Analyzing, Modeling and Prototyping a Traffic Management System

Federico Pirazzoli

1 Brief initial project overview

In this initial part I will go over the project requirements and a brief overview of what the project encompasses, I will also point out throughout the report where activities made on the SPE end of the project have helped in developing the SAP requirements more in-depth.

1.1 Project Premise and Requirements

The idea of the project is to develop a Traffic Management System, simulating the capability of automatically (or manually) controlling traffic lights at intersections to optimize and better handle the flow of traffic, by identifying the cars in each lane, all of which will be viewed through the use of a front-end, by which users can register, log in, also observe and control (if they have the permits) the traffic lights.

So the basic requirements for the whole application itself are:

- A User can register/log-in to the system;
- A User can monitor and/or operate the way the traffic lights are working, using the front-end UI to receive feedback on their actions;
- The system can identify and show the number of cars in each lane of the intersection;
- The system is capable of using pre-defined or changeable configurations to determine whether to maintain the current traffic flow, or to prioritize certain lanes instead of others to free up congestions;
- The system is also capable of prioritizing lanes, using traffic lights, where Active Emergency Vehicles are present (e.g. on duty Ambulances) in order for them to reach the desired destination faster.

2 Project Analysis

Both the SAP and SPE course emphasized the importance of properly analysing and designing an application, or a software system in general, therefore as, again, both courses provided DDD (Domain Drive Design) as a successful methodology of making up a system, and also because the SPE requirements explicitly demanded a DDD analysis for the project, we used this approach to both do a macro and micro analysis of the overall application, so I will also include requirements obtained from the group analysis done in the SPE part.

2.1 Knowledge Crunching

2.1.1 Event Storming

Initially we decided that it would've been best if each member of the group gathered as much information as they could regarding everything related to ground traffic management (that is, traffic management related to vehicles that travel on the ground, since also air and sea traffic management exist, and obviously, we don't need information related to those), so we could then get together and properly proceed with the analysis of the domain, since unfortunately we couldn't get a hold of a domain expert.

After gathering information we performed a Knowledge Crunching session by Event Storming, and one of the tools we used was Miro, that we've seen during the SAP course. Through the use of Miro we managed to create and gather ideas and knowledge on the topic, especially through the creation of a Mind Map, which contains the key concepts related to the project's environment. As it is with Mind Maps, we created also a hierarchy (as you will see below a horizontal hierarchy) of concepts to better understand the domain.

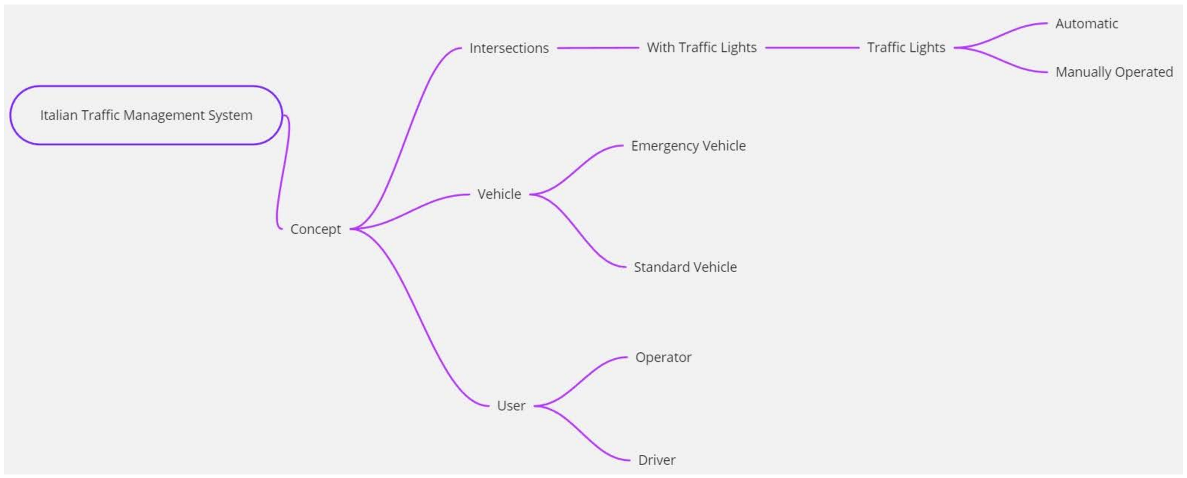


Figure 1: Mind Map representing the Key Concepts of the domain

2.1.2 Ubiquitous Language

After the Mind Map was created and established, we declared an Ubiquitous Language for the domain, that is a vocabulary of names and concepts that will be used throughout the project, which include:

