ECUGuardian - Software Architecture

Infor**MAT** - Mattia **Piazzalunga** & Matteo **Severgnini**

Table of contents

Introduction & Theory

The theory cited in this project for explanatory purposes is extracted from lectures on "Software Architecture", given by Prof. Daniela Micucci, of the Master's Degree in Computer Science at Bicocca University in Milan, Italy.

https://elearning.unimib.it/course/info.php?id=51434

Theory was used to introduce the project and its context, as well as to highlight the importance of this type of architecture documentation.

We provide the link to the project's github repository and to the file (.vpp).

What is software architecture? - a bit of theory

We start talking **SW architecture** in the early 1990s, with the emergence of the need to provide **solid structure to the system before actual development**. It is the starting point of any software development.

According to an IEEE definition, software architecture is *the fundamental organization of a system, embodied in its components (computational components), their relationships to each other and to the environment, and the principles that govern its design and evolution*. It is, therefore, a **higher level** of design than concrete design: we do not define how an element is represented, but we highlight its existence, under certain constraints. In general, therefore, there is no 1:1 mapping with representation, but certain observations about elements that are not useful for reasoning about the system are omitted. **There is,** however, **no perfect architecture; it depends strongly on the type of system**.

NOTE. the analysis shows us what the system is supposed to do, without the how. Opposed to concrete design, which, based on diagrams, highlights to me the final implementation.

Architecture, in general, **matches elements with functional requirements, ensuring nonfunctional requirements**.

Structures - a bit of theory

Structures are the fundamental part of SW architecture. A structure is a **set of elements held together by a relationship**. It becomes **architectural** if it supports reasoning about the system and its properties.

We can, therefore, redefine the concept of architecture as a set of structures, each of which is a set of elements.

In the complexity of modern systems, we make use of **three types of structures based on the aspect to be modeled** (they capture different aspects of the system, I use them to verify aspects of different qualities):

- **Modules**
  We partition the system into implementation units called modules, which are static and focus on how the functionality of the system is grouped.
  We have software elements (classes, layers, feature groups) linked by relationships.
  NOTE. We highlight, at this level, simply which modules interact with each other.
  Module views are commonly mapped to component and connector views.
- **Components and connectors**
  Module instances are the components. Components and connectors give us insight into the dynamic nature of the system: how are constraints and nonfunctional requirements met at run-time? This framework focuses on how components services, peers, clients, servers, filters) interact via connectors (call-returns, process synchronization, pipes, protocols, API rests, ...)
- **Allocation structures**
  They describe the mapping between software structures and environnements (between software structures and non-software structures such as CPUs, file systems, networks, development teams). They can be used for both modules and components.
  They can be:
  - **Organizational**
    To which team assign the development of a module.
  - **Development**
    How to split a module into files or where to save them (github for example)
  - **Deployment**
    Which component to deploy and on which machine (server, etc.).
  - …

NOTE. we can see that, in structures, we always have elements and connections between them;

Structures, however, are related, representing as we said, the same system from different points of view.

In general, in any case, structures are "an idea." The "on paper" representation of a structure is called a **view**. A common format for representing the view is the UML model.

NOTE.

- Anything that does not have a constraint we do not treat at the architectural level!!! In architecture we do not treat superfluous elements, we leave them out (we have abstraction). Saying, for example, how classes communicate within a module is the task of concrete design.
- Documentation, especially architectural, is, in general, important to allow we to get wer hands on a system.

Functionality - a bit of theory

Functionality/functional requirements are the basis of a system. However, systems are often changed because nonfunctional requirements (quality requirements) are not met.

In general, however, we have:

- **Functional requirements.** These requirements highlight what the system should do and how it should react to stimuli.
- **Quality requirements.** These requirements express quality requirements on functional aspects (e.g., how fast an operation should perform or how error-resistant it is) or on the system as a whole (e.g., development time).
- **Constraints.** These are choices imposed by the architect (e.g., use Ruby because all programmers use it) or choices dictated by the environment (e.g., the company makes only web apps, the company uses only certain frameworks, ...).

Quality Attribute - a bit of theory

A software architecture, if chosen correctly, must satisfy **quality attributes**. A quality attribute is a **measurable or verifiable property of a system** that is used to indicate **the degree to which it meets the needs of stakeholders**. It is important to point out at the outset that quality attributes often **depend on the environment and the constraints it dictates**.

In general, however, **it is not possible to guarantee them all in a system**; one must understand which aspects are predominant and select architectural solutions that are promoters of these requirements. The **most important set of requirements is the ASR** (Architectural Significant Requirements).

In each case, as we mentioned earlier, **these qualities must be measurable**. Moreover, we must be able to associate only one dimension of quality with certain problems and without having ambiguity in the choice.

Documenting an architecture - a bit of theory

Creating an architecture is not enough: **it must be communicated in a way that enables stakeholders to use it properly to do their work**. It is essential, then, to understand how stakeholders use architecture documentation: the uses determine the information to be captured.

- **Education**
  By introduction to the system to new team members, analysts, external evaluators, or new architects;
- **Primary vehicle for communication among stakeholders**.
  Can be especially useful in supporting communication between architect to developers or between architects and prospective architects.
- **Basis for system analysis and construction.**
  Architecture tells implementers what to implement. Each module defines the interfaces provided to other modules and requests to other modules.

It is, also, a **foundation document to the evolution of the architecture**.

Views are the most important concept associated with software architecture documentation. A software architecture is a complex entity that cannot be described in a simple, one-dimensional way. A view is a representation of a set of system elements and the relationships

between them (i.e., structures). Views, then and as we know, **allow us to divide a software architecture into different interesting and manageable representations of the system**.

Different views support different goals and uses. Therefore, the views to be documented depend, again, on the intended use of the documentation.

In addition, an architecture may require **documentation of behavior that describes how elements interact with each other**.

Our task is to document: relevant views and information that applies to more than one view. Therefore, the **documentation package consists of**:

- **Views;**
- **Documentation beyond views**.

When we study an architecture diagram, we see only the end product of a thought process. **Recording the design decisions** beyond the elements, relationships, and properties we choose is critical to helping we understand how we arrived at the result, and properties is critical to understanding how we arrived at the result.

How to identify a viable architectural solution? - The strategy of this project

One technique (among many) for **properly** designing an architectural solution is to **design the architecture based on views**. The steps to be identified next are (in order):

1. **Problem Architecture**
   At this stage, one tries to understand the problem, checking the understanding of functional requirements and identifying non-functional requirements. In general, one focuses on:
   - What does the system need to do?
   - Who are the actors?
   - What are the data? And what transformations do they undergo so that the data of interest to the system can be realized?
2. **Logic Architecture**
   In this step we go on to decompose the functionality of the system into logical modules, i.e., indivisible blocks (separations of concern). In this step the architecture must guarantee the identified ASRs. In general, we focus on 'component identification: how to "divide" the system in the best possible way with respect to the various functionalities/functional requirements.
3. **Concrete Architecture**.
   In this step we are going to look at how the logical components interact (what protocols etc.) and what technological solutions allow us to realize the logical components;
4. **Deployment Architecture.**
   In this point we focus on the structure components and connectors deployed in the environnement (software components that need to be installed on the hardware).
5. **Implementation Architecture** (*not addressed in this project*).
   In this step, technologies, platforms, programming languages are chosen as far as possible, …

Important, in any case, is to consider from the outset the **constraints imposed on us**.

To represent the system, in any case, we will make use, as already pointed out, **of views represented through the UML model**.

Project track

We proceed by reporting the outline of the project...

“

A company is engaged in the design and installation of **power distribution substations**. The substations are **scattered throughout the territory** and are equipped with **sensors** for instantaneous measurement of the power delivered. An operational management system is to be designed that:

- **Acquires "real-time" data from the individual power station**s. The data detectable by the sensors is the instantaneous power delivered (in kw);
- **Checks for anomaly situations** (anomaly power peaks compared to the limits set for the control unit);
- In the **case of anomaly situations**, turns off the control unit and notifies the central technical service of the anomaly;
- Supports the decisions of the central technical service to **identify the most suitable operator** (by availability, geographic proximity, technical expertise related to the type of control unit) to repair the fault;
- **Notifies the operator of the action to be taken**;
- Allows the operator to notify the technical service to **initiate the intervention and its completion**;
- **Collects** instantaneous power consumption **data** in order to **define new power unit distribution policies**.

“

Initial study

Acronyms used in the report

We provide a list of **acronyms & synonymous** used in this report to have, at each point in this report, clear references:

1. SA, Software Architecture;
2. kw/h, kilowatt-hour;
3. ECU, Electrical Control Unit, Control Unit.
4. Operator, Technical.
5. CTS, Central Technical Service.

Assumptions

Since we were given little information from the outline, we proceed by making **assumptions** (*in reality to be verified with stakeholders*):

- We assume to work, in terms of current regulations, in **Italy and to Italian contractual situations**;
- Assume that the **control unit is viewed as a "single block" with its sensors**. Such a control unit has a certain level of **software "intelligence",** such as:
    - **Converting analog sensor signals to digital and sending them**;
    - **Accepting power on and off commands**.
- Suppose we operate in the **Borgo Palazzo neighborhood area** of the municipality of **Bergamo**, where there are, according to data from the Regulatory Authority for Energy Networks and Environment (**ARERA**), as of December 31, 2022, **10.800 housing units** with active electric meters.

Again according to ARERA, that district refers to only **one power distribution plant**, assumption we will make for this project.

ARERA
Autorità di Regolazione per Energia Reti e Ambiente

https://shorturl.at/kwxI4

- The power packs referred to in the trace we assume refer to **domestic power packs**, thus meters in people's homes, whose **maximum peak power is 3kw/h;**
- By **anomaly situations we refer to**:
    - **Power peaks compared to the limits provided for the control unit**. As limits we use those provided by **e-distribuzione**, the largest company in Italy in the electricity Distribution and Metering sector serving more than 31.5 million Customers connected to the grid. A limit is referred to as **10% higher than the committed power subscribed at the contract level**. For example, for a 3 kw/h contract, it is possible to draw up to 3.3 kW without time limit. In addition, **if we exceed 3.3 kw/h, we are given the option to draw up to 4 kw/h for three**

hours; at three hours, a warning is triggered. If **more than 4 kw/h is drawn, the power supply control device is triggered at two minutes**.

shorturl.at/rHOT5.

- **Overheating**.
  Usually, **the operating temperature of an ECU is between [-25 C°; +55 C°]** degrees for e-distribution ECUs. For this reason, a temperature beyond the range triggers, after the classic three checks, the signaling (https://shorturl.at/cpt05).

