to keep to client requirement.	None
SPA	We consider the experience of the team with such applications and concluded that Angular was the middle term for us all.	React, Vue.js, Next.js
DB	To manage all the dynamic information for the system, we decided to use JPA because it is easier to integrate with Java Spring Boot and we can easily find support for impediments during development.	Postgres, MySQL, MS SQL Server, MONGODB

Design Decisions and Location	Rationale	Alternatives
Deployment	In order to deploy, manage and scale the application we decided to used the cloud platform Heroku. This decision was based on past developments in wich the process of CI/CD was managed trough Heroku, hence the team feels more comfortable.	Firebase, AWS, Azure, Netlify
Security	Analysing the different options to manage security, JWT was found unanimously the best one. Not only because some of the elements have configured APIs using this protocol, but because the learning curve is steeper.	Oauth 2.0, Api Keys

Step5: Instantiate architectural elements, allocate responsibilities and define interfaces

Presentation Layer: Display the UI and facilitate user interaction. What the user will see from the application;

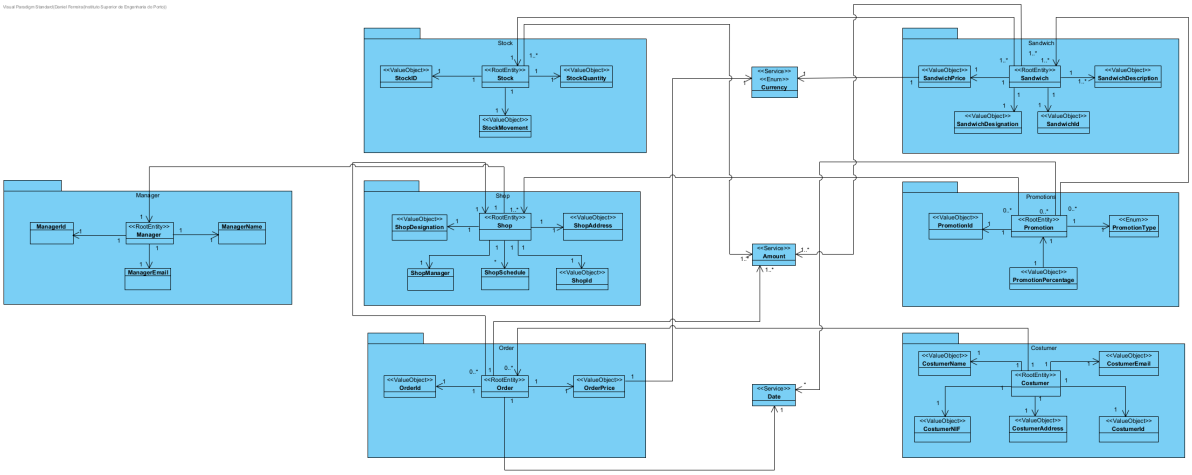
Service Layer: Middleware between Presentation and Business layer. This allows to keep the business layer intact when exposing application functionality. No business rules should be included here.

Business Layer: Application core where all the business logic and workflows are implemented. Should not be exposed to the application exterior domain, only when accessed by service layer.

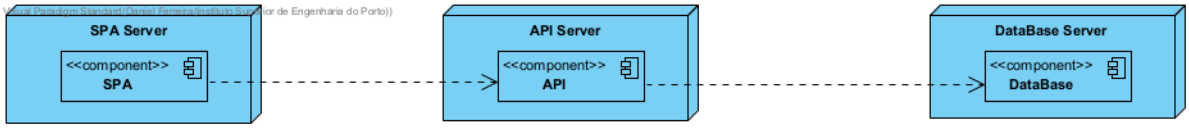
Data Layer: Data layer to abstract the business layer of the all data logic and management.