- **Intersections**: Location where roads meet, featuring Traffic Lights for traffic management;
- **Lane/s**: Road segments for vehicle travel;
- **Priority**: Privilege for certain Vehicles due to travel nature;
- **Vehicle/Standard Vehicle**: Autonomous road entity traveling on Lanes;
- **Emergency Vehicle**: Vehicle with priority in lane travel;
- **Vehicles on Lanes (VoL)**: Concept denoting the number of Vehicles on a Lane at a given time;
- **Monitoring**: Users observe Intersection states, including Traffic Lights and Lanes;
- **Manually Operating Intersections (MOI)**: Operators control Traffic Lights at Intersections;
- **Traffic Lights**: Control Intersection flow with Red, Yellow, and Green states;
- **Driver**: User with advanced privileges capable of MOI;
- **Operator**: User with advanced privileges capable of MOI.

2.1.3 Use Cases

Use cases are also a fundamental component in DDD analysis used to understand what the average, or a ”normal”, user would want to interact with the application, what kind of feedback it would want and finally what they would expect by achieving their goal, as it is as a matter of fact, the definition of use cases is a process usually involving an interaction with business users (stakeholders) to understand what users would do with the system which can be conceived as a set of scenarios tied together by a common user goal, therefore a scenario is a sequence of steps describing an interaction between a user and a System.

The Use Cases we thought up are simple but contain critical information as to what the application should entail:

Log into the System
Goal Level: Sea Level
Main Success Scenario:

1. Client reaches the website via search engine or URL
2. Client reaches the homepage of the website
3. Client clicks on the Login button present in the page
4. Client upon reaching the Login page inserts his credentials and tries to login into the System
5. Client reaches Intersections page

Extensions:
4a: The Website responds appropriately to the user based on the System response e.g.: if there’s a problem with the credentials inserted

Register into the System
Goal Level: Sea Level
Main Success Scenario:

1. Client reaches the website via search engine or URL
2. Client reaches the homepage of the website
3. Client clicks on the Register button present in the page
4. Client upon reaching the Register page inserts the requested credentials and tries to register into the System
5. Client reaches Intersections page

Extensions:
4a: The Website responds appropriately to the user based on the System response e.g.: if the credentials inserted for registering are appropriate

Observe Intersection
Goal Level: Sea Level
Main Success Scenario:

1. Client reaches the website via search engine or URL
2. Client Logs into the website
3. Client upon reaching the Intersection page will choose the desired intersection to observe
4. Client access Intersection

Extensions:
2a: We assume the Client already has been registered to the System
4a: In case of errors from the System itself, the website will display appropriate information to the viewing users

Interact with Intersection
Goal Level: Sea Level
Main Success Scenario:

1. Client reaches the website via search engine or URL
2. Client Logs into the website
3. Client upon reaching the Intersection page will choose the desired intersection to observe
4. Client access Intersection
5. Client perform wanted operations on the Intersections based on the buttons present in the interface

Extensions:
2a: We assume the Client already has been registered to the System
4a: In case of errors from the System itself, the website will display appropriate information to the viewing users
5a: The System checks whether the user has the necessary permissions in order to perform these operations

2.1.4 User Stories

2.2 Application Overview

Thanks to the creation of the Use Cases just mentioned, and also through the creation of two context maps I managed to have, at first, a "high level" view of what the application is supposed to be, then after refining the initial map with the implementation of building blocks, I had a better understanding of what the application should be on a hypothetical level. This helped me understand what kind of operations would be necessary to implement for each of the possible services I was going to implement.

Here below is the high level context map, with general details on what the application should contain:

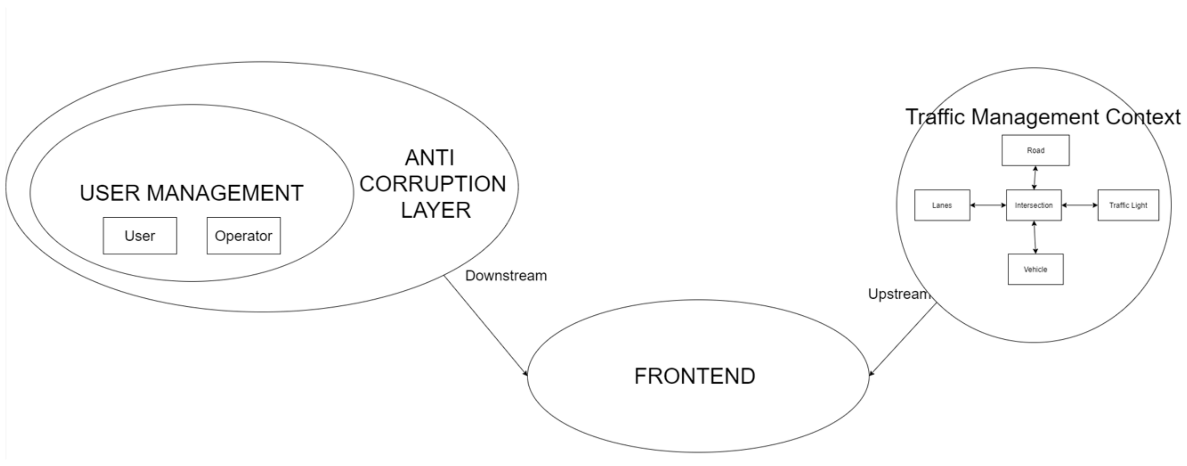


Figure 2: High Level Context Map

Meanwhile, this one below represents something more akin to what the implementation of each entity should be:

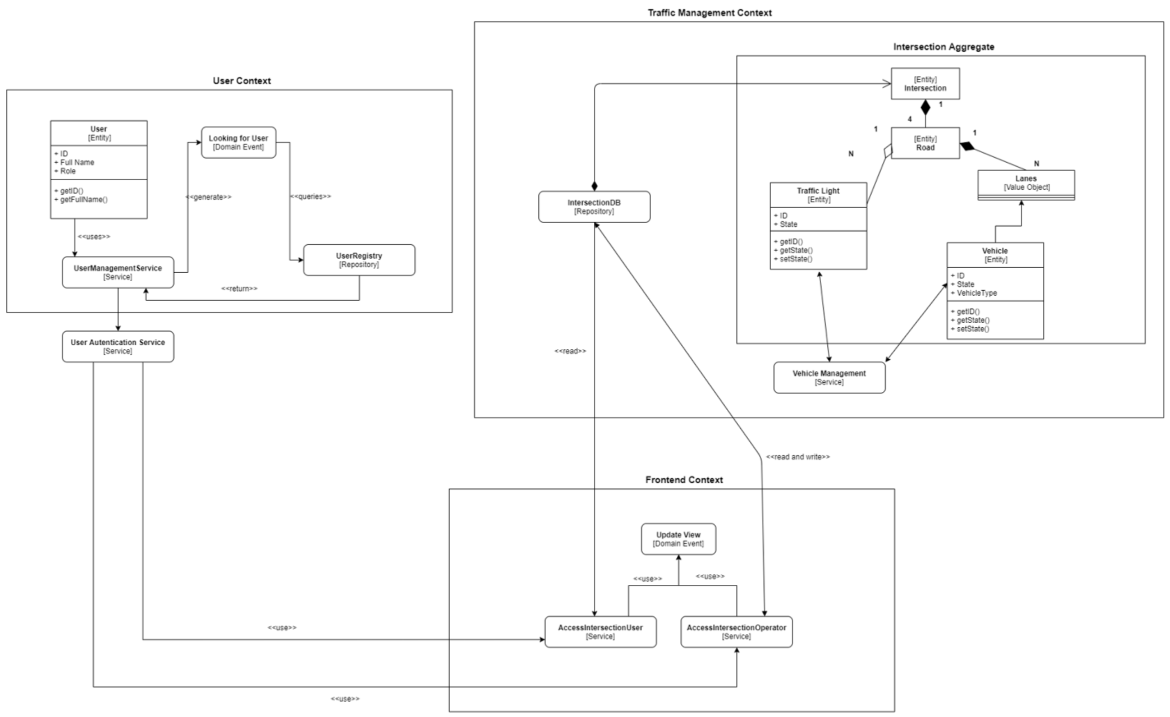


Figure 3: More in-depth Context Map with Building Blocks

From both of the maps, we can identify 3 main entities that will make up the application:

- **User Context**: Which is responsible for storing and maintain all information related to users who have registered and need to log-in into the system;
- **Front-end Context**: Which will compromise of the interface users will see in order to interact and observe the main part of the application (the traffic management);
- **Traffic Management Context**: Finally the last entity is responsible for encapsulating the main part of the application: the traffic management responsibility and simulation.

Once the 3 entities were cemented, the next step was to think about what kind of architecture and technologies would best fulfill the role of the project itself, considering obviously what we had learned throughout the SAP course.