Important to point out that **an anomaly will only be reported for temperature-related problems or kilowatt consumption greater than 4kw/h**. Consumption between 3kw/h+10% and 4kw/h will not be considered as a problem to be reported to the "central tenical service," but as normal above-average consumption (We will go, at most, to disable the ECU).

e-distribuzione

- We **estimate,** at this point, **the number of anomalies**. According to **ARERA**, in Italy we have 40,000,000 ECUs, 32,000,000 electric (80%) and 8,000,000 mechanical (20%). In 2022, 1,500,000 failures were recorded, of which 1,200,000 were to electrical and 300,000 to mechanical. So a failure rate of 3% for electric ECUs and 3.75% for mechanical ECUs.

  https://shorturl.at/kwxI4

  For electric ECUs only 60% of failures were due to internal (and not accidental) problems. Since in our reality we only consider this type of failure we have 0.03x0.6=0.018=1.8%. For electric ECUs the percentage of integer failures rises to 70%, so 0.0375x0.7=0.026=2.6%.
  In total, then, we have the 0.8x0.018+0.2*0.026=0.02=**2% failures**.
  Considering that we operate with 10,800 ECUs in our reality**, we can estimate that 216 ECUs report faults annually and, by extension, 18 ECUs monthly**.
- The system running on the control unit, every 60 seconds (**3 "blocks" of data generated**), **averages the latest data** (**trying to avoid "spurious" data**) and checks if an anomaly is present.
- The **operator is identified geographically by latitude and longitude** (location) provided by a **GPS instrument**, this is because he may not even be in a building and, therefore, does not have a reference address available;
- Working in Italy and having **associated one ECU per housing unit**, we are going to associate in a 1:1 relationship an ECU, which in any case will be identified with an id internal to the system, **with an address**;
- Every 30 days the study of the collected data takes place for the definition of new policies, where by **policy we mean the insertion, shifting or upgrading of the distribution unit**;
- The **devices provided to technicians and CTS** are supplied with a "**smart component**" for **managing notifications** (our system sends notifications and the "component" queues and manages them);
- The **technician is the one who manually reactivates the ECU** during the intervention (if the fault is resolved), not the system.
- We **estimate**, based on assumptions, the **presence of 4 deployed technicians and 3 operators in the CTS.**

... other, less important assumptions are set out at the same time as the explanatory views.

Problem architecture

Introduction - a bit of theory

The problem architecture must **define**:

- The **functionality that the system must offer to external actors** (functional requirements)
  Diagram: UML use case diagram.
- **The types of information/data that the system needs to handle for the realization of functionality**. By now all systems work on the data where I have to express constraints, …
  Diagram: UML class diagram.
- The information flows that the system must support to achieve the functionality defined in step 1 by operating on the data defined in step 2 (in/out). **How information evolves within my system**.
  Diagram: UML activity diagram.
- Represent the control flows that the system must support to ensure functionality. They specify why certain things happen, why a component is activated, …
  Diagramma: UML activity diagram.

NOTE.

- the same diagram with a certain syntax can have multiple semantics *(e.g., we can use the class diagram to model both data and concrete components)*;
- we have strongly data-driven systems that are therefore to be dealt with at the architectural level.

Questions are attempted to be answered to capture the basic elements of the problem:

- **Static aspects**. It's those questions that allow me to identify, then, the forms.
  - **Who**. Who is part of the system? Who are the actors?
  - **What**. What are the data? What entities make up the system?
  - **Where**. Where are the data and the who of the system? Is the user sitting at the terminal or iterating with a tablet?
- **Dynamic aspects**
  - **How**. How do the transformations take place? How is information manipulated so that the system does what is required?
  - **Why**. Why do certain things happen? Why are activities activated? What are the mechanisms dietri the innate?
  - **When**. When do things need to be done? (delay, timing, ...) How often?

Use case diagram - "WHO" specification + “WHERE” for the actors

link here

Actor is everything I take for granted that already exists and that in no way I have to implement. Actors are not only human beings, but also **pieces of software**: if we exploit the services or command it we do. Actors are **all external, as-is entities**: we exploit it.

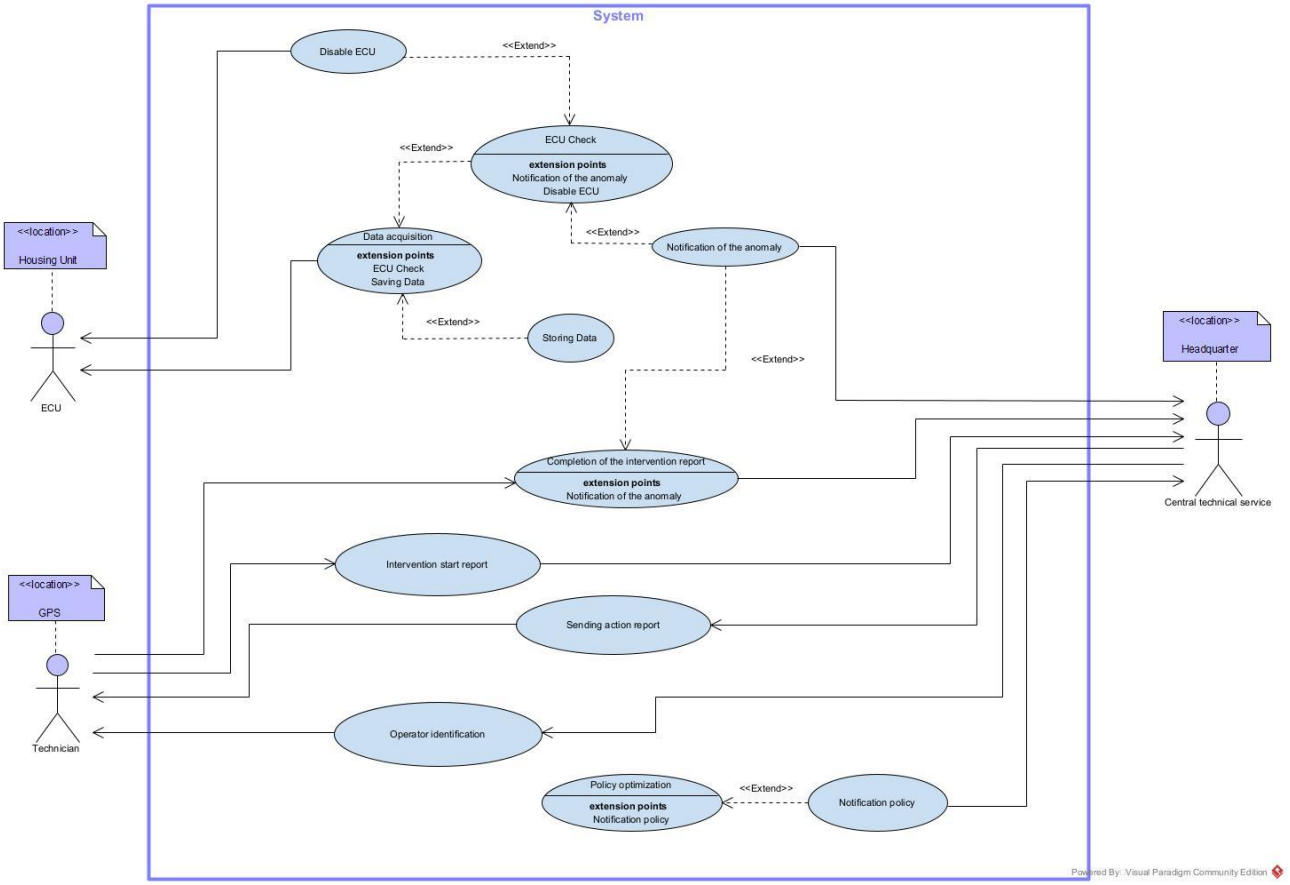
NOTE. there is no such thing as the system actor! The system is the whole piece of software, it is the set of use cases.

It is, moreover, **important for actors to specify where**: a person moving or not influencing the system differently.

Another important aspect to consider is whether the actor is **internal or external to the system** itself: if the actor is not internal to the system we need to figure out what communication interfaces to have with it.

NOTE. if the actor is internal to the system I need a datum that models it to me semantically.

Our solution.



Let us specify of the **rationale** to allow **full understanding of the choices less intuitable from the diagram.**

Element	Responasability
Data acquisition	The system, running on the ECU, acquires the data block from the ECU itself every 20 seconds.
Storing Data	The system, running on the ECU, takes the prepared data from "data acquisition" and saves it in the central system. This process occurs every 60 seconds allowing 3 blocks of data to be acquired.
ECU check	The part of the system running on the ECU verifies and interprets the signals provided by the "intelligent component" of the ECU and, in case of an anomaly, has the ability to disable it (Disable ECU) and, if necessary, notify the "Central Technical Service" of the anomaly (Notificatio of the anomaly).

	It is essential that this component be deployed on the ECU to disable the ECU at the same time as the fault, even if it cannot be called remotely.
Operator identification	The system, when invoked by the "Central Technical Service," allows the identification of an operator using certain features (technical knowledge + geographical area of interest). The real-time geographic location is requested from the intelligent component in use by the technician using GPS.
Intervention start report	The system allows the operator to notify the CTS to signal the start of the intervention.
Sending action report	The system allows an operator to be notified to initiate an intervention.
Completion of the intervention report	The system allows the operator to notify the CTS to signal the end of the intervention. If the anomaly has not been resolved or has been partially resolved, the system allows the generation of a new anomaly
Policy optimization	The system, every 30 days, consults the persistent history to create statistics and outline new ECU distribution policies, such as adding or removing an ECU. If a new policy is feasible, the system notifies (Notification policy) the "Central Technical Service."

In addition, we highlight that the **actors are three**:

- **ECU** associated with a house;
- **Technician** on the move and located by GPS running on the device provided;
- The **Central Technical** Service to which several operators belong.

Data diagram - "WHAT" specification + “WHERE” for the data

link here

To help us understand what the data model is, let us differentiate it from the domain model. The domain model tries to capture with entities what we are talking about. It is a broader concept that represents an in-depth understanding of a particular domain or area of activity. It focuses on understanding the entities, concepts, rules and relationships that characterize the domain in question.