Step 6: Sketch views and record design decisions

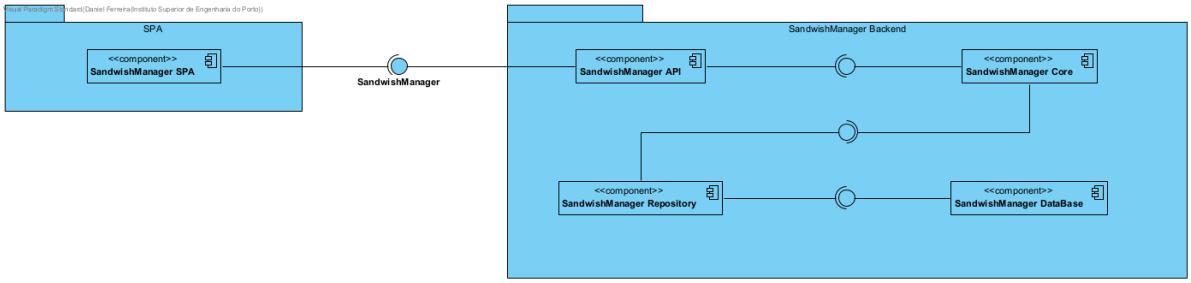
Domain Model



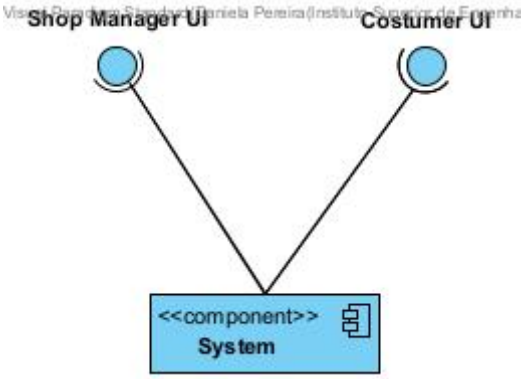
Deployment Diagram



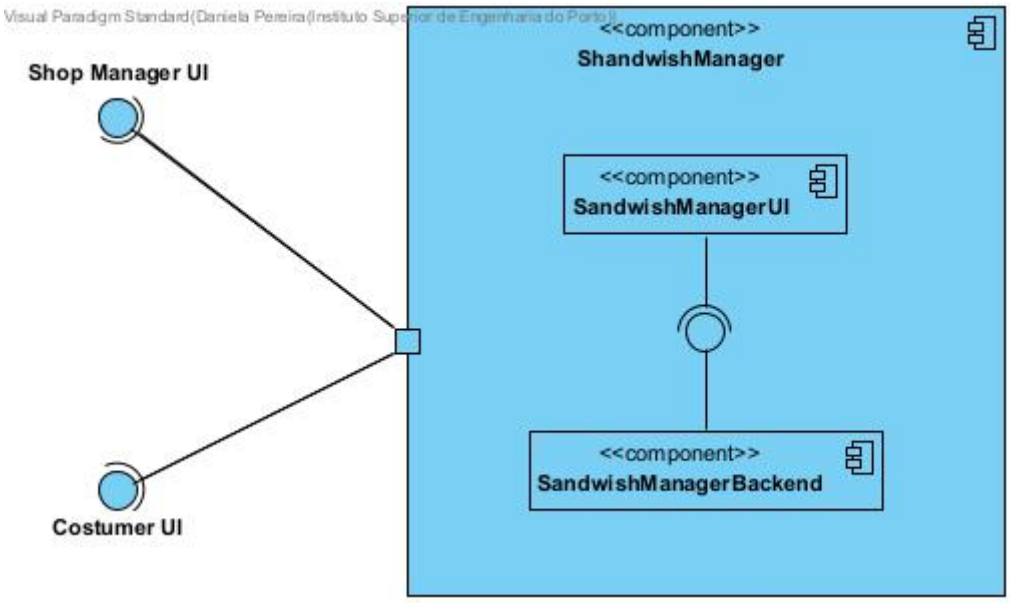
Logical Views



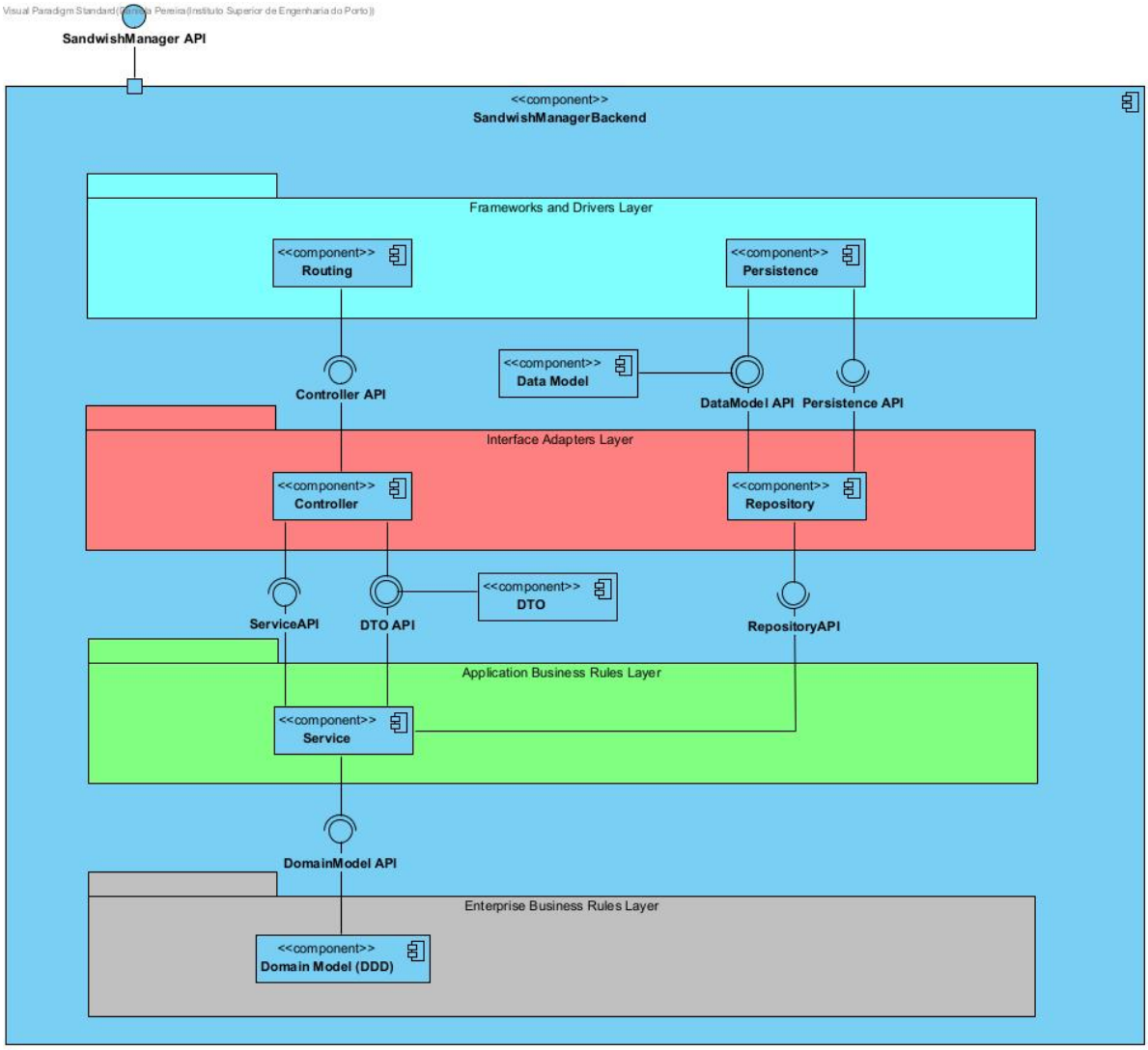
Logical View Level 1



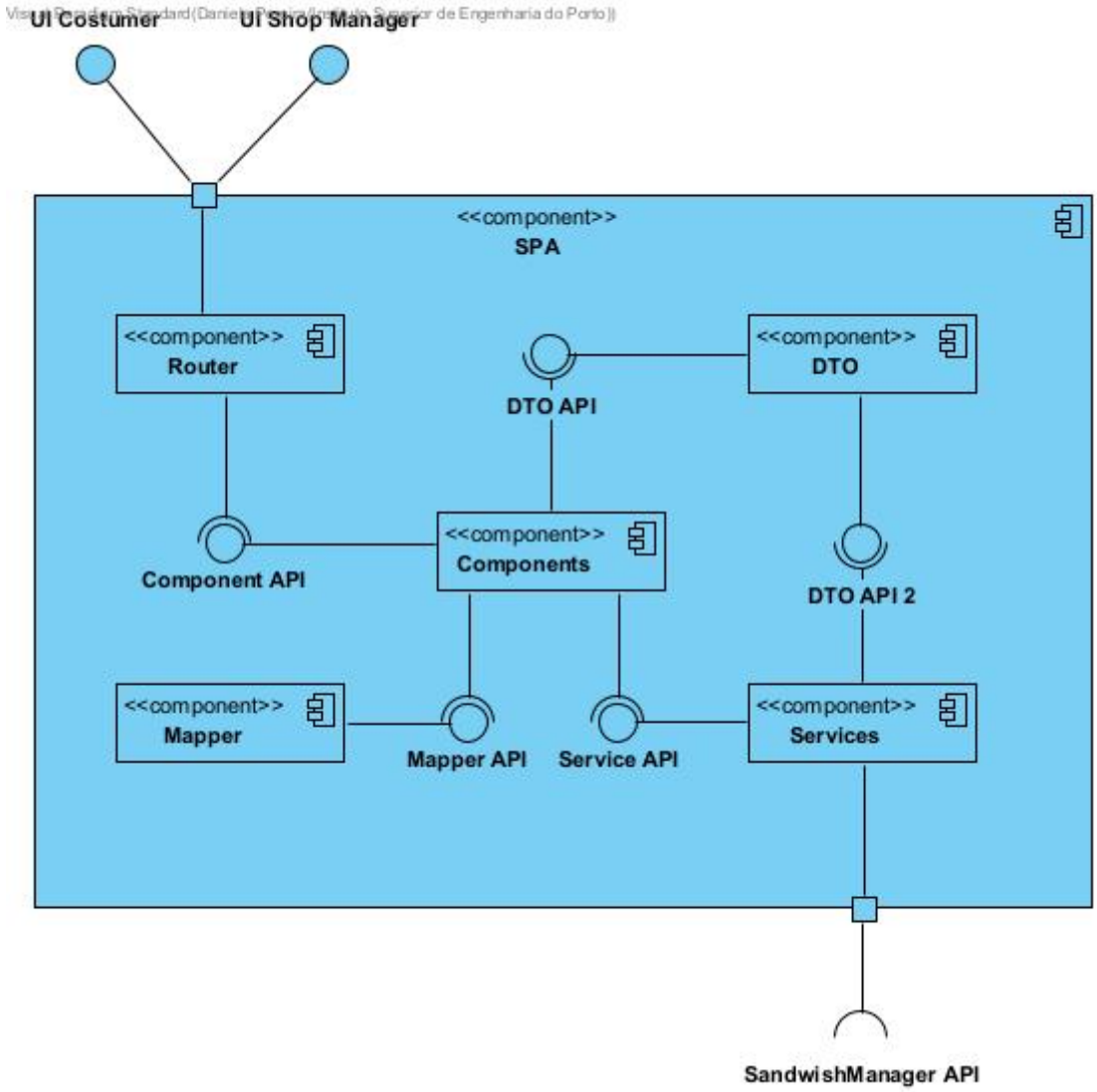
Logical View Level 2



Logical View Level 3 - Backend



Logical View Level 3 - SPA



Step 7: Analys current design and review iteration goals

	Iteration 1	
Not Addressed	Partially Addressed	Completely Addressed
CRN2	CRN4	CRN1
CRN5	CON1	CRN3
CON3	CON2	CON5
CON4	QA1,CON7	-
CON6	QA2	-
QA3	-	-
QA4	-	-

ADD - Iteration 2

Step 1: Considered Inputs

Driver Type	Description
UC's	All
Concerns	CRN1: Establishing an overall initial system structure.
	CRN2: Some team elements inexperience with a Spring-based systems
	CRN3: The teams reduced size
	CRN4 : Allocate the tasks to the members of the team
	CRN5 : Achiving the goal for the quality standards in a short amount of time
Constraints	CON1: The system is developed using Open-Source Technologies.
	CON2: The application should be available in the near four weeks.
	CON3: The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer
	CON4: The API is to be then accessible through a single page application (SPA).
	CON5 : The application must use Spring Technology
	CON6 : The system must ensure 99% of unauthorized login attempts are detected
	CON7 : The application must run on several browsers and devices

Step 2: Iteration Goal

This iteration goal is to support primary functionality.

Step 3: Elements to decompose/refine

- Decompose and refine elements from the reference architecture and deployment pattern selected in the previous iteration. Detail and document how different components from different layers
  interact and behave to enable desired application functionality.
- Further specify application presentation layer to be in line with some UI expectations.
- Domain Model

Step 4: Design Concepts

Domain model

The domain model will be specified by DDD standards with the use of Value Objects, Entities and Aggregates.
This will not only further increase the project team knowledge in the application target domain, but also to understand the best way to organize the system primary functionality.

Domain objects

Domain objects / Components that will be utilized by each UC should be specified.
This can be achieved by designing SD diagram for a specific UC as well as identifying, explaining and reasoning each one of the components that are split between the multiple application layers.

Step 5: Instantiate architectural elements, allocate responsibilities and define interfaces

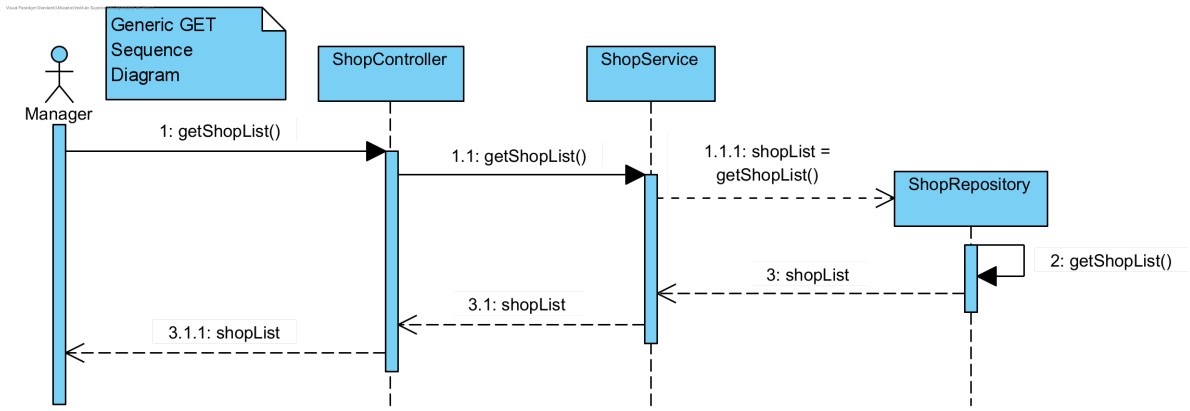
UC

Element	Responsibility
ShopController	Handle HTTP requests. Application entry point
ShopDTO	Shop deprived of any business logic that will be exposed to the exterior through the controller. Can be described as a bag of data.
ShopService	Shop service to abstract business logic and access to ShopController. This enables the change of funcionality without, necesseraly, changing the business core of the application.
ShopMapper	Handles Shop to ShopDTO conversion.
Shop	Shop domain entity representation.
ShopRepository	Shop repository. Consists in an abstraction to the data layer, enabling the application to use other database in the future if there is a need for it.