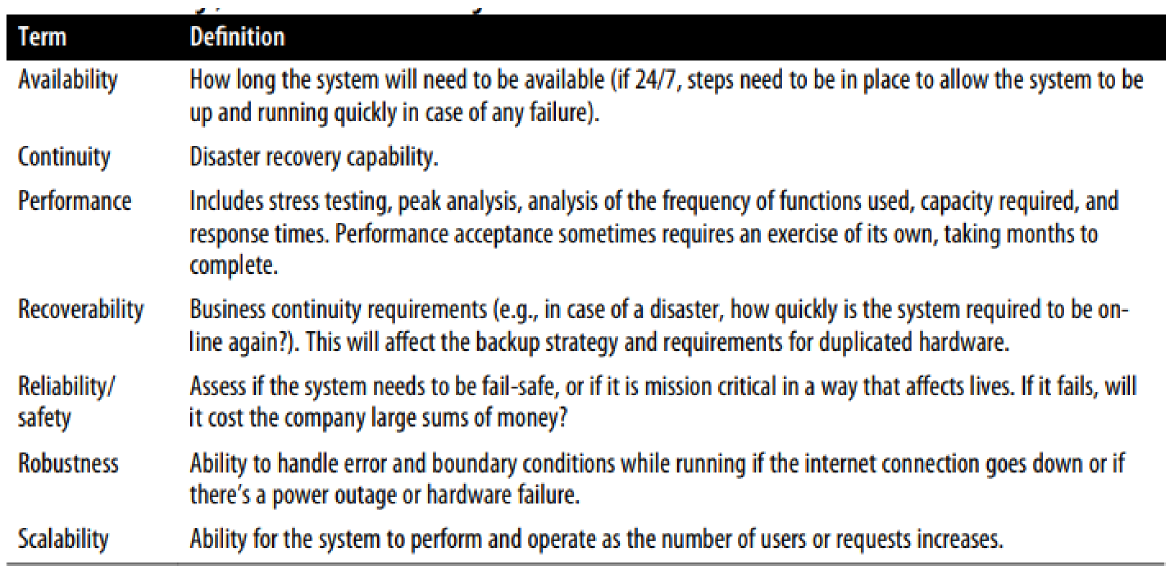
2.3 Quality Attributes

Another important factor to consider are the Quality Attributes (QAs) found to be the most needed, and successfully achieved/implemented, in order for the application to work properly. As a brief reminder, with QA we mean a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders beyond the basic function of the system, and as a matter of fact the properties are divided between **Functional Properties (FP)**, which are properties needed (and necessary) for the application to perform the work it was created for, so basically what was discovered during the Requirements Analysis; and also **Non-Functional Properties (NFP)**, which is instead a constraint on the manner in which the system implements and delivers its functionality. The properties that will be discussed here are connected to the practices, architectures and technologies used in the application, which will be discussed in the next chapter, therefore I will mention them here in order to explain why some QAs are fulfilled, but I will go into them more specifically later on.

As mentioned, this is not a definitive way to separate and identify Quality Attributes, as a matter of fact there are also different ways to categorize them, but I will be referring to these three in order to better explain what attributes are needed and satisfied by the application.

Additionally, Quality Attributes could be categorized (as it is not set in stone this separation) based on what aspect their related to, as shown below:

2.3.1 Operational Architecture Characteristics

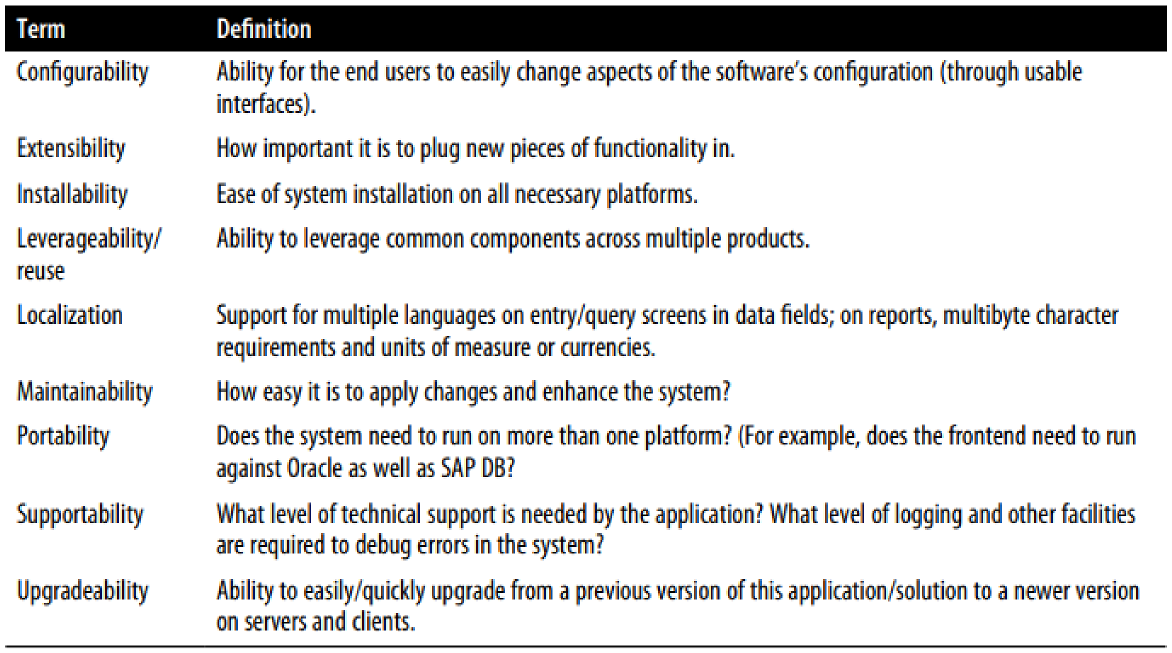


Term	Definition
Availability	How long the system will need to be available (if 24/7, steps need to be in place to allow the system to be up and running quickly in case of any failure).
Continuity	Disaster recovery capability.
Performance	Includes stress testing, peak analysis, analysis of the frequency of functions used, capacity required, and response times. Performance acceptance sometimes requires an exercise of its own, taking months to complete.
Recoverability	Business continuity requirements (e.g., in case of a disaster, how quickly is the system required to be on-line again?). This will affect the backup strategy and requirements for duplicated hardware.
Reliability/ safety	Assess if the system needs to be fail-safe, or if it is mission critical in a way that affects lives. If it fails, will it cost the company large sums of money?
Robustness	Ability to handle error and boundary conditions while running if the internet connection goes down or if there's a power outage or hardware failure.
Scalability	Ability for the system to perform and operate as the number of users or requests increases.

Figure 4: Operational Architecture Characteristics

The Operational Architecture Characteristics heavily overlap with operations and DevOps concerns, and because of the various implementations done via the SPE requirements, a lot of these are already satisfied by the implementation of a Continuous Integration, Development and Deployment pipeline. Most of the qualities written here are monitored through the use of Kubernetes, which is the tool we used to deploy automatically by SSHing into an AWS EC2 instance (basically a cloud virtual machine), triggered by a Github Action (a script that gets executed once pushes get made to certain branches). With Kubernetes, not only we can monitor whether the system is doing well or not, but we can also, through the use of pods, deploy even more services if needed on the fly, all the while resources get balanced automatically between the services by Kubernetes. In terms of the most critical ones though, definitely what stands on top is Robustness and Reliability, as the system (in a real world scenario) is meant to handle traffic intersections which obviously is a very sensitive and dangerous environment, where human lives could be put at a great risk.