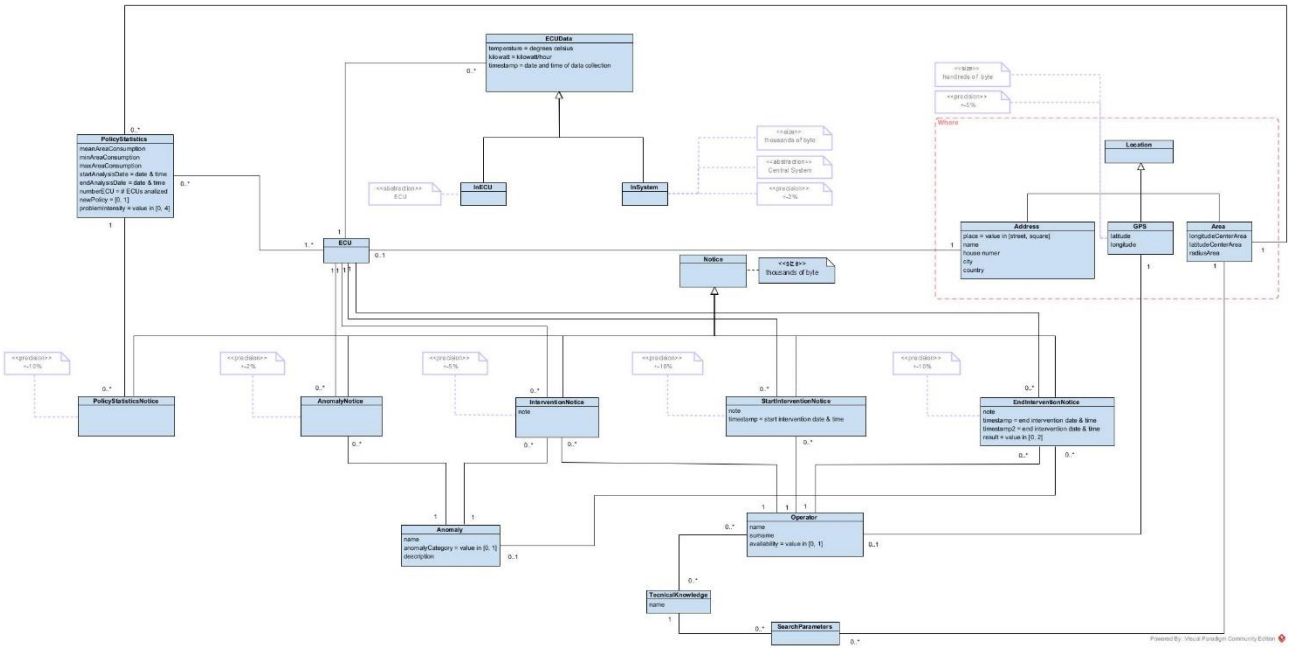
When we make a **data model**, however, we are simply dealing with information: the **data**. The data model is a specific part of the domain model and focuses primarily on **the structure of the data used in a system** or application. I will only have the transit data; everything else I take out. Here, also, I have **attributes that have no visibility or type.**

Some warnings:

- attributes have neither quality nor type;
- **relationships between positions** can also be modeled;

- **An actor can be brought into the system by modeling it as a datum,** with a representation that is necessary for our purposes. *For example, we are interested in knowing the origin of data we can associate with a datum that is the actor itself.*
- To show that a datum is different depending on when it is processed (multiple evolutions) we use **generalization**;
- We can **add units of measurement**;
- We can **add labels to better characterize the data**. *For example, specifying the size in terms of bytes of the data allows us to think later about how much bandwidth is needed for the transfer. It is also important, for example, to specify precision to understand how much delay or loss of information we can tolerate: changing precision often forces us to change the type of system;*
- We can also specify **abstraction by label**: from what system aspect do we see the object to "call" it a certain way? What level of abstraction are we at (semantics changes)? This information is useful, then, when we do partitioning of the system into modules.

Our solution.



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
Location	Location and its associated children shape the locations of our reality: • The area, defined by center and radius, is associated with the search operators (SearchParameters) and statistics to define a new policy (PolicyStatistics), so as to delineate the portion of the territory to be considered; • A single GPS location is associated with at most one operator to locate it (not afferent at a given time to a home, but to a long in 2D space); • Address locates, on the other hand, a control unit.

	We are talking about data that weighs, in transmission, a thousand bits on which we must have a fair amount of precision, especially if we are talking about the GPS position.
ECU	The ECU is an actor modelled as a datum, which is essential to know the origin of the datum (ECUData), but also to always have the reference to the ECU in the notifications.
ECUdata	ECUData is the actual data generated by the ECU sensors (transformed into digital by the intelligent component of the ECU). The data consists, at a certain timestamp, of temperature and kilowatt-hours, both of which are useful for studying anomalies. The data temporarily sits on the software part of the system running on the control unit (InECU) for the study of any anomaly (average data from three extractions are used). Once these data are studied, they are forgotten. *We will also highlight this later, but it is essential that a "software piece" is running on the ECU because in the event of an anomaly, but with communication problems with the central server, the ECU must still have the ability to be disabled.* The data are, every 60 seconds, sent to the central system to be saved in the central dataset (InSystem) and allow subsequent studies. The data block in transit will involve a thousand bytes that must be sent with a high precision: we do not want a very rigid system, but it is important that these data arrive accurate, with low losses and minimal latency. We expressed this semantics through a generalization that allows us to understand how data, even if it is the same, has a different meaning and is treated differently depending on where it is located.
Notice	All actors in our system exchange notifications of a thousand bytes. All anomaly-related communications are always associated with a ECU. Those, in addition, that interface with an operator will have one associated with it (to which multiple notifications can reach). • AnomalyNotice which is concerned with the notification of the anomaly to the Central Technical Service must have high accuracy. Notification must be reported promptly and with minimal loss. We can, however, tolerate very small leaks-we are not talking about critical systems. Here we have, of course, an associated anomaly. • InterventionNotice, i.e., reporting of anomaly to the technician, must also have an associated anomaly and low latency, although we can tolerate lower accuracy on this type of notification. • Notifications of the start and end of intervention are the ones we can tolerate less accuracy of, since they are only technical-Central Technical Service reports with little important data.

	Here the generalization is useful not to specify the location of the data, but to highlight that these are "notifications": the intent is to express the different types of notifications that the system can handle. We have, in addition, notifications associated with the policy, which have been described in another rationale.
Anomaly	The system keeps a pool of anomalies to refer to (the smart component and notifications are based on this possible pool of reportable anomalies). Each anomaly has a name, description, and a reference macro-category (0 - temperature, 1 - kilowatts, ...). Notes. it is not necessary to include this data within the start and end notifications. The anomaly should be associated only in the case where the operator was unable to solve it totally.
Operator	Each operator has some basic information (the classical identification data), as well as the "technical knowledge" (TechnicalKnowledge) that distinguishes it (to operate) and a binary availability flag at that given instant. The relationship between Operator and GPS is an 1-1 because we are not interested in keeping a history of positions, but the operator will always have only the associated position at the given time provided by the GPS system.
PolicyStatistics	As we know, every 30 days, the system attempts to report whether there are enforceable policies to provision ECUs at appropriate levels, such as inserting, moving, or upgrading the distribution center. Whenever a new policy is found to be applicable (and only then), the system sends a notification to the Central Technical Service (PolicyStatisticsNotice). The statistics cover some basic statistical indicators, data on the analyzed power plants, an intensity of the need to implement the new policy (five-value scale), time range of the analyzed data, as well as the location of the analyzed power plants (identified by an area defined by center and radius).
SearchParameters	When identifying the operator for an intervention, it is necessary that we delineate the area around the control unit in which to locate the technician, as well as provide the skills we are looking for in it.

Note. We have not indicated data such as id in this diagram since that is the task of the concrete design. Instead, we have emphasized the main attributes needed to model functional and non-functional requirements (architectural level).

Activity diagrams – "WHY" & "HOW" specifications

link to all diagrams here

In this section we seek to understand **what actions need to be performed on the data** to succeed in fulfilling the requirements outlined.

Note.

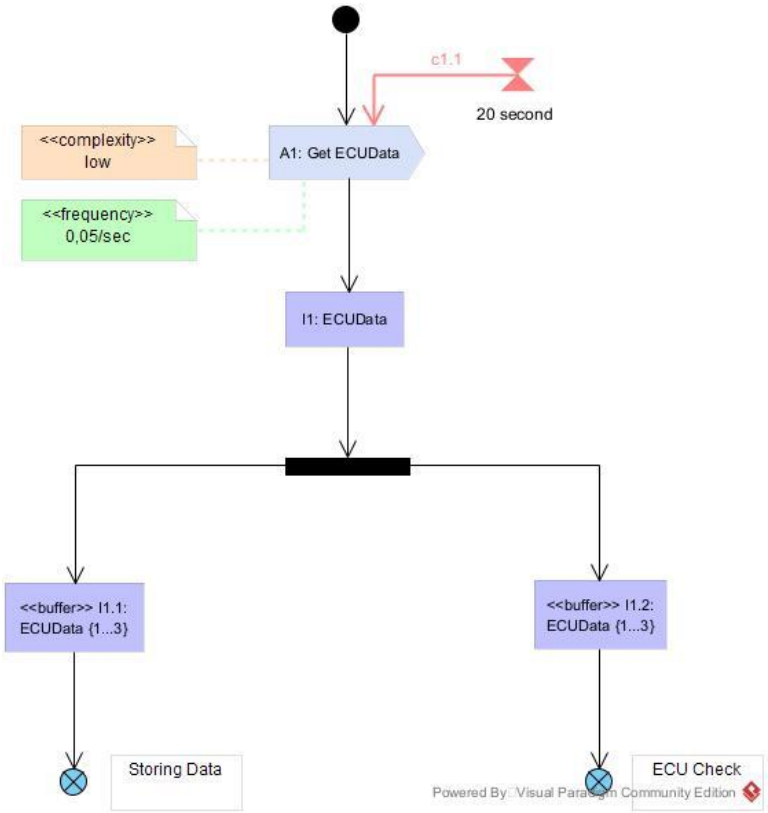
- we highlight through this diagram the close **link between static and dynamic views**;
- **identifiers on data and actions are critical**, especially for when we use the same data in different activity diagrams.
- **We do not put anomaly or incorrect data in these diagrams**; there are ad-hoc views.

A good rule of thumb is that for each use case we try to understand what **macro-actions are expected to be done** and **what data flows between them**.

Also, in the HOW diagram we can insert **labels** to indicate, for example, the **complexity** of a certain action: one he aspects that allows us to size the system both in terms of hardware platforms and in terms of what to collapse into the same module. The modules must be cohesive and compact.

Along with the HOW diagram we insert the **WHEN**, an extremely important aspect because it, too, allows us to **properly size the system**. Going to figure out the delay/timing/response time as well as the frequency is really complex. We have to **make assumptions and calculate the respective values**, considering, always, the worst case.