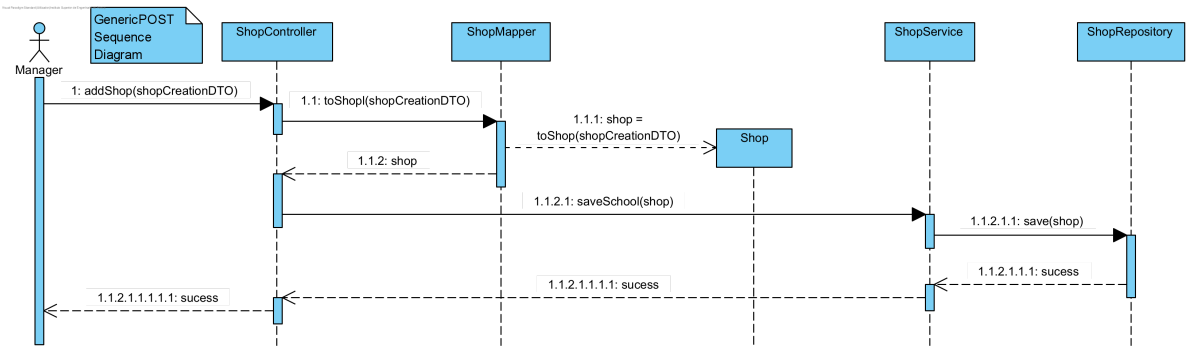
This can be extended for other UC/Aggregates.