2.3.2 Structural Architecture Characteristics

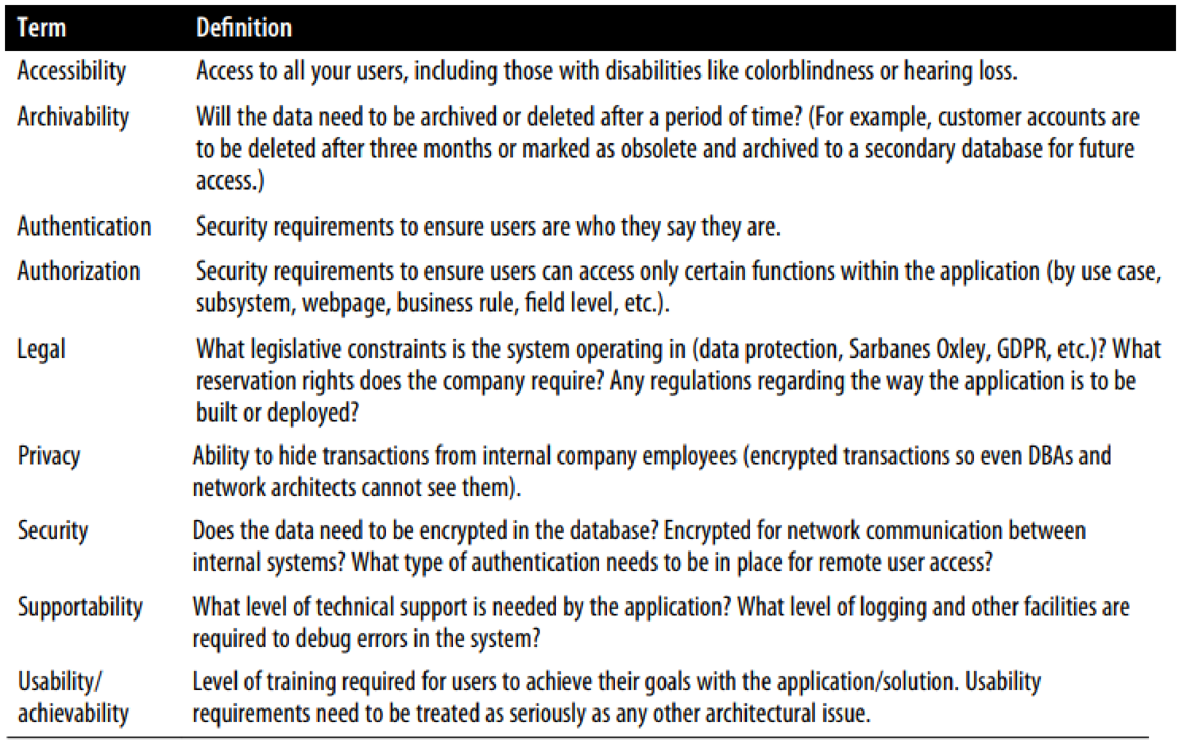


Term	Definition
Configurability	Ability for the end users to easily change aspects of the software's configuration (through usable interfaces).
Extensibility	How important it is to plug new pieces of functionality in.
Installability	Ease of system installation on all necessary platforms.
Leverageability/ reuse	Ability to leverage common components across multiple products.
Localization	Support for multiple languages on entry/query screens in data fields; on reports, multibyte character requirements and units of measure or currencies.
Maintainability	How easy it is to apply changes and enhance the system?
Portability	Does the system need to run on more than one platform? (For example, does the frontend need to run against Oracle as well as SAP DB?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Upgradeability	Ability to easily/quickly upgrade from a previous version of this application/solution to a newer version on servers and clients.

Figure 5: Structural Architecture Characteristics

The Structural Architecture Characteristics instead have attributes like *Configurability*, *Extensibility*, *Installability*, *Leverageability* and *Maintainability* which are all very easy to conform to when it comes to the services not related to the Traffic Management System, since they're basically all microservices, running REST APIs, created through modular frameworks (like Spring Boot for example), unfortunately when it comes to the core functionality, that is of the traffic management part through agents, because we're using JaCaMo, we're constrained to the functionalities that the library offers, as it also operates differently when it comes to setting it up from traditional software, therefore (hypothetically in a real scenario) it might need specialized individuals that will work on the open source library to make it more extendable and easy to use.
Meanwhile attributes like *Portability*, *Supportability* and *Upgradeability* are all simplified and easy to perform by the use of DevOps practices through the SPE requirements.

2.3.3 Cross-Cutting Architecture Characteristics



Term	Definition
Accessibility	Access to all your users, including those with disabilities like colorblindness or hearing loss.
Archivability	Will the data need to be archived or deleted after a period of time? (For example, customer accounts are to be deleted after three months or marked as obsolete and archived to a secondary database for future access.)
Authentication	Security requirements to ensure users are who they say they are.
Authorization	Security requirements to ensure users can access only certain functions within the application (by use case, subsystem, webpage, business rule, field level, etc.).
Legal	What legislative constraints is the system operating in (data protection, Sarbanes Oxley, GDPR, etc.)? What reservation rights does the company require? Any regulations regarding the way the application is to be built or deployed?
Privacy	Ability to hide transactions from internal company employees (encrypted transactions so even DBAs and network architects cannot see them).
Security	Does the data need to be encrypted in the database? Encrypted for network communication between internal systems? What type of authentication needs to be in place for remote user access?
Supportability	What level of technical support is needed by the application? What level of logging and other facilities are required to debug errors in the system?
Usability/ achievability	Level of training required for users to achieve their goals with the application/solution. Usability requirements need to be treated as seriously as any other architectural issue.