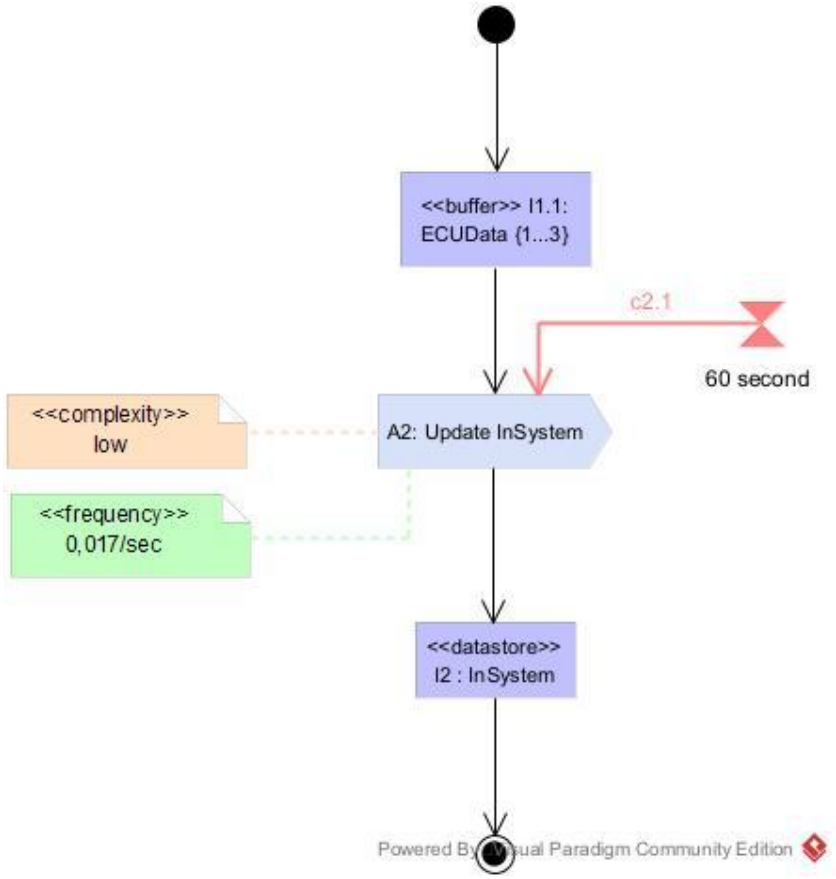
"Data acquisition" use case.



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A1	The task allows the extraction, frequently every 20 seconds, of a "data block" from the intelligent component of the control unit. The complexity is low.

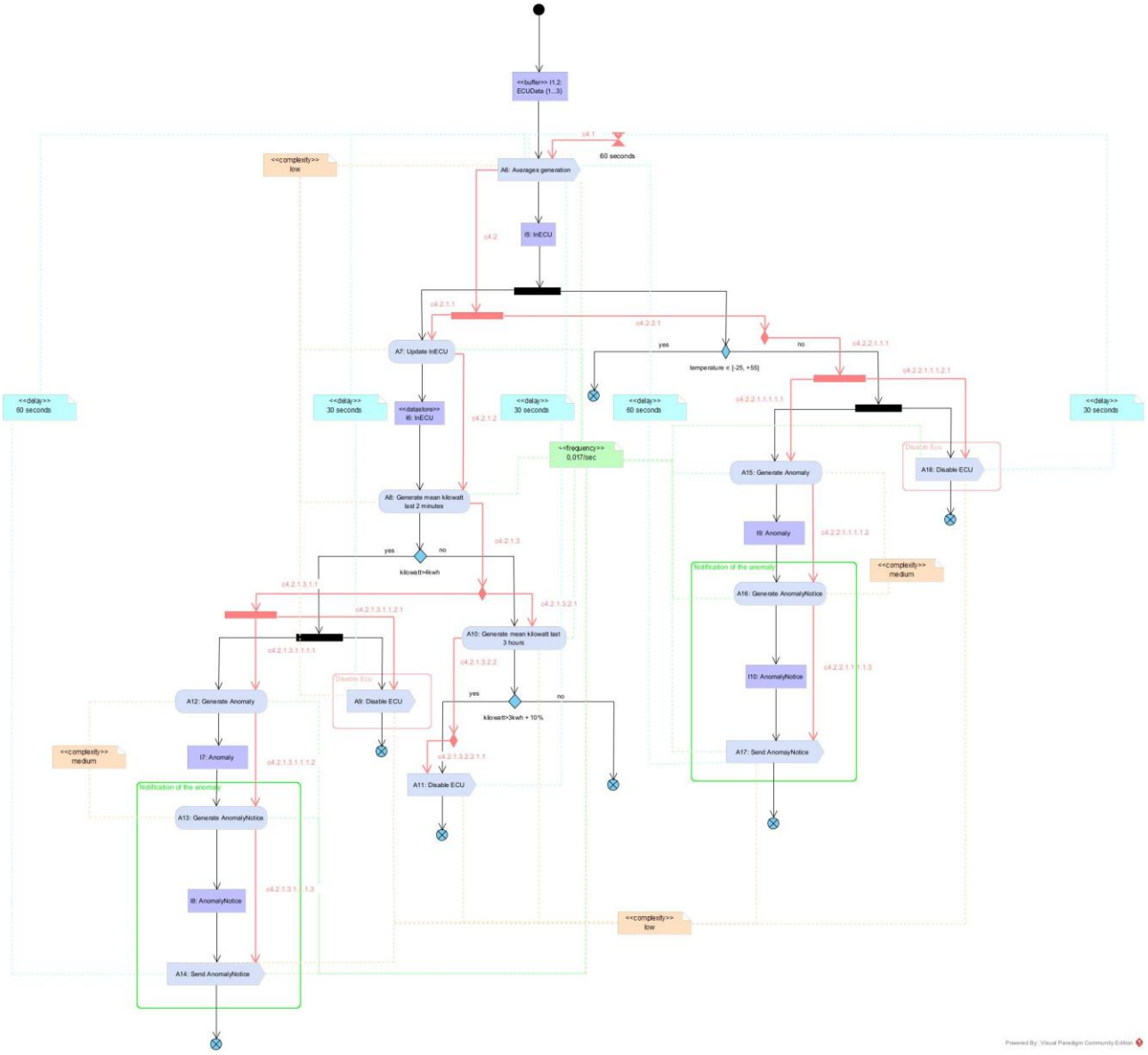
“Storing data” use case.



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A2	The task allows the extraction, at a frequency of every 60 seconds, of the three "data blocks" in the buffer of our software running on the control unit to save them in the I2 datastore and allows, subsequently, an analysis of them. The complexity is low.

“ECU check” use case with its extensions: “Disable ECU” & “Notification of the anomaly”.

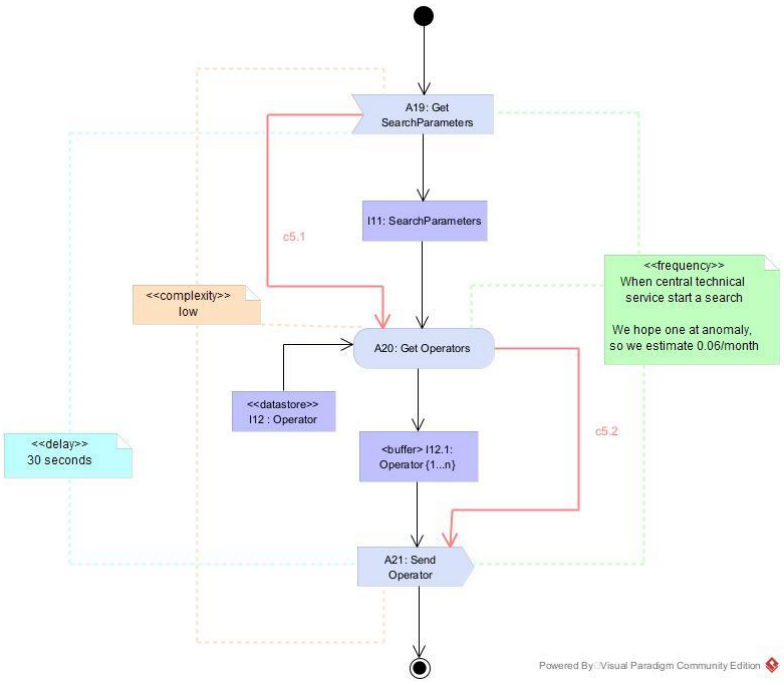


Let us specify of the **rationale** to allow full understanding of the choices. *As the diagram is very large, we recommend viewing it at this link.*

Element	Responasability
A6	The task allows the generation of an average, every 60 seconds, of the data in the ECU buffer ("three blocks") to prevent a single anomalous/"dirty" data from affecting the reporting of the anomaly. The complexity is low.
A7	Since it is necessary to maintain a history of up to 3 hours to allow adequate study of anomalies (as indicated by ARERA), this activity updates an internal datastore in the control unit with the I6 data. The complexity is low.
A8	The activity generates the average kilowatts of the last 2 minutes because if it exceeds 4kw/h, the anomaly is to be reported and the controller to be disabled. The complexity is low.
A9	The activity, if there is an anomaly, deactivates the control unit. The complexity is low.

A10	The activity generates the average kilowatts of the last 3 hours because if it exceeds 3kw/h + 10%, the control unit is to be turned off. The complexity is low.
A11	The activity, if there is an anomaly, deactivates the control unit. The complexity is low.
A12	The activity allows, should an anomaly signal be identified, to generate it. The complexity is medium.
A13	The activity allows, whenever there is an anomaly to be reported, to generate a notification. The complexity is medium.
A14	The activity allows the generated anomaly to be forwarded to the Central Technical System. The complexity is low.
A15	The activity allows, should an anomaly signal be identified, to generate it. The complexity is medium.
A16	The activity allows, whenever there is an anomaly to be reported, to generate a notification. The complexity is medium.
A17	The activity allows the generated anomaly to be forwarded to the Central Technical System. The complexity is low.
A18	The activity, if there is an anomaly, deactivates the control unit. The complexity is low.