Step 6: Sketch views and record design decisions

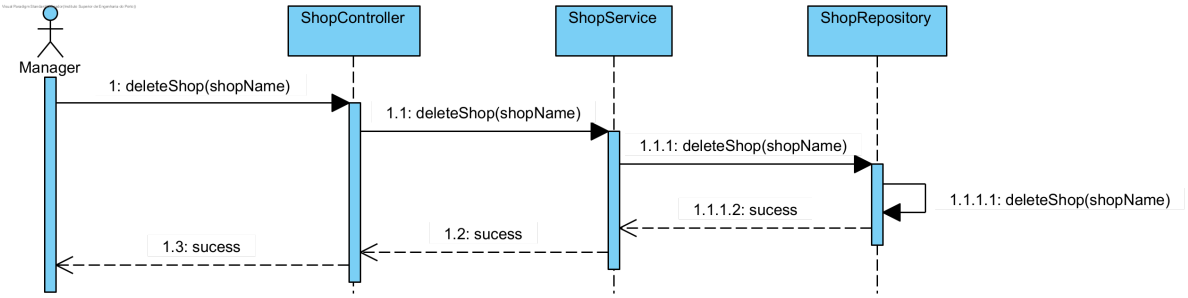
Sequence Diagram - GET



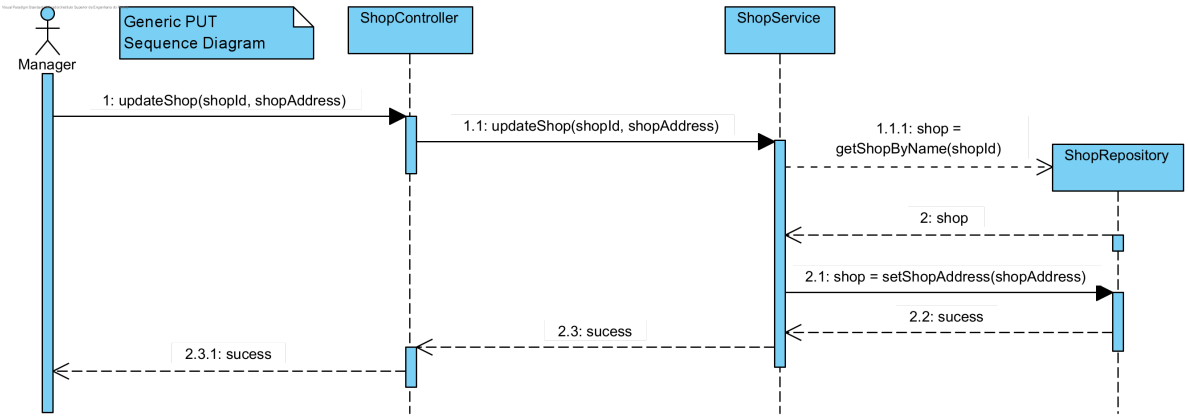
Sequence Diagram - POST



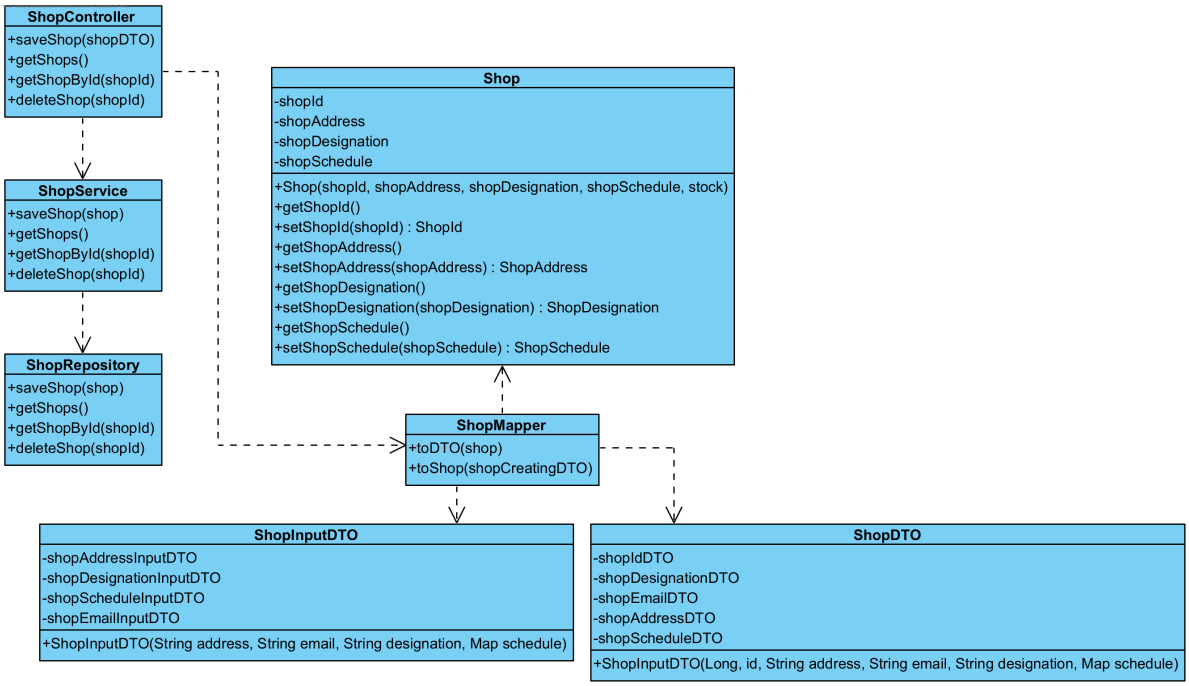
Sequence Diagram - DELETE



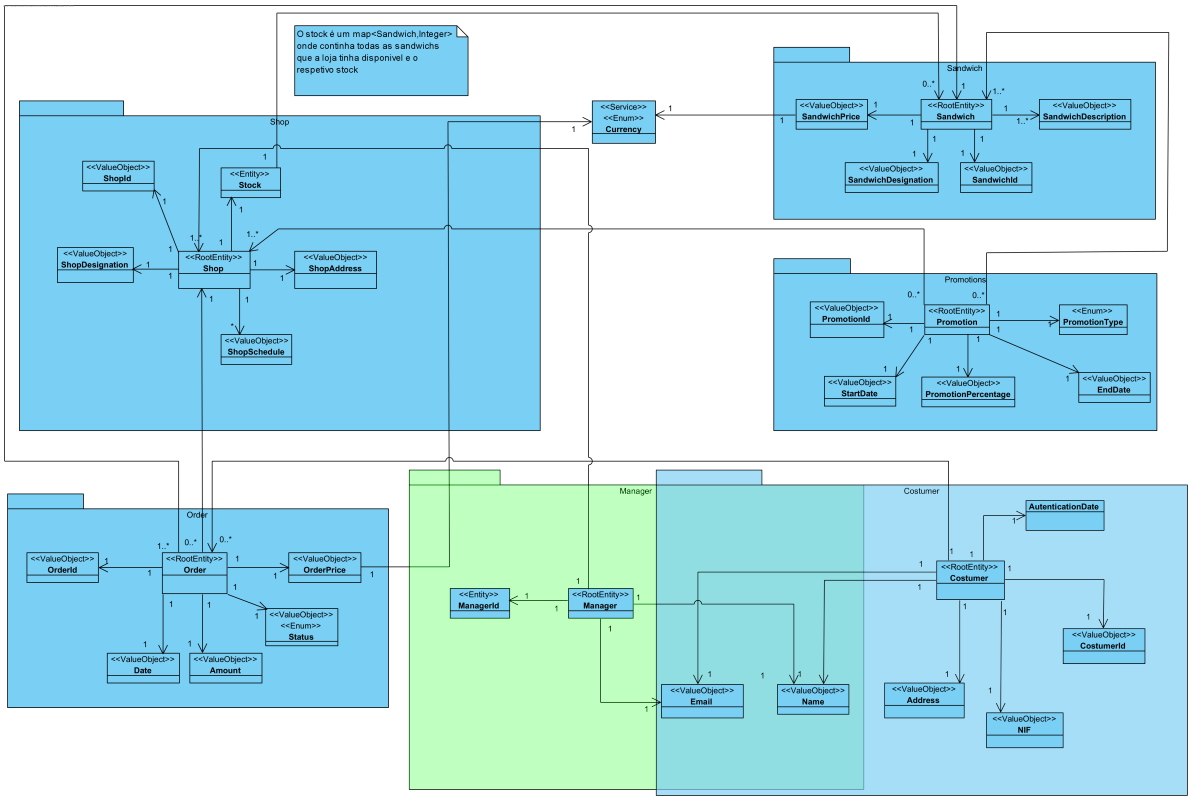
Sequence Diagram - PUT



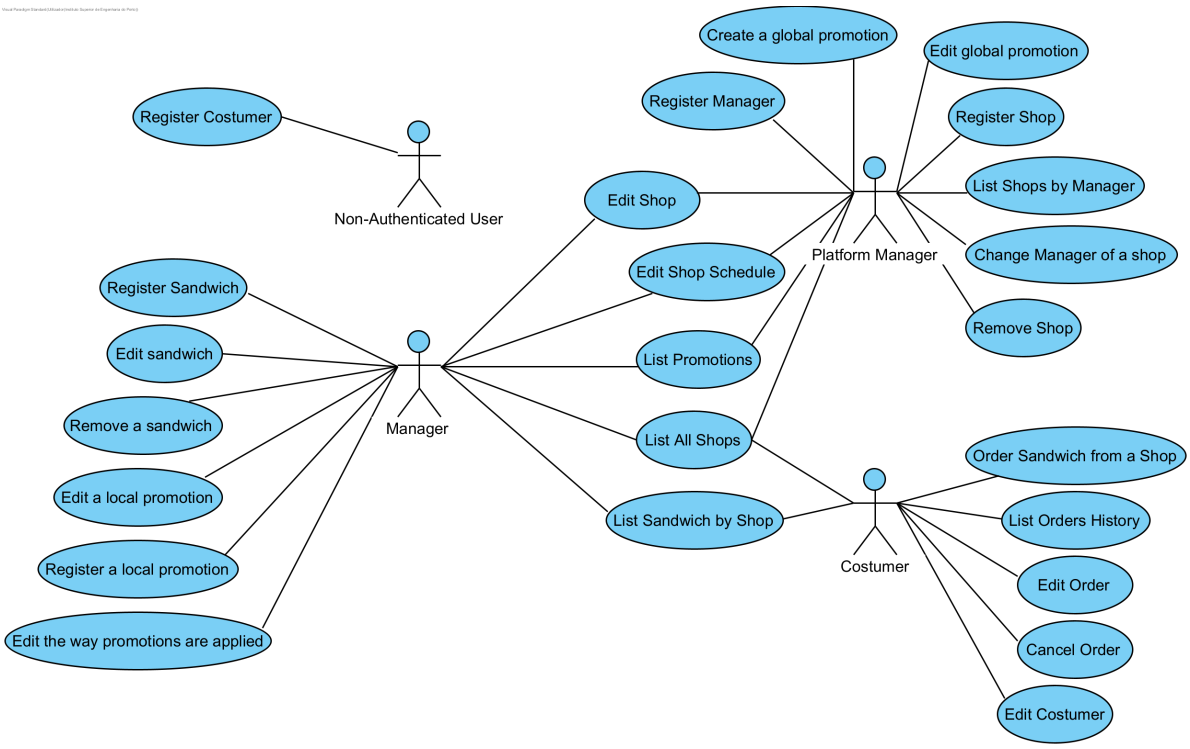
Class diagram



Domain model



Use Case Diagram



ID	Description	Actor	Priority
US1	Register in the platform as a Manager	Platform Manager	High
US2	Register in the platform as a Costumer	Non-Authenticated User	High
US3	Change Manager of a Shop	Platform Manager	High
US4	Register Shop	Platform Manager	High
US5	Edit Shop	Platform Manager, Manager	Medium
US6	Edit Costumer	Costumer	Medium
US7	Remove Shop	Platform Manager	Medium
US8	List all Shops	Platform Manager, Manager, Costumer	High
US9	Create global promotions	Platform Manager	High
US10	Edit global promotions	Platform Manager	Medium
US11	Create local Promotion	Manager	High
US12	Edit local Promotion	Manager	Medium
US13	Edit Promotion Application	Manager	Medium
US14	List Promotions	Manager, Costumer	Low
US15	Register Sandwich	Manager	High

ID	Description	Actor	Priority
US16	Edit Sandwich	Manager	Medium
US17	Remove Sandwich	Manager	High
US18	List all Sandwiches from Shop	Manager, Costumer	High
US19	Order Sandwich	Costumer	High
US20	Edit Order	Costumer	Medium
US21	Cancel Order	Costumer	High
US22	List Order History	Costumer	Low

Design decisions

Design decision	Rationale
Use of domain objects as well decomposing them between all layers	Provide at least one example of domain objects in each of the defined application layers, maybe even with a use case context. This will be important to provide context to the previous chosen architecture and pattern.
Use Spring framework	CON5
Use Angular for the front-end/ UI of the application	CON7
Class Date	The class Date is no long a Service and is an attribute from Order and Promotion. In this context, Date has some conditions and business rules that doesn't allow the group to set as a Service.
Class Amount	The class Amount is no long a Service and is an attribute from Order. In this context, Amount has some conditions and business rules that doesn't allow the group to set as a Service.
Delete Aggregate Stock	For this project, stock isn't one of the main concerns so instead of being an aggregate is an entity that belongs to the aggregate Shop
Class Email and Name	The manager and the costumer share the same conditions and business rules regarding the email and name classes. The group decided to create a single email and name class to avoid code duplication.

Quality decisions

ID	Quality Attribute	Scenario	Associated Use Stories

ID	Quality Attribute	Scenario	Associated Use Stories
QA1	Usability,Performance	The application must run on several browsers and devices	All User Stories
QA2	Security, Modifiability	Usage of Domain Primitives	All User Stories
QA3	Modifiability	The application must be suitable for future modification	All User Stories
QA4	Testability, Performance, Modifiability,Maintainbility	The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer	All User Stories

Step 7: Perform analysis of current design and review iteration goal and design objectives

Updated Kanban board

	Iteration 2	
Not Addressed	Partially Addressed	Completely Addressed
CRN2	CON1	CRN1
CRN5	CON2	CRN3
CON3	QA3	CRN4
CON4	-	CON5
CON6	-	QA1,CON7
QA4	-	QA2

ADD - Iteration 3

Step 1: Considered Inputs

Driver type	Description
Concerns	CRN5: Achiving the goal for the quality standards in a short amount of time
Quality attributes	QA4 : The application must be suitable for future modification
	QA5 : The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer

Step 2: Iteration Goal

- In this third iteration the goal is to re-evaluate the structure to confirm that supports all the primary functionalities;
- Implement all the primary functionalities;
- Integrate the Sonargraph-Explorer;
- Guarantee that the code standards are at least 70%.

Step 3: Elements to decompose/refine

- Decide how functionality tests will be addressed.

Step 4: Design Concepts

Design decisions and location	Rationale
Introduce modifiability tactics, more precisely, introduce Defer binding time - Polymorphism tactic to application repository / data layer	This will allow for database technology to be changed later on the application lifecycle without greatly impacting previous functionality
Introduce high cohesion and low coupling tactics	This will allow the solution to be more robust and easier to maintain and modify.
Introduce Builder pattern, fail fast, failing for a good state and immutability	Satisfy in order to ensure a more secure solution design

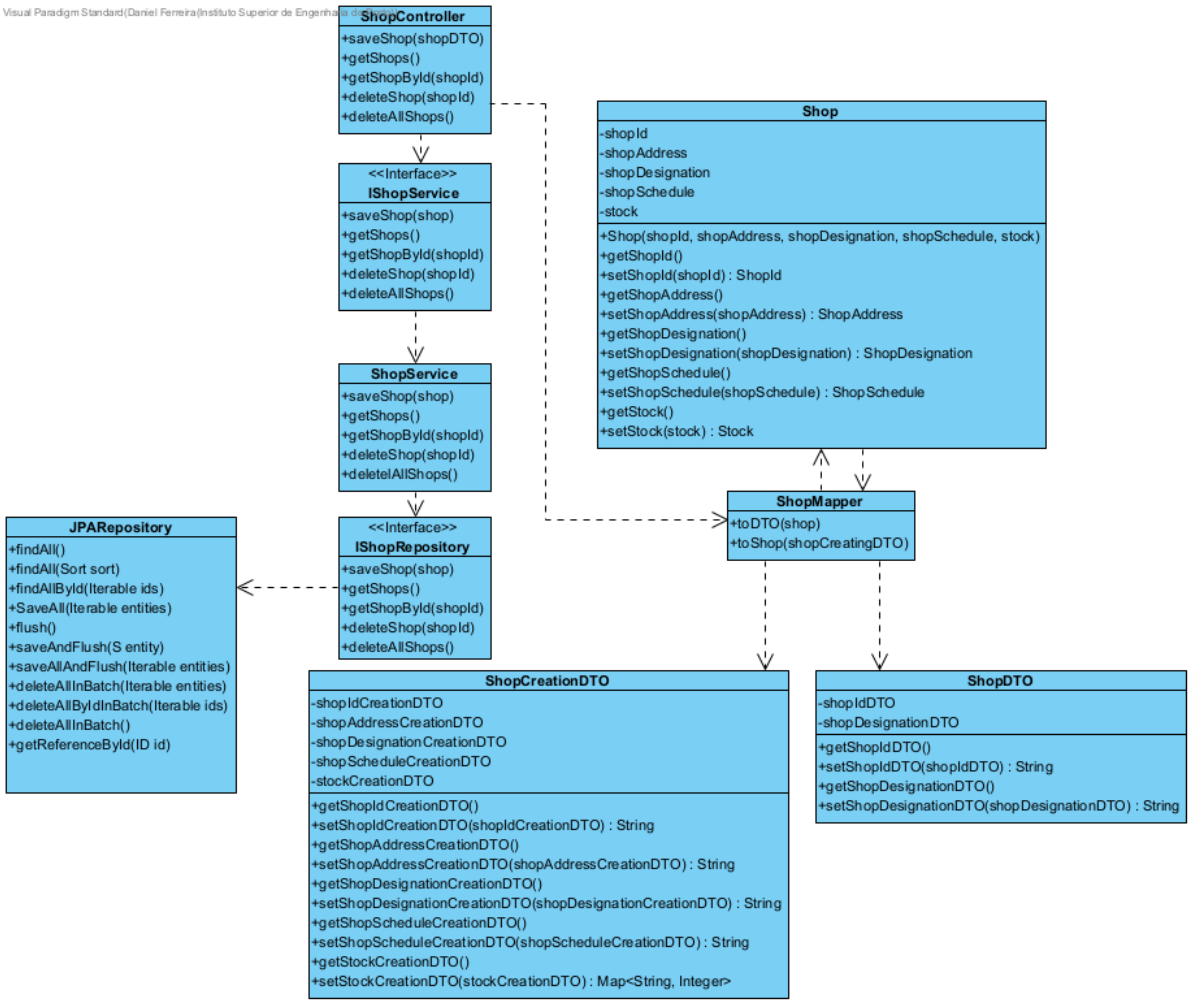
Step 5: Instantiate architectural elements, allocate responsibilities and define interfaces