Figure 6: Cross-Cutting Architecture Characteristics

Finally when it comes to the Cross-Cutting Architecture Characteristics, most of the focus should be on attributes like *Authentication*, *Authorization*, *Legal*, *Privacy* and *Security*. In the project, Authentication and Authorization were made very simple, as a mock-up really, in order to just concentrate on the core functionalities, but in a real world scenario, general security of all of the systems should be a top priority, because of the fact that users with authorization can change the flow of traffic, therefore only certain individuals need completely protected access to those areas. The Privacy aspect comes from the fact that hypothetical sensors put at intersections in order to make the system operational, need to gather only the necessary data from their surroundings, which are the cars situated in each lane.

3 Architectures and Technologies

I will go over the architecture chosen for each entity first, then discuss what technologies have been chosen, why and their implementation.

3.1 Entities Architecture

In regards to what the project actually aims for, that is satisfy the requirements for the SAP (and SPE) exam/s, for each of the three entities I decided to follow a Hexagonal Architecture Style, by means of the Ports and Adapters approach.

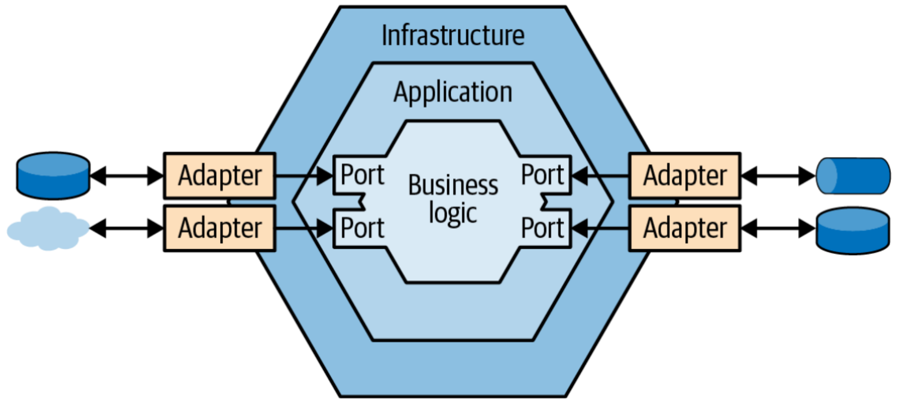


Figure 7: Classic example of Hexagonal Architecture

I will not go more in-depth in this subsection what the actual implementation is for each entity, as that will be discussed later on when technologies chosen are explained.

The main idea is still to take advantage of the capabilities of the Hexagonal Architecture, that lets us decouple the system's business logic from its infrastructural components, so that instead of referencing and calling the infrastructural components directly, the business logic layer defines "ports" that have to be implemented bu the infrastructure layer, by the which then "adapters" are also implemented, which are concrete implementation of the ports' interfaces for working with different technologies. In fact, each of the entities, aside from the front-end, sports a REST API, by which the front-end instead garners and checks information with each of the service: the front-end will use the User Context API to check operations regarding log-in and registering, meanwhile it will use the Traffic Management Context ones to acquire information regarding the state of intersections and change the behaviour of the traffic lights through them.

In particular, the last entity, the Traffic Management Context, offers the REST endpoints to access the service, and simultaneously performs and emulates the operations of an intersection by using the MAP, Multi-Agent Paradigm/Programming, by using agents to represents entities inside the intersection itself¡ as a matter of fact, each traffic light is modeled as an agent, which are capable of coordinating their operations, which is the changing of their color, and therefore, which car on which lane is permitted to move, all the while the intersection itself is the environment in which the traffic lights are operating in, by using a specific technology and language.

3.2 Technologies, Implementation and Characteristics

All of the components which I'm gonna describe for each entity has been containerized, in order to make them portable and, on the SPE part, to implement a CI (Continuous Integration) and CD (which stand for Continuous Development and Deployment) pipeline.

3.2.1 User Context

The main component of the User Context entity was developed using the Spring Boot framework, that is a Java framework, which lets users create various micro-services based on their needs, since the framework itself is made up of several "plugins" which can offer different functionalities, all of which can be easily set up and accessed by using the most common build tools, like Gradle and Maven.

In my case I used the Spring Boot Web Framework, which let me build a web application which exposed a RESTfull API externally to let other services or application access and modify information related to users, which is done again natively by Spring Boot since it also offers, internally, the possibility to connect to external DBs to store the data, in my case I used a H2 DB, which is a Java Relational DB, which is also been containerized. Therefore the application exposes REST HTTP endpoints, and based on the operation performed, the Spring Boot application would communicate with an external H2 DB which is connected to.