"Operator identification" use case

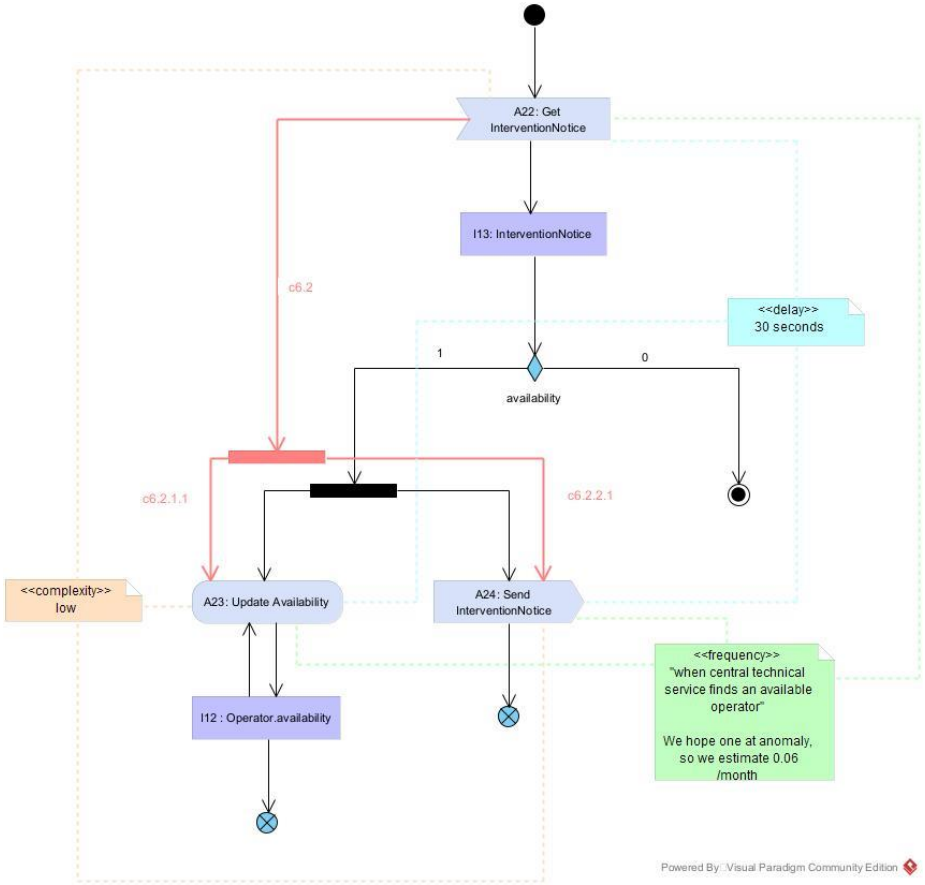


Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A19	The activity allows, through input from a CTS operator, to obtain the search parameters needed to identify a suitable technician to resolve the anomaly, based on his or her location and TechnicalKnowledge.
A20	The activity allows, by interfacing with the datastore operator, to return operators that satisfy the I11 parameters.

A21	The activity allows operators found to be sent to the CTS for selection of the appropriate one.

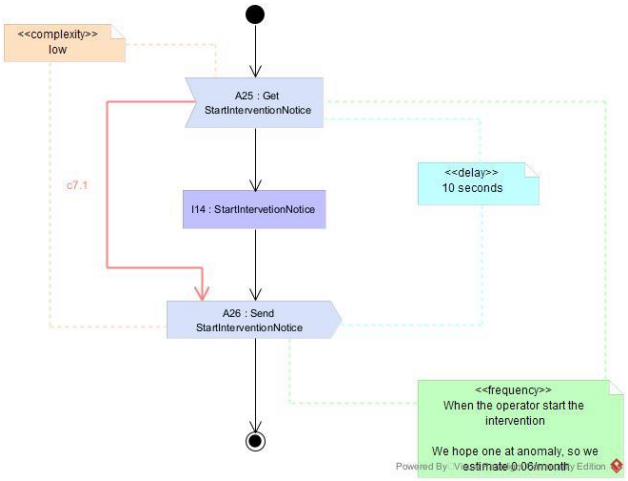
"Sending action report" use case.



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A22	The activity allows, through input from a CTS operator, to generate a notification of need for intervention (associated with a technician).
A23	The activity allows, if the operator is available, to make it "unavailable" by assigning it, by extension, to the anomaly described in the notification.
A24	The activity allows, if the operator is available, to forward the intervention notification to him.

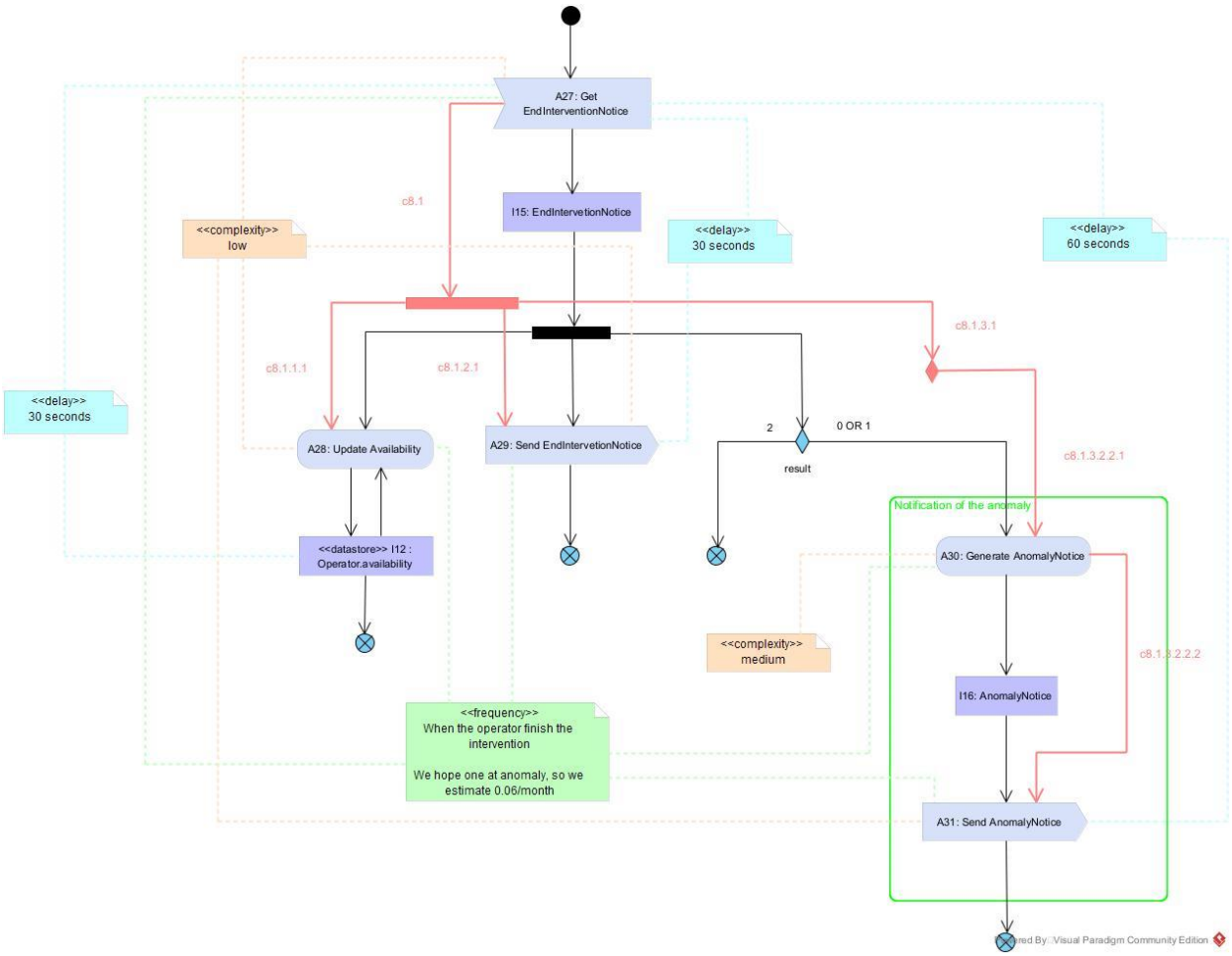
“Intervention start report” use case



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A15	The activity allows, through input from a technical, to generate a notification of initiation of intervention.
A26	The activity allows to forward the start intervention notification to the CTS.

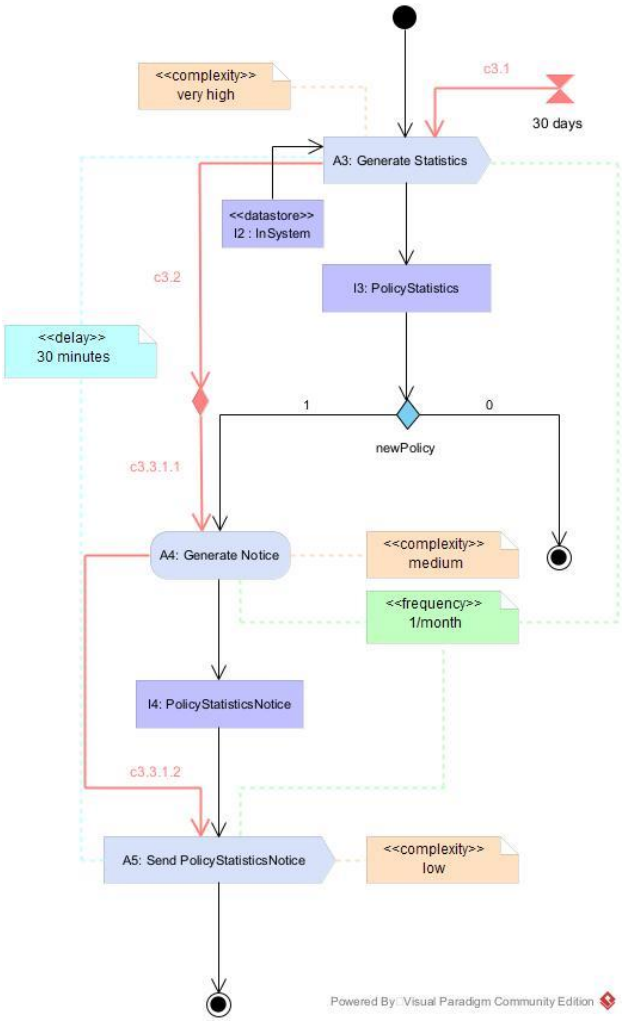
“Completion of intervention report” use case



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A27	The activity allows, through input from a technical, to generate a notification of end of intervention.
A28	The activity allows to make the technical "available" (finished the intervention).
A29	The activity allows to forward the end intervention notification to the CTS.
A30	The activity allows, if the anomaly has not been resolved or has been partially resolved, to generate a new anomaly notification.
A31	The activity allows to forward the anomaly notification to the CTS.

"Policy Optimization" use case with its extension: "Notification policy"



Let us specify of the **rationale** to allow full understanding of the choices.

Element	Responasability
A3	The activity allows the generation, frequently every 30 days, of statistics associated with the data through complex techniques to evaluate new policies, where policy is defined as inserting, moving, or upgrading the distribution unit. The frequency is once a month and the complexity is very high.