Shop Aggregate

Element	Responsibility
ShopController	Example of a controller implementation, responsible for the endpoints that connects the solution with it's exterior requests.
IShopService	Example of a service's interface implementation, which adds another layer of abstraction guaranteeing that the Controller and the Service do not contact with each other.
ShopService	Example of a service implementation, responsible for managing the requests and proccessing them.
IShopRepository	This will now be an interface that should be implemented for each of the diferent data layer technologies used.
ShopRepository	Example of a repository implementation. In this case this is a JPA repository.
ShopEntity	JPA Entity. Main objective is to detach JPA annotations from DDD entities.
ShopDTO	Data Transfer Object responsible to circulate throughout the solution. Represents another layer of abstraction since does not carry sensible information.
ShopMapper	Class do map domain entities to JPA entities.
ShopTests	Unit tests using JUnit to test sandwich functionality.

Step 6: Sketch views and record design decisions

Class diagram updated - Shop

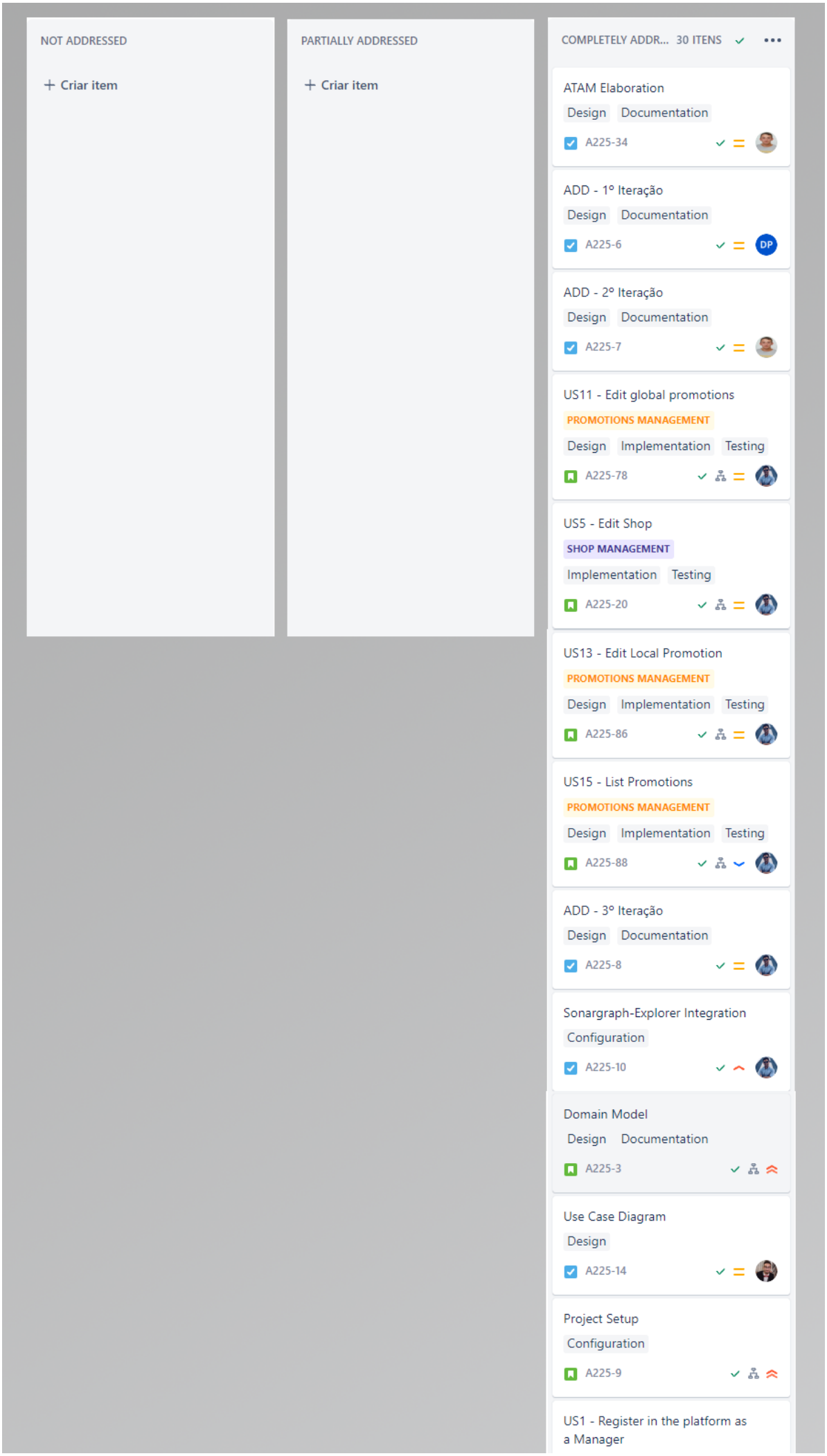


Design decisions

Design decision	Rationale
Widespread use of interfaces and implementation of these for specific technology purprose (SandwichRepository-SandwichRepositoryJPA)	QA4, make the application as modifiable as possible allowing the introduction of new database technologies later on the lifecycle.
Develop unit tests	CRN5, to ensure the core application code is behaving and most probably will behave as expected.
Use of ValueObjects from DDD to introduce some type of immutability	QA2, follow developed domain model.
Fail fast	Detect input errors the fast as possible to ensure no invalid value get to the lower layers of the solution.
JPA Criteria API Queries	use of JPA Criteria API queries when doing database queries which increase application security when it comes to SQL injection

Step 7: Perform analysis of current design and review iteration goal and design objectives

Updated Kanban board


NOT ADDRESSED
+ Criar item
PARTIALLY ADDRESSED
+ Criar item
COMPLETELY ADDR... 30 ITENS
ATAM Elaboration
Design Documentation
A225-34
ADD - 1º Iteração
Design Documentation
A225-6
DP
ADD - 2º Iteração
Design Documentation
A225-7
US11 - Edit global promotions
PROMOTIONS MANAGEMENT
Design Implementation Testing
A225-78
US5 - Edit Shop
SHOP MANAGEMENT
Implementation Testing
A225-20
US13 - Edit Local Promotion
PROMOTIONS MANAGEMENT
Design Implementation Testing
A225-86
US15 - List Promotions
PROMOTIONS MANAGEMENT
Design Implementation Testing
A225-88
ADD - 3º Iteração
Design Documentation
A225-8
Sonargraph-Explorer Integration
Configuration
A225-10
Domain Model
Design Documentation
A225-3
Use Case Diagram
Design
A225-14
Project Setup
Configuration
A225-9
US1 - Register in the platform as a Manager