To be a bit more precise, the user entity itself that gets saved onto the DB contains various fields, just to mimic in a simple way a possible implementation of a user, the fields that it contains are:

- **ID**, a simple Long number which identifies the user, and that also Spring Boot adds automatically and increases when creating a new user, with its counterpart on the DB side being an auto-increment Primary Key basically;
- **Name**, a simple field containing the users name;
- **Username**, again a field containing the users surname;
- **Password**, a field containing the users password, which gets hashed by Spring Boot;
- **UserRole**, and finally an enum representing the role of the user itself, which can be either Driver or Operator (operations via API that interact with the UserRole can be expressed either through text, e.g. writing the word "Driver", or using numbers like a normal enum, e.g. writing the number 0 to identify the Driver).

The application offers the following endpoints to let changes be made on the DB, by either adding, deleting or modify users, all done and expressed RESTfully of course as the implementation requires to:

- **GET**:
  - ***/users***, which lets services get information regarding existing users, conforming with the REST implementation;
  - ***/users/{id}***, which lets services access information regarding a specific user based on the ID given;
  - ***/users/check***, which will let the service know if the user credentials that have been provided are when performing the RPC are correct (and therefore the user can log-in), which returns a boolean;
- **POST**:
  - ***/users***, which lets services create and add new users to the DB by providing the necessary information (name, surname, role etc.);
- **PUT**:
  - ***/users/{id}***, which lets services modify the information related to a certain user, identifying them by their id, with the data provided in the RPC call;
- **DELETE**:
  - ***/users/{id}***, which lets services delete a user from a DB by providing the right id which identifies them.

All of the endpoints will return a code based on the fact that the operation has gone well, or has failed, alongside other information they might return.

3.2.2 Front-end Context

The Front-end Context entity is made up of a simple Vue application, by the which users can connect, register or log-in into the system and then monitor and execute operations related to intersections.
As said previously, this is done by performing RPCs on the REST endpoints of both the User Context and Traffic Management Context entities, by including inside the container containing the Vue application, also a Nginx reverse proxy, which re-directs the calls made by the browser (Vue application), to the appropriate endpoint based on the fact that the application is in a testing environment or is being deployed, and it also hosts the built version of the Vue application.

3.2.3 Traffic Management Context

Finally the last entity is made up of the core of the application, which in this project is being emulated, through the use multi agent programming, and this paradigm is implemented through the use of a Java library called JaCaMo.

JaCaMo stands out as an agent programming library since it combines 3 other components which are fundamental in BDI programming (Behaviour-Desire-Intention), which is a way program the actions of agents.

The componets of JaCaMo are:

- **Ja** as **Jason**: which is an interpreter for an extended version of AgentSpeak, which is an agent-oriented programming language, and it's based on logic programming and the BDI software model architecture for (cognitive) autonomous agents. It implements the operational semantics of that language, and provides a platform for the development of multi-agent systems, with many user-customisable features.
- **Ca** as **Cartago**: which is a Java-based Framework for Programming Environments in Agent-oriented Applications, therefore a "space" in which environments can act with and with each other.
- **Mo** as **Moise**: which is an organisational platform for Multi-Agent Systems (MAS) based on notions like roles, groups, and missions. It enables a MAS to have an explicit specification of its organisation. This specification is to be used both by the agents to reason about their organisation and by an organisation platform that enforces that the agents follow the specification.

When it comes to the project, JaCaMo is used to build traffic lights as agents, through the use of the AgentSpeak language, which interact in an environment, which is the intersection. Through the use of AgentSpeak and having the intersection as an environment entity built with Cartago, the traffic lights can coordinate with each other in making sure their color, or state, will change only on specific occasions, for example, traffic lights will change colors as pairs (so if there are 4, as two pairs), in order to make sure the only cars which will be permitted to move are the ones whose lanes are facing each other, while the other traffic lights will wait for the other pair to turn red.

The agents in our project additionally, are created as "artifacts" which are entities that can interact "externally" with each other in a environment and are also identifiable, also the intersection itself is an artifact; this also lets perform operations from outside the JaCaMo application itself, through the use of another library called JaCaMo REST, that exposes REST endpoints for other services to use, and that I used in my application. The operations that are exposed by the artifacts, are the ones that let user, that have the appropriate permissions, change the state of the traffic lights. Finally the use of artifacts also permitted me to set changeable configurations of the waiting times of traffic lights through the use of .json files.