A4	The activity allows, whenever there is a new policy to be applied (newPolicy == 1), to generate a notification. The complexity is medium.
A5	The activity allows the policy to be forwarded to the Central Technical System. The complexity is low.

Logic Architecture

Introduction - a bit of theory

The goal of logical architecture is to define **clusters of activities** called **logical components**,. Components are pieces of software that can be instantiate from 1...N. The idea is to cluster activities so that we get components:

- **Compact** (or Cohesive, Homogeneous)
  It is good to collect in a component functionality that is homogeneous with each other in terms of nonfunctional requirements;
- **Isolated** (or Decoupled)
  It is good that information flows and resource sharing between components should be as limited as possible. In other words, the component should not depend on other components. This ensures to minimize overhead in communication, have flexibility in distributed deployment, and achieve easy maintainability.

In general, then, we have a set of **activities** (identified earlier) **that are to be allocated to the correct components**. There is no general rule for partitioning activities into components. This is a relevant issue that requires **experience**.

There is no such thing as the right architecture: we have to choose the **best compromise with respect to the relevant dimensions of the problem**. What is generally done is to choose one dimension as the driver and then partition based on it. The dimensions to be considered are either provided by the stackholders (through, for example, elicitation techniques-which we will not be able to do in this project), or dictated by the environement (the constraints it gives us).

The **dimension** is a property we want to consider, along with its set of assumable values.

Dimension	Description	Level	Typology
Abstraction	Level of abstraction of information processed (what)	**Architectural**	**Structural**
Complexity	Computational complexity		
Frequency	Cadence with which activities are carried out.		
Delay	Maximum time required to process.		
Location	Physical/virtual location.		
Extra Flow	Intensity of information flows between component instances and the external environment (i.e., actors).		**Dynamics**
Intra Flow	Intensity of information flows between component instances.		
Sharing	Intensity of information sharing (typically persistent data) with other component instances.		
Control Flow	Intensity of control flows between instances of the components.		

** The dimensions given are not all possible dimensions, but those that we will analyze in our study*

Once you choose a partitioning on one dimension, you then have to choose the **multiplicity** (how many instances will be in the system).

But ... **how do we evaluate whether an adopted solution is acceptable**?

- **Spread**, tells us how compact a solution is.
  Projection of a component onto a dimension is the set of values taken by its assets in that dimension. In this view, the spread (measure of projection) of a component on a dimension is a quantitative estimate of how wide the projection of a dimension is on that dimension. If a component takes on so many values it "sucks".

  Given a dimension D, the spread of the chosen solution (in terms of component partitioning) might be:

$$spread_D = \frac{\sum_1^n spread_i}{n}$$

- **Interference**, tells me how isolated a solution is.
  Overlap of a component on a dynamic dimension is the union of its intersections with other components.We have two aspects:
  - Structural aspect, i.e., the number of interfaces/resources required that therefore create dependencies with other components;
  - Dynamic aspect, that is, the intensity of simultaneous flows. Invocation, for example, of the same interface simultaneously from multiple component instances.

  Interference of a component on a dimension is a quantitative estimate of its overlap on that dimension. Interference estimation should be done by considering component instances simultaneously running, not on the individual component.

  The number of component instances and actors that are at runtime must be considered. It is, therefore, an estimation for which there is no clear-cut formula; in fact, one must consider the snapshot of the system at a given instant (which and how many instances are running) and estimate the values of each dimension.

Warning. It is wrong to reason about the individual component; what you have to do is to look at the whole solution and evaluate.

The given definitions make it possible to determine whether **components are compact and insulated**:

- **Compactness** means that the spread of a component over all structural dimensions is low. Spread is related to compactness because it indicates the homogeneity of the component that is related to structural properties,
- **Isolation** means that the interference of a component on all dynamic dimensions is low. Interference is related to isolation since it indicates the level of interaction between components.

The **footprint is the set of all spreads and interferences**. The goal, therefore, is to obtain as **small a footprint as possible on all dimensions**. We can see this as an alternative way of formulating the well-known criterion of high cohesion and low coupling.

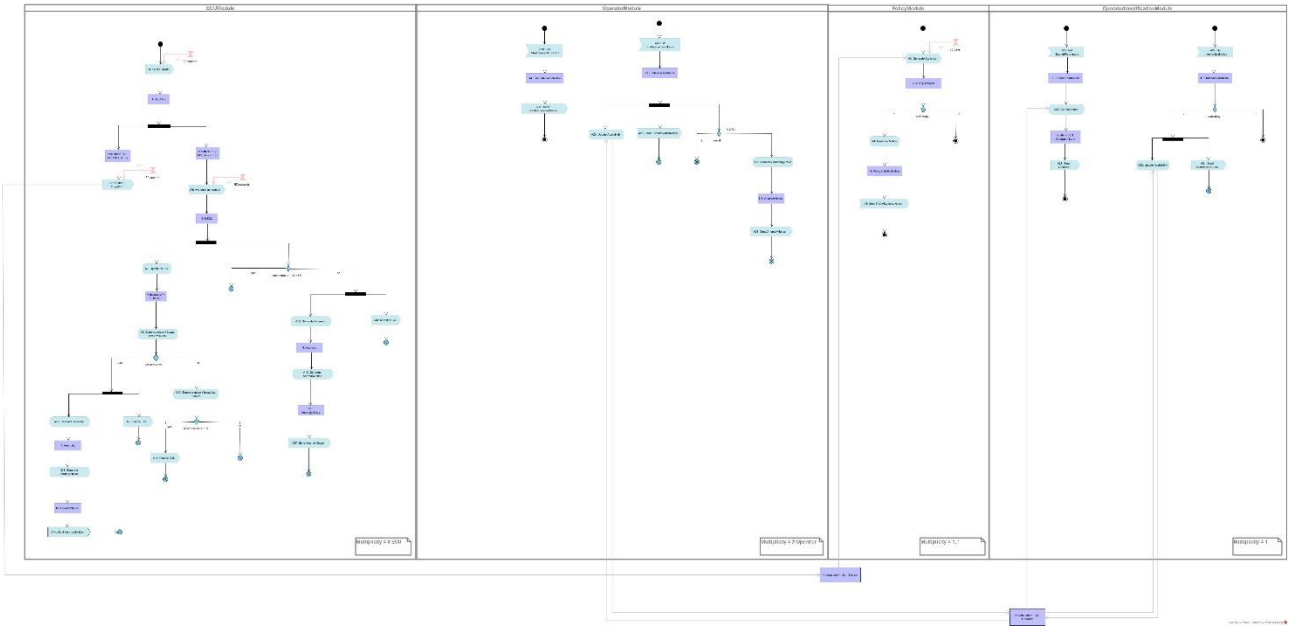
We use, therefore, a **Radar diagram to view the goodness of the solution**.

How do we ultimately **evaluate the goodness of a solution**?

- If a solution has homogeneous activities from the perspective of a static dimension (e.g., complexity), it will have a **low spread for that dimension**;
- If a solution has few interactions (with actors, other components, or datastores), it will have **low interference for that dimension**

Our solution.

link here



We report, therefore, **some values of the static dimensions of our system**:

- **Frequency**.
  The cardinality of the dimension "Frequency" is |Frequency| = 4. Let x be a value assumed by "Frequency," x ∈ {"0.06/month," "0.05/sec," "0.017/sec," "1/month"}.
- **Complexity**.
  The cardinality of the dimension "Complexity" is |Complexity| = 3. Let y be a value assumable by "Complexity", y ∈ {"very high", "low", "medium"}.
- **Delay**.
  The cardinality of the dimension "Delay" is |Delay| = 4. Let z be a value assumable by "Delay," z ∈ {"30 seconds," "60 seconds," "10 seconds," "30 minutes"}.
- **Location.**
  The cardinality of the dimension "Location" is |Location| = *. Let a be a value assumable by "Location", a ∈ {* GPS + * ADDRESS + Central System}.
- **Abstraction** - Which in our case means "different data we deal with."
  The cardinality of the dimension "Abstraction" is |Abstraction| = 16, having 16 different types of data. Let b be a value assumable by "Location," b ∈ {I1, ..., I31}.

We guided the **division of our system into modules by following the following drivers**:

1. First, fundamental is the breakdown by driver **location**:
   a. The **ECUModule** must reside on the ECU; it will, therefore, have multiplicity equal to the number of ECUs. This is to decrease the number of iterations with

Central Technical Service: it is useless, for example, to send data for fault checking to Central Technical Service when it is possible to check on the ECU itself. Otherwise, the CTS would have to handle so much data coming, every 60 seconds, from 10,800 ECUs. This choice, in addition, allows the ECU to be disabled, in case of an anomaly, even if communication with the CTS is not possible. In addition, for the same reason, the data storing task is also placed in this component (it will save data in the datastore in common: I2).
Let us, therefore, analyze some estimates that we will need for the analysis of the solution as a whole (remember that these are estimates, it is not an exact figure).
***NB**: It is incorrect to analyze the footprint of only one component; however, we are estimating some values associated with the size of this component to allow us, later, to estimate the footprint of the solution as a whole.*

Dimensioni	Proiezione	Spread
Abstraction	The module deals with 7 of the 16 data.	7/16 = 44/100
Complexity	The module incorporates 2 of 3 complexities.	2/3= 67/100
Frequency	The module incorporates 2 of 6 frequencies.	2/6 = 33/100
Delay	The module incorporates 2 of 4 delays present.	2/4 = 50/100
Location	The module is placed on the ECU and must interact with the CTS. Considering, even if only, the 10,800 positions of the ECUs, we can say that we have a near spread value close to 0.	approx. 0/100

Dimensioni	Why	Interferenza
Extra Flow	The module interacts with the ECU on which it is placed and the CTS operators. Considering, even just, the 10,800 ECUs we have, we can say that we have a near value of "null" interference.	null
Intra Flow	The module does not exchange information with others.	-
Sharing	Sharing a datastore.	medium
Control Flow	The module does not interface with other modules.	-

See the diagram at the following link

b. The **OperatorModule** must reside on the devices provided to the operators, it will, therefore, have multiplicity equal to the number of Technical. This is to decrease the number of iterations with Central Technical Service and to group the functionalities concerning iteration from the operator to the CTS. This module, therefore, will not have to worry about competitively handling all requests regarding the use cases "Intervention Start Report" and "Completion of

intervention report," but will have to interface only with those of the Technical of reference.
***NB**: It is incorrect to analyze the footprint of only one component; however, we are estimating some values associated with the size of this component to allow us, later, to estimate the footprint of the solution as a whole.*

Dimensioni	Proiezione	Spread
Abstraction	The module deals with 3 of the 16 data.	3/16 = 19/100
Complexity	The module incorporates 2 of 3 complexities.	2/3= 67/100
Frequency	The module incorporates 1 of 6 frequencies.	1/6 = 17/100
Delay	The module incorporates 2 of 4 delays present.	2/4 = 50/100
Location	The component is placed on the technician's device and must interact with the CTS. We know, by assumption, that there are 4 technicians (4 positions) + 1 of the CTS..	2/5 = 40/100

Dimensioni	Why	Value
Extra Flow	The component interacts with the technician operating from the device and CTS operators.	low
Intra Flow	The module does not exchange information with others.	-
Sharing	Sharing a datastore.	medium
Control Flow	The module does not interface with other modules.	-

See the diagram at the following link

c. The **remaining two components**, according to this driver, **remain clustered on the central system**.