MANAGER MANAGEMENT

Implementation Testing

A225-69

US3 - Change Shop Manager

MANAGER MANAGEMENT

A225-72

US4 - Register Shop

SHOP MANAGEMENT

Implementation Testing

A225-18

US6 - Edit Costumer

COSTUMER MANAGEMENT

Implementation Testing

A225-174

US7 - Remove Shop

SHOP MANAGEMENT

Implementation Testing

A225-19

US8 - List All Shops

SHOP MANAGEMENT

Implementation Testing

A225-21

US10 - Create global promotions

PROMOTIONS MANAGEMENT

Implementation Testing

A225-77

US12 - Create Local Promotion

PROMOTIONS MANAGEMENT

Design Implementation Testing

A225-70

US16 - Register Sandwich

SANDWICH MANAGEMENT

Implementation Testing

A225-37

US17 - Edit Sandwich

SANDWICH MANAGEMENT

Implementation Testing

A225-38

US18 - Remove Sandwich

SANDWICH MANAGEMENT

Implementation Testing

A225-36

US19 - List all Sandwiches from Shop

SANDWICH MANAGEMENT

Implementation Testing

A225-39

US20 - Create Order Sandwich

ORDER MANAGEMENT

Implementation Testing

A225-89 DP

US21 - Approve Order

ORDER MANAGEMENT

Design Implementation Testing

A225-90 DP

US22 - Cancel Order

ORDER MANAGEMENT

Design Implementation Testing

A225-91 DP

US23 - List Order History

ORDER MANAGEMENT

Design Implementation Testing

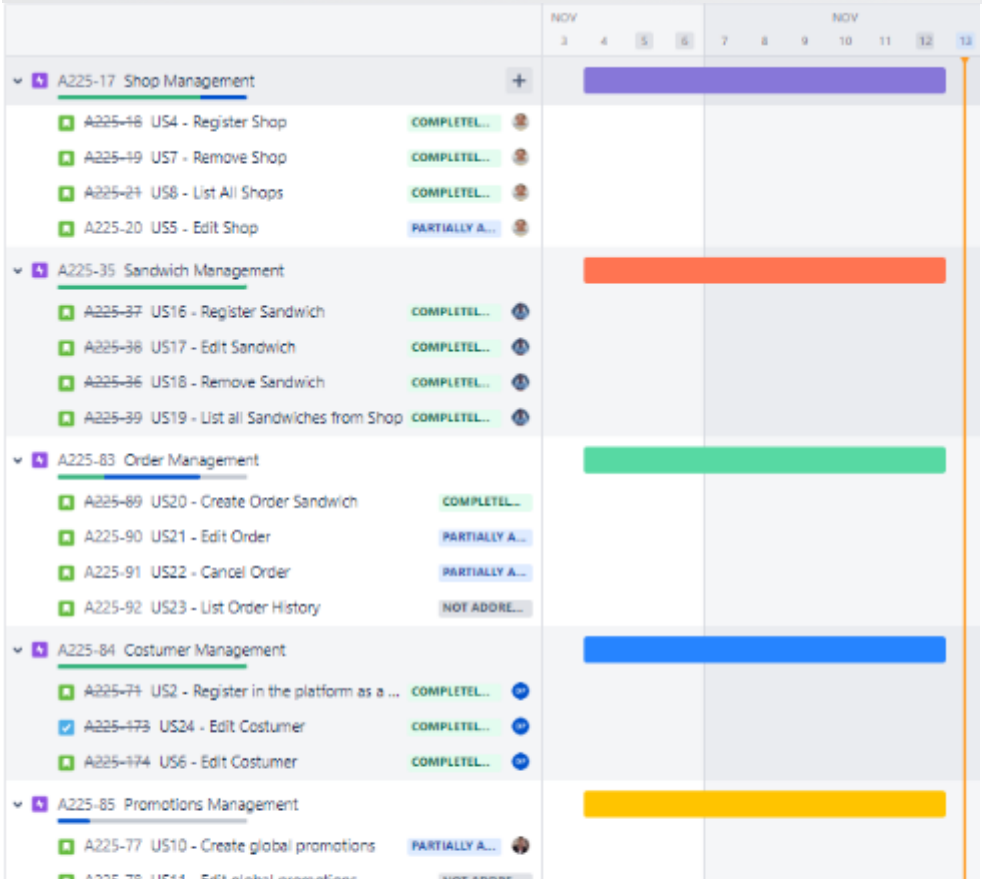
A225-92 DP

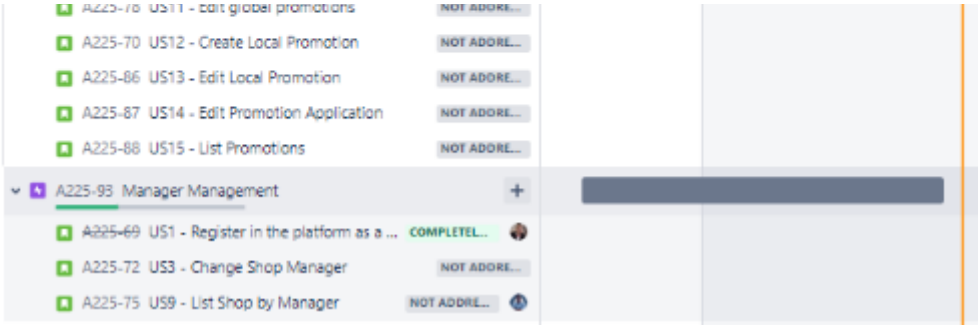
US14 - Edit Promotion Application

PROMOTIONS MANAGEMENT

Design Implementation Testing

A225-87





	Iteration 3	
Not Addressed	Partially Addressed	Completely Addressed
-	QA1,CON7	CRN1
-	-	CON1
-	-	CRN2
-	-	CON2
-	-	CRN5
-	-	QA4
-	-	CON3 , QA5
-	-	CRN3
-	-	CRN4
CON4	-	CON5
CON6	-	QA2
-	-	-
-	-	-

Architecture Tradeoff Analysis Method Document

Present ATAM Document

The ATAM process consists of gathering stakeholders together to analyze business drivers (system functionality, goals, constraints, desired non-functional properties) and from these drivers extract quality attributes that are used to create scenarios. These scenarios are then used in conjunction with architectural approaches and architectural decisions to create an analysis of trade-offs, sensitivity points, and risks (or non-risks). This analysis can be converted to risk themes and their impacts whereupon the process can be repeated. With every analysis cycle, the analysis process proceeds from the more general to the more specific, examining the questions that have been discovered in the previous cycle, until such time as the architecture has been fine-tuned and the risk themes have been addressed.

Present Business Drivers

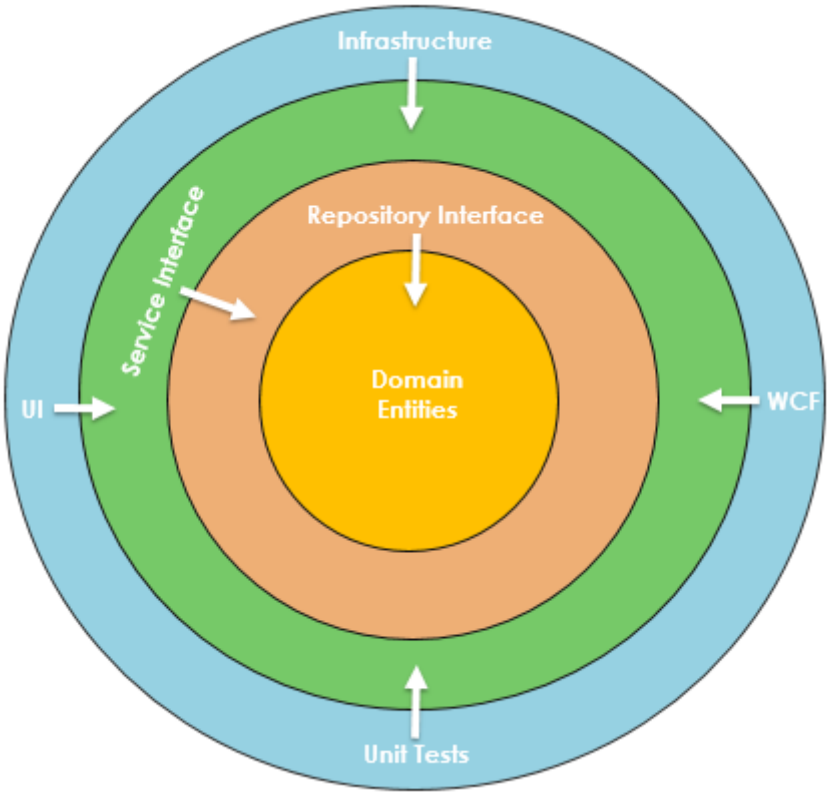
A very innovative organization that aggregates many schools at Porto has become concerned about the food available for its academic community, especially its students, many of whom are displaced from their relatives' homes, have little economic means, and are not skilled in preparing and cooking healthy meals.

The institution stopped having vending machines that offered food of poor nutritional quality. Instead, affordable meals began to be served in the canteen with delicious lunches and dinners. The sandwiches proved to be of great value, so Gorgeous Sandwich, a company dedicated to healthy sandwiches, was hired to open a new space and sell sandwiches.

Gorgeous Sandwich proved to be a great value for the students due to their nutritious and well made sandwiches.

Present The Architecture

After an analysis of the problem, the group decided to use Onion Architecture as a solution. Onion Architecture addresses the challenges faced with 3-tier and n-tier architectures, and to provide a solution for common problems. Onion Architecture layers interact to each other by using the Interfaces. Onion Architecture is comprised of multiple concentric layers interfacing each other towards the core that represents the domain. The architecture does not depend on the data layer as in classic multi-tier architectures, but on the actual domain models. At the center of Onion Architecture is the domain model, which represents the business and behavior objects. Around the domain layer are other layers, with more behaviors. The following image is an example of the Onion Architecture:



Domain Entities

At the center part of the Onion Architecture, the domain layer exists; this layer represents the business and behavior objects. The idea is to have all of your domain objects at this core. It holds all application domain objects.

Repository Layer

This layer creates an abstraction between the domain entities and business logic of an application. In this layer, we typically add interfaces that provide object saving and retrieving behavior typically by involving a database. Creates a generic repository, and add queries to retrieve data from the source, map the data from data source to a business entity, and persist changes in the business entity to the data source.