2. We subdivide the module located on the core system according to the **frequency driver**.
   a. **PolicyModule**.
   The "PolicyOptimization" occurs once a month and has a "very high" complexity because it combines machine learning techniques to assess the need for new deployment policies.For this reason we thought of dedicating a module specifically for it "PolicyModule".The module will be created and destroyed once the operation is finished (multiplicity 1..*), this is because the cost of creating the component is much less than its operating time. Also, it would be unnecessary to keep such a component "active" consuming resources, running once a month for an estimated 30 minutes.
   ***NB**: It is incorrect to analyze the footprint of only one component; however, we are estimating some values associated with the size of this component to allow us, later, to estimate the footprint of the solution as a whole.*

Dimensioni	Why	Value
Abstraction	The module deals with 2 of the 16 data.	2/16 = 13/100
Complexity	The module incorporates 3 of 3 complexities.	3/3= 100/100
Frequency	The module incorporates 1 of 6 frequencies.	1/6 = 17/100
Delay	The module incorporates 1 of 4 delays present.	1/4 = 25/100
Location	The module interacts only with the CTS. Considering, even if only, the 10,800 positions of the ECUs, we can say that we have a near spread value close to 0.	approx. 0/100

Dimensioni	Why	Value
Extra Flow	The module only interacts with the CTS operators. Considering, even just, the 10,800 ECUs we have, we can say that we have a near value of "null" interference.	“null”
Intra Flow	The module does not exchange information with others.	-
Sharing	Sharing a datastore.	medium
Control Flow	The module does not interface with other modules.	-

See the diagram at the following link

b. **OperatorIdentificationModule**.
The last module deals with "Operator Identification" & "Sending Action Report." These are closely related aspects since a technician must be identified before he or she can be connected to a request for troubleshooting. For this reason they have been grouped together: they have similar frequencies and complexity.It is important to point out, moreover, that in the assumption, given by estimates, of about 18 monthly anomalies, we can, safely, think of dedicating a single module (multiplicity 1), always running, for handling these types of operations.
***NB**: It is incorrect to analyze the footprint of only one component; however, we are estimating some values associated with the size of this component to allow us, later, to estimate the footprint of the solution as a whole.*

Dimensioni	Why	Value
Abstraction	The module deals with 3 of the 16 data.	3/16 = 19/100
Complexity	The module incorporates 1 of 3 complexities	1/3= 33/100
Frequency	The module incorporates 1 of 6 frequencies.	1/6 = 17/100

Delay	The module incorporates 1 of 4 delays present.	1/4 = 25/100
Location	The module must interface with operators in various positions and the CTS.	low

Dimensioni	Why	Value
Extra Flow	The module must interface with operators and the CTS.	low
Intra Flow	The module does not exchange information with others.	-
Sharing	Sharing a datastore.	medium
Control Flow	The module does not interface with other modules.	-

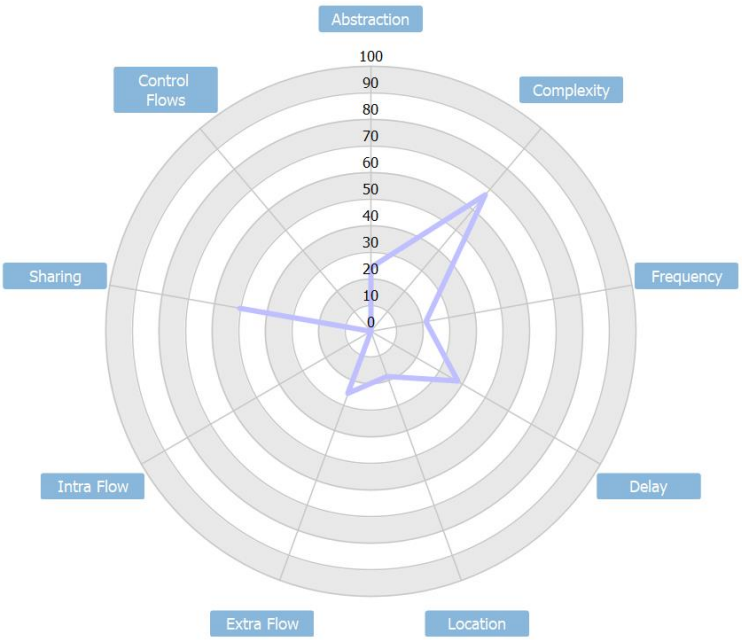
See the diagram at the following link

Let us analyze, then, the **footprint of the entire solution**.

		Dimension	Values	
statistics	**proiezioni**	**Abstraction**	low	Estimate calculated from the estimates of individual modules (we did "the average").
		Complexity	high	Estimate calculated from the estimates of individual modules (we did "the average").
		Frequency	low	Estimate calculated from the estimates of individual modules (we did "the average").
		Delay	low/medium	Estimate calculated from the estimates of individual modules (we did "the average").
		Location	very low	Estimate calculated from the estimates of individual modules (we did "the average").
dynamics	**overlap**	**Extra Flow**	low	By operating on the appropriate drivers and acting on the multiplicity of modules (1 ECUModule per ECU and 1 OperatorModule per technician) we were able to lower the overall interference of our solution regarding Extra Flow.
		Intra Flow	-	The modules do not exchange information with others.
		Sharing	medium	Some modules share datastores. ECUModule & PolicyModule I2,

				OperatorModule & OperatorIdentificationModule I12.
		Control Flow	-	The modules do not interface with each other.

		Dimension	Values
statistiche	**spread**	Abstraction	24
		Complexity	67
		Frequency	21
		Delay	38
		Location	18
dynamic	**interference**	Extra Flow	25
		Intra Flow	0
		Sharing	50
		Control Flow	0



radar here

Our solution is a **good solution**. The dimension that impacts the footprint the worst, however, is **complexity**. We can, therefore, think of mitigating this problem by making use of **appropriate hardware**.

Concrete Architecture

Introduction - a bit of theory

We have seen, with the logical architecture, which components make up our system, but we have not defined **how they interact with each other and how they are internally structured**. Concrete architecture specifies the **set of concrete components** by showing the **internal design of the components and the interactions between them**. It has the objective of defining:

- **Information and control flows are concretized in terms of interaction styles**:
  - communication mechanisms (control flows);
  - information transfer (information flows).

  Let's go into detail (how the connectors should be created). There are two basic types of communication between components:

  - **Synchronous**. The parties continuously listen and act on the other party's responses. Synchronous communication, therefore, means that a component remains inactive until it receives a call, a response, ...
  - **Asynchronous**. Parties do not actively listen to messages. Communication typically has a delay (between when the sender initiates the message and when the recipient responds, if a response is expected). Unlike synchronous, asynchronous communication allows the component to continue functioning after generating a call or an answer.

  In particular:
  - **Control flows** send only control signals, no data. It is important to underline that they only implement the "why", not the "when": sending a report does not imply that the component must be activated. Basically the control flows are:
    - **Synchronous;**
    - **Asynchronous**.
  - As regards **information flows**, they only transfer data/information, no control signals. The controls can be implemented by different components. Two basic styles:
    - **Push**. Data sending. Based on an asynchronous communication mechanism (whoever sends the data releases it as soon as the data is sent). We expect the recipient of the data (destination) to expose an API so that it can be called.
    - **Pull**. Data recovery. Based on a synchronous communication mechanism (there is a wait for the data).
- The **internal design of the components**. The internal design of the component is closely linked to the interaction styles with other components. They can be:
  - **Active** (autonomous) **vs passive** (called by a synchronous event: cannot work autonomously, requires synchrony);
  - **Stateless vs statefull** (preserves a state between executions).

  It is therefore important to immediately highlight that internally we are not going to define, given a component, which modules it is composed of, but we are simply going to highlight whether they are components that have a state (i.e. the way in which they will interact with the others) and whether they are active vs passive.

A **component** is, basically, **a software unit that should have a functionality**. We expect, therefore, that there is an "execute method", not visible to the outside (it is not part of the API), this is because the actual execution of the functionality must be performed according to **internal strategies of the component** which may, or may not, depend on the the fact that control flows arrive rather than data flows (it is not certain that the execute will start immediately once the data has been received!).

NB:

- the when is something hardwired as an internal strategy of the component that does the processing. The only thing visible from the outside, therefore, are the interfaces, not the execute;
- the concrete components are made from the class diagram and the iterations from sequence diagrams.