Service Layer

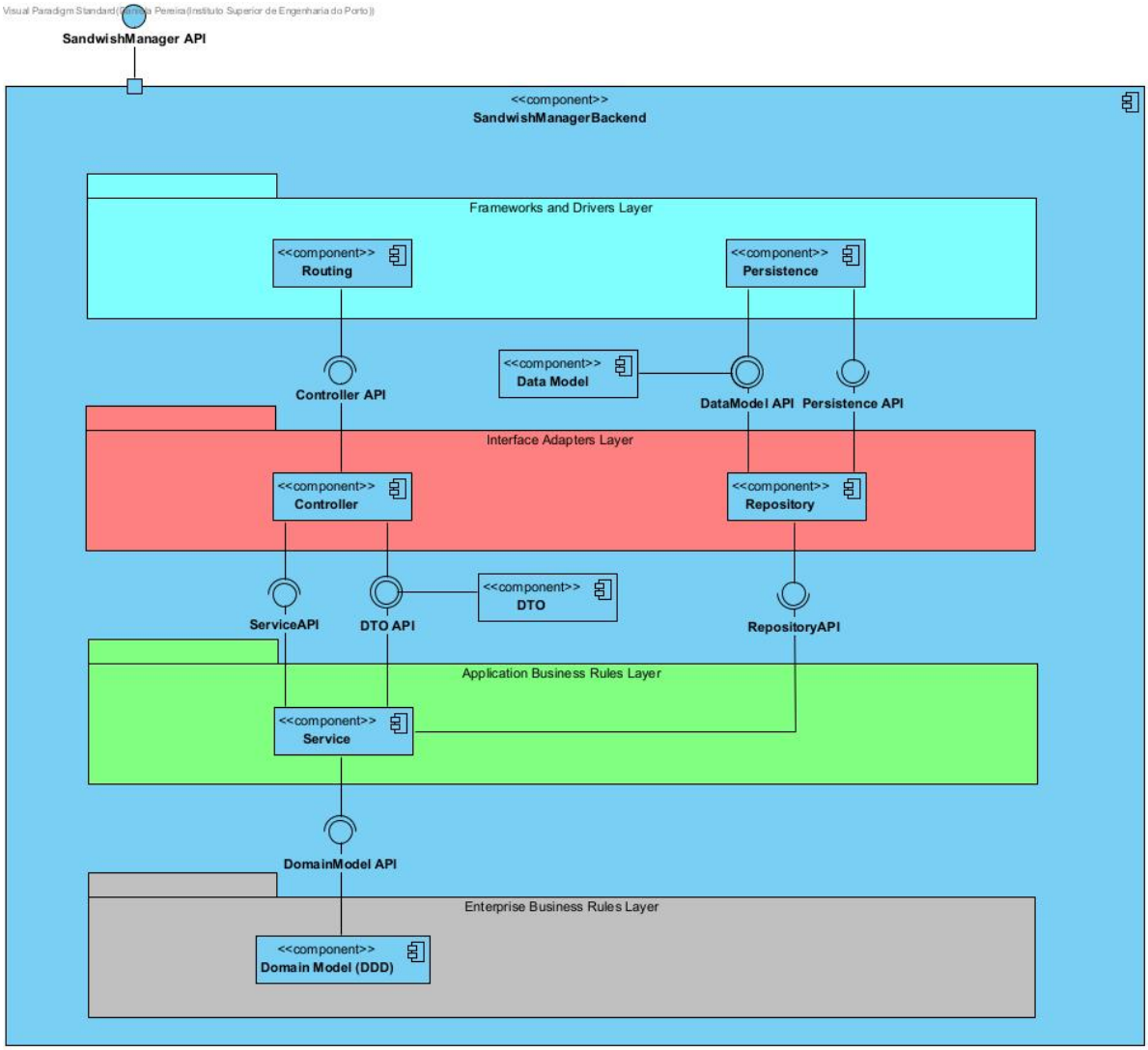
The Service layer holds interfaces with common operations, such as Add, Save, Edit, and Delete. Also, this layer is used to communicate between the UI layer and repository layer.

UI Layer

It's the outer-most layer, and keeps peripheral concerns like UI and tests. For a Web application, it represents the Web API or Unit Test project.

Identify Architectural Approach

For this solution the onion architecture was used, and the following image shows the various layers of the project developed.



The Domain Model component is divided between Value Objects and Entities, thus following the Driven Domain Design (DDD) pattern. There are Service and Repository interfaces that link the different layers. There is also the database component which the group decided to use MySQL since it is an open-source tool, a requirement of the statement, and is a relational database. The DTO pattern was also used in order to simplify the data input into the system.
The controller component is responsible for receiving CRUD (Create, Read, Update, Delete) requests from the client.

There is a selection of deployment patterns to choose from:

- Non distributed deployment:
- Distributed deployment:
- Client-server deployment.
- n-tier deployment(2-tier,3-tier,4-tier).

Due to concern CRN3 and condition CON2, the application must be developed in a short span of time (4 weeks) with a short team so a deployment pattern which allowed for all architecture layers to be implemented simultaneously would greatly reduce the risk of not delivering in time.
Taking the previous statement into consideration, the most adequate for this software solution would be a 2-Tier deployment where Client and database are deployed separately. This would allow all these layers can be developed simultaneously, security could be implemented and modifiability would increase as well.

Analyze Architectural Approach

Domain Model

Entity:

In domain-driven design, an entity is a representation of an object in the domain. It is defined by its identity, rather than its attributes. It encapsulates the state of that object through its attributes, including the aggregation of other entities, and it defines any operations that might be performed on the entity.

In the following image is an example of an entity of the solution. Each entity represents one table in the database. The primary key of the table is the entity id which is generated automatically. The column names are defined by the Column tag and the various properties such as unique or nullable.

```
@Entity
@Table(name = "sandwich", schema = "arqsoftdb")
public class Sandwich {

    @Column(name = "sandwichPrice", nullable = false)
    public Price sandwichPrice;
    @Column(name = "sandwichDesignation", nullable = false, unique = true)
    @NotNull(message = "Sandwich Designation may not be null")
    public Designation sandwichDesignation;
    @Column(name = "sandwichDescription", nullable = false)
    public SandwichDescription sandwichDescription;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", nullable = false)
    private Long sandwichId;
```

Domain

Value Object is an object that represents a concept from your problem Domain. It is important in DDD that Value Objects support and enrich Ubiquitous Language of your Domain. They are not just primitives that represent some values - they are domain citizens that model behaviour of your application.

Value Object classes all have an empty constructor with the keyword protected. The filled constructors are instantiated with the set method, and this method is private and is where the business checks are done.

```java
@Embeddable
public class Email {

    private String email;


    protected Email() {
    }

    public Email(String email) {
        setEmail(email);
    }

    public String getEmail() {
        return email;
    }

    private void setEmail(String email) {
        Assert.isTrue(email.matches(Constants.EMAIL_REGEX), message: "Email is incorrect.");
        this.email = Objects.requireNonNull(email);
    }
}
```

Controller

Controller is a class that handles user requests. It retrieves data from the Model and renders view as response. The controller maps requested URL's to the classes that are referred to as controllers.

Each controller class has his CRUD operations. In the mapping tags the url of the route is presented so that each CRUD operation can be accessed.

```java
@PostMapping("/addSandwich")
public ResponseDTO addSandwich(@RequestBody InputSandwichDTO inputSandwichDTO) {
    return sandwichService.addSandwich(inputSandwichDTO);
}

@GetMapping("/getAllSandwich")
public ResponseDTO getAllSandwich() {
    return sandwichService.listAll();
}

@GetMapping("/getSandwichById/{sandwichId}")
public ResponseDTO getSandwichById(@RequestParam Long sandwichId) {
    return sandwichService.listBySandwichId(sandwichId);
}

@PutMapping("/editSandwich/{sandwichId}")
public ResponseDTO editSandwich(@PathVariable("sandwichId") Long sandwichId, @RequestBody InputSandwichDTO inputSandwichDTO) {
    return sandwichService.editSandwich(sandwichId, inputSandwichDTO);
}

@DeleteMapping("/deleteSandwich/{sandwichId}")
public ResponseDTO deleteSandwich(@PathVariable("sandwichId") Long sandwichId) {
    return sandwichService.deleteSandwich(sandwichId);
}
```

Service

A service layer is an additional layer in the application that mediates communication between a controller and repository layer. The service layer contains business logic. In particular, it contains validation logic.

The service classes are the controller's CRUD request handler. In the *addSandwich* example, an input is given and will be converted to a domain instance and then this instance is stored in the database. At the end it will return a *ResponseDTO* object consisting of a success code and an operation, or in case of failure an error code and error message.

All other CRUD operations are very similar to the one described above, but for the respective operation.

```java
@Autowired
private SandwichRepository sandwichRepository;

@Autowired
private SandwichMapper sandwichMapper;

@Override
public ResponseDTO addSandwich(InputSandwichDTO inputSandwichDTO) {
    try {
        Sandwich sandwich = this.sandwichMapper.toSandwich(inputSandwichDTO);
        Sandwich savedSandwich = this.sandwichRepository.save(sandwich);
        return new ResponseDTO(StatusCode.CREATED.getCode(), this.sandwichMapper.tosandwichDTO(savedSandwich));
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
}

@Override
public ResponseDTO listAll() {
    try {
        return new ResponseDTO(StatusCode.OK.getCode(), this.sandwichRepository.findAll());
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
}

@Override
public ResponseDTO listBySandwichId(Long sandwichId) {
    try {
        Optional<Sandwich> optionalSandwich = this.sandwichRepository.getSandwichBySandwichId(sandwichId);
        if (sandwichId != null && optionalSandwich.isPresent()) {
            return new ResponseDTO(StatusCode.OK.getCode(), optionalSandwich.get());
        }
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "The sandwich with sandwich id %s does not
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
```

```java
@Override
public ResponseDTO editSandwich(Long sandwichId, InputSandwichDTO inputSandwichDTO) {
    try {
        Optional<Sandwich> optionalSandwich = this.sandwichRepository.findById(sandwichId);
        if (!optionalSandwich.isPresent()) {
            return new ResponseDTO(StatusCode.NOT_FOUND.getCode(), String.format(format: "There is no sandwich with the id: %s", s
        }
        String sandwichDesignation = inputSandwichDTO.sandwichDesignation;
        if (sandwichDesignation != null) {
            Optional<Sandwich> optionSandwichDesignation = this.sandwichRepository.findSandwichBySandwichDesignation(new Designat
            if (optionSandwichDesignation.isPresent()) {
                return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "There is already a sandwich with t
            }
        }
        Sandwich sandwich = optionalSandwich.get();
        Sandwich toBeSavedSandwich = this.sandwichMapper.toSandwich(new InputSandwichDTO(inputSandwichDTO.sandwichPrice != null ?
        this.sandwichRepository.delete(sandwich);
        Sandwich savedSandwich = this.sandwichRepository.save(toBeSavedSandwich);
        return new ResponseDTO(StatusCode.UPDATED.getCode(), this.sandwichMapper.tosandwichDTO(savedSandwich));
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
}

@Override
public ResponseDTO deleteSandwich(Long sandwichId) {
    try {
        Optional<Sandwich> optionalSandwich = this.sandwichRepository.findById(sandwichId);
        if (!optionalSandwich.isPresent()) {
            return new ResponseDTO(StatusCode.NOT_FOUND.getCode(), String.format(format: "There is no sandwich with the id: %s", s
        }
        this.sandwichRepository.deleteById(sandwichId);
        return new ResponseDTO(StatusCode.DELETED.getCode(), String.format(format: "Sandwich successfully deleted!"));
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
}
```

Repository

The repository is used to create an abstraction layer between the data access layer and the business logic layer of an application. Implementation of repository patterns can help to abstract your application from changes in the data store and can facilitate automated unit testing.

The Repository class extends JpaRepository and often uses the methods implemented by the interface, such as findAll(), findById() and deleteById().
Often it is necessary to build a query in order to get the intended results. In the following image is an example of a query. The nativeQuery tag is used to be able to build the query in SQL instead of JPQL. The Modifying and Transactional tags are used for update and delete operations. The Param tag defines the name given to the attribute that will be inserted in the query.

```java
@Repository
public interface ScheduleRepository extends JpaRepository<ShopSchedule, Long> {

    @Modifying
    @Transactional
    @Query(value = "DELETE ss.* FROM shop_schedule ss INNER JOIN shop s ON ss.shop_id = s.shop_id WHERE (s.shop_id =:id)", nativeQuery = true)
    void deleteScheduleByShopId(@Param("id") Long shopId);

}
```

Mapper

The mapper class is responsible for transforming domain objects into DTO and vice versa. This class is called when it is necessary to transform the DTO into dominium in order to be able to perform operations in the program, and from dominium into DTO when returning values to the user.

```java
@Component
public class SandwichMapper {

    public SandwichDTO tosandwichDTO(Sandwich sandwich) {
        return new SandwichDTO(sandwich.getSandwichId(), sandwich.getSandwichPrice().getPrice(), sandwich.getSandwichDesignation().getDesignation(),
                sandwich.getSandwichDescription().getDescription());
    }

    public Sandwich toSandwich(InputSandwichDTO inputSandwichDTO) {
        return new Sandwich(new Price(inputSandwichDTO.sandwichPrice), new Designation(inputSandwichDTO.sandwichDesignation),
                new SandwichDescription(inputSandwichDTO.sandwichDescription));
    }

}
```

DTO

DTO stands for Data Transfer Object, which is a design pattern. The DTO pattern serves to simplify the data, so that it is easier to then return the data to the user. It is also used as an input for data that is later converted to a domain object by the mapper.

```java
public class SandwichDTO{

    public long sandwichId;
    public String sandwichPrice;
    public String sandwichDesignation;
    public String sandwichDescription;

    public SandwichDTO(long sandwichId, String sandwichPrice, String sandwichDesignation, String sandwichDescription) {
        this.sandwichId = sandwichId;
        this.sandwichPrice = sandwichPrice;
        this.sandwichDesignation = sandwichDesignation;
        this.sandwichDescription = sandwichDescription;
    }
}
```

Constants

The class constants is for registering all constants and checks via REGEX

```
public class Constants {

    public static final String NAME_REGEX = "[A-Z][a-z]+";
    public static final String EMAIL_REGEX = "[a-z]+@[a-z]+.[a-z]+";
    public static final String NIF_REGEX = "^[0-9]{9}$";
    public static final String PRICE_REGEX = "[0-9]+.[0-9]{2}$";
    public static final String DEFAULT_INPUT_REGEX = "^[-, a-zA-Z0-9]+$";
    public static final String DATE_REGEX = "^((19|2[0-9])[0-9]{2})-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|3[01])$";
    public static final String SCHEDULE_REGEX = "(0?[0-9]|1[0-9]|2[0-3])h([0-5][0-9])-(0?[0-9]|1[0-9]|2[0-3])h([0-5][0-9])";
    public static final int MIN_CHARACTERRS_ADDRESS = 10;
    public static final int MIN_CHARACTERS_DESIGNATION = 10;
    public static final int MIN_CHARACTERS_DESCRIPTION = 15;
    public static final String[] DAY_STRINGS = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday",
            "Sunday" };
}
```

Quality Assurance

ID	Quality Attribute	Scenario	Associated Use Stories
QA1	Usability,Performance	The application must run on several browsers and devices	All User Stories
QA2	Security, Modifiability	Usage of Domain Primitives	All User Stories
QA3	Modifiability	The application must be suitable for future modification	All User Stories
QA4	Testability, Performance, Modifiability, Maintainbility	The system must achive at least 70% of the level calculated for the code quality standards, through the Sonargraph-Explorer	All User Stories

The table above shows the quality standards applied in the solution.

Regarding QA1, the application works via localhost and Swagger. Swagger is a platform whose function is to receive URL's and through them make REST requests to the solution. Besides this, it is capable of automatically generating API documentation.
This requirement was not completely fulfilled, because although it is possible to make requests through different computers and in different browsers, the application is not deployed at the moment.

Regarding QA2 this was successfully achieved, since the system has value objects that are invariant and the checks on the constructors of the value objects are performed. An example of a Domain Property is the class NIF or Date, since these have unique and invariant checks.

Using SonarGraph it is possible to ascertain that the developed code has 90.48% maintainbility, which makes QA3 and QA4 successfully met. It can be concluded that good programming practices were used.

CWT Mitigations

CWE-476: NULL Pointer Dereference

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

```
private void setCostumerNIF(String costumerNIF) {
    Assert.isTrue(costumerNIF.matches(Constants.NIF_REGEX), "NIF should only have 9 numbers.");
    this.costumerNIF = Objects.requireNonNull(costumerNIF);
}
```

The requireNonNull method is used to check that the value entered is not null.

CWE-766: Critical Data Element Declared Public

The software declares a critical variable, field, or member to be public when intended security policy requires it to be private.

```
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name = "id", nullable = false)
private Long id;

@Column(name = "name", nullable = false)
private Name costumerName;

@Column(name = "email", nullable = false)
private Email costumerEmail;

@Column(name = "nif", nullable = false)
private CostumerNIF costumerNIF;

@Column(name = "address", nullable = false)
private Address costumerAddress;

protected Costumer() {
}
```

In all classes, their respective variables are all private so as not to be a vulnerability for the system.

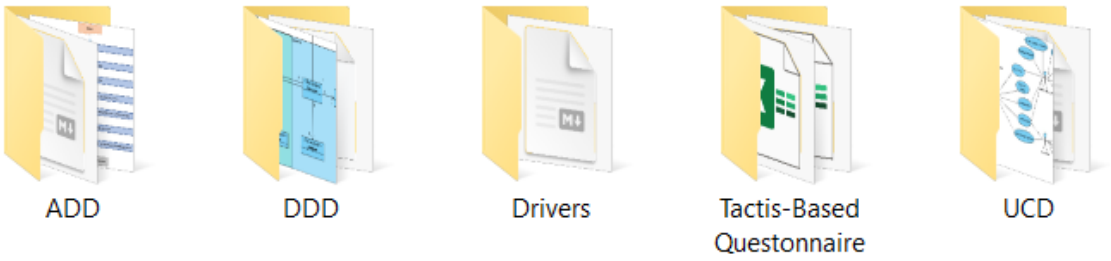
CWE-1099: Inconsistent Naming Conventions for Identifiers

The product's code, documentation, or other artifacts do not consistently use the same naming conventions for variables, callables, groups of related callables, I/O capabilities, data types, file names, or similar types of elements.

All files and variables follow the same predefined nomenclature

CWE-1053: Missing Documentation for Design

The product does not have documentation that represents how it is designed.



CRUD operation diagrams have been made, as all documentation regarding the DDD and the three ADD iterations.

CWE-1054: Invocation of a Control Element at an Unnecessarily Deep Horizontal Layer

The code at one architectural layer invokes code that resides at a deeper layer than the adjacent layer, i.e., the invocation skips at least one layer, and the invoked code is not part of a vertical utility layer that can be referenced from any horizontal layer.

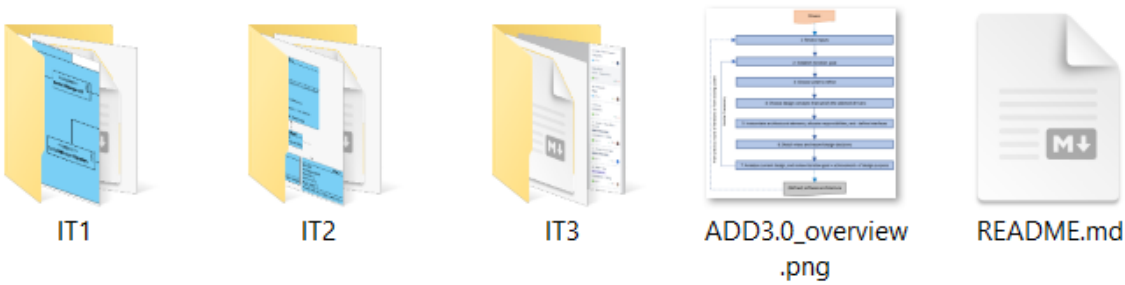
```
@PostMapping("/addSandwich")
public ResponseDTO addSandwich(@RequestBody InputSandwichDTO inputSandwichDTO) {
    return sandwichService.addSandwich(inputSandwichDTO);      1181130, 5 days ago
}
```

```
@Override
public ResponseDTO addSandwich(InputSandwichDTO inputSandwichDTO) {
    try {
        Sandwich sandwich = this.sandwichMapper.toSandwich(inputSandwichDTO);
        Sandwich savedSandwich = this.sandwichRepository.save(sandwich);
        return new ResponseDTO(StatusCode.CREATED.getCode(), this.sandwichMapper.tosandwichDTO(savedSandwich));
    } catch (Exception e) {
        return new ResponseDTO(StatusCode.BAD_REQUEST.getCode(), String.format(format: "%s : %s", e.getClass(), e.getMessage()));
    }
}
```

As can be seen from the images above, no layer has been bypassed. This is an example of how all the code was developed.

CWE-1068: Inconsistency Between Implementation and Documented Design

The implementation of the product is not consistent with the design as described within the relevant documentation.



The entire development process was documented in all three iterations of ADD. Since each iteration is an upgrade of the previous one, there will be no discrepancies between what is implemented and what is developed.