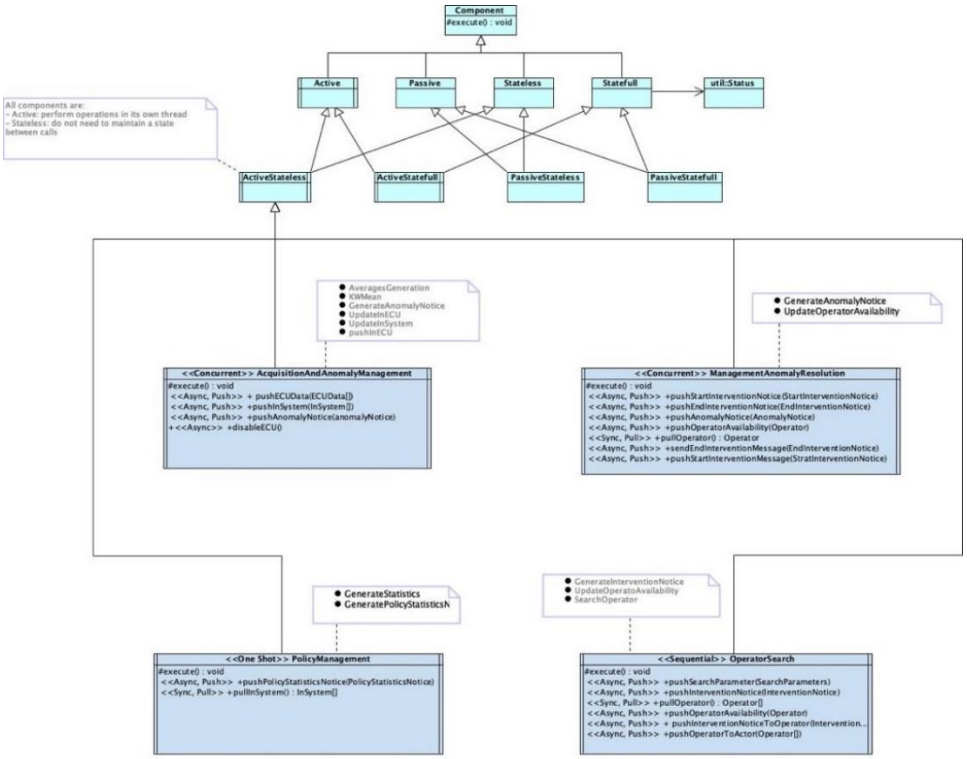
In short, concrete architecture does nothing more than take as input the logical components that were identified in the logical architecture phase and enrich them with more concrete information regarding precisely **the way in which they interact with the other components and the internal design.**

Our Solution

By class diagram, as highlighted in the theoretical section, we show the **internal design of the components**. The concrete components are related 1:1 to the logical components. We have:

- **AcquisitionAndAnomalyManagement,** for handling all communication from the technician to the CTS regarding the intervention (running on each technicians' device).
- **ManagementAnomalyResolution,** for anomaly management and identification, as well as saving data for later studies (running on each ECU).
- **PolicyManagement,** for the study and definition of new policies (the component is created and destroyed at each run)
- **OperatorSearch,** for identification of an appropriate operator to resolve the anomaly (running on the central system).

diagram here



All components implemented in the class diagram are:

- **Active**. It was chosen to make the components active because, the way the system is designed, each component must be autonomous and de operations performed in its own thread.
- **Statetless**. The components are stateless because, as specified in the assumptions, the system relies on a message exchange that shares details of operations. There is, therefore, no need to preserve state between one execution and the next.

Most of the functionality implemented by the components is of the **information flow type**. In fact, the different **modules implemented do not communicate directly with each other**, but predominantly communicate with external components such as actors or databases, which makes it necessary to implement functions involving data exchange.

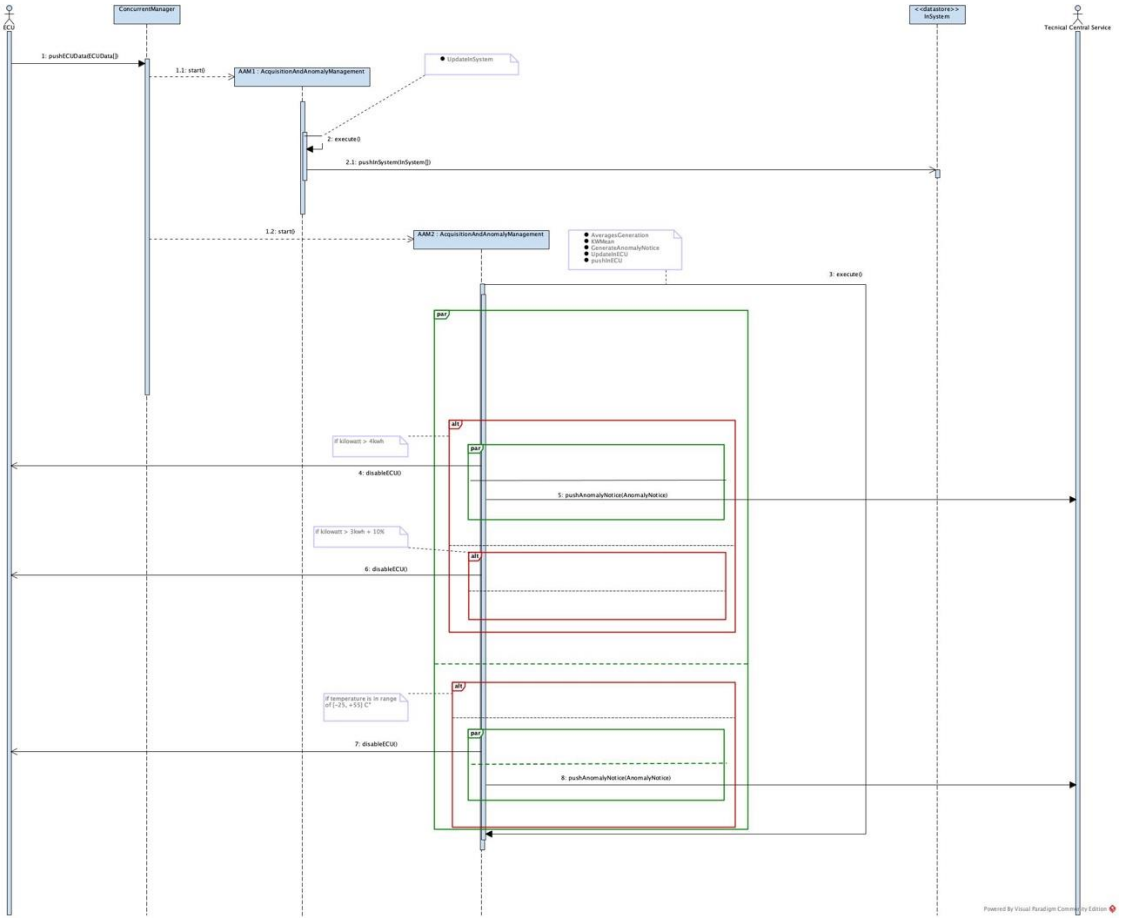
Note. The annotations in the diagram provide an overview of the actual functionalities that will be implemented that are "internal" to execute().

At this point we are going to report the **sequence diagrams** showing the operations/iterations of the different components.

Note. Notes have been included in the diagrams that, once again, show in detail the operations included in the execute() method, as well as the conditions of the fragment alternatives and any additional information that enhances understanding.

link to all diagrams here

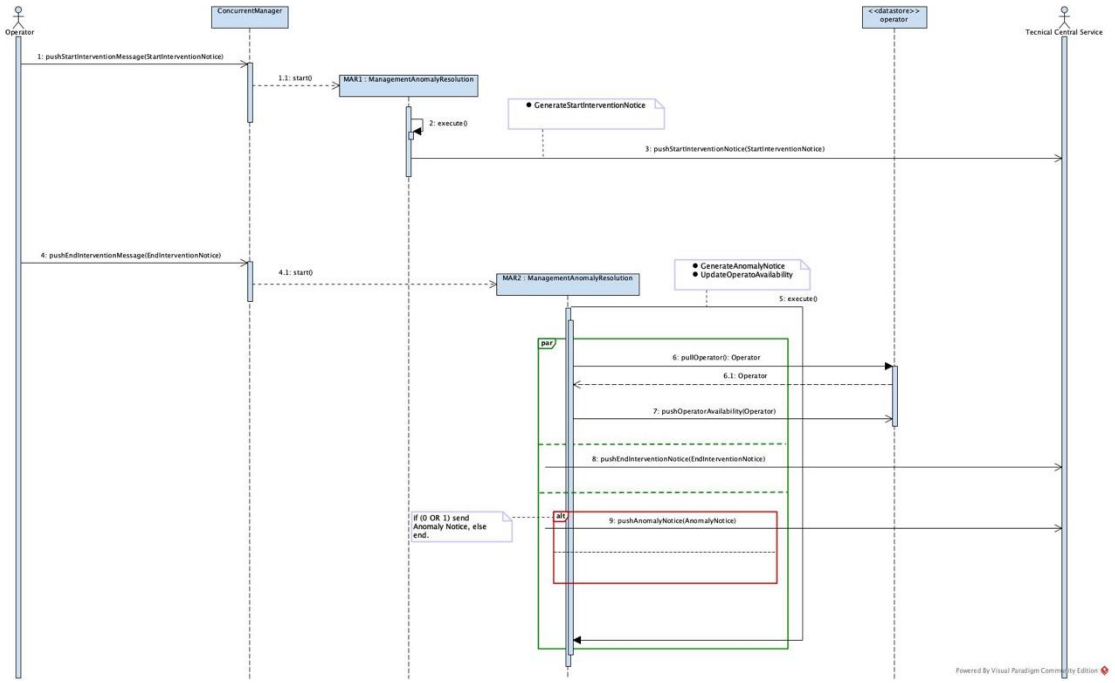
AcquisitionAndAnomalyManagement



The following component was implemented with as "**Concurrent**" executor. As we know, each ECU **manages internally the identification and handling of the anomaly** (logical module: ECUModule), as well as the **persistent storing of data**. We will, therefore, have an internal process within each ECU that controls **several threads, each of which handles, in parallel**, different streams: data storing and anomaly handling.
The ECU actor sends the ECUData[] data to a ConcurrentManager, which will handle the parallel threads of the process:

- A **first thread will take care of storing the data on the InSystem datastore** (I2);
- A **second thread will take care of handling the anomaly**.

ManagementAnomalyResolution

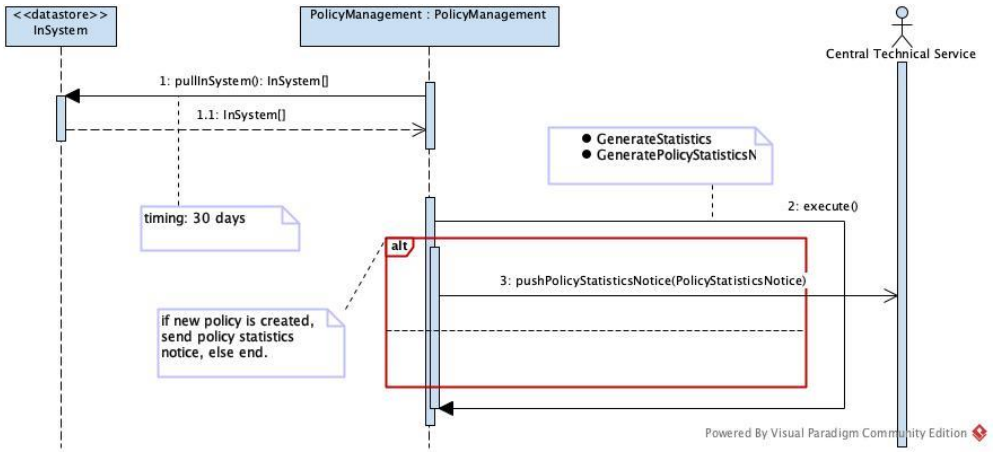


The following component has also been implemented as a **"Concurrent"** executor and is running on **each device provided to the technicians**. Once again, a main process creates and controls the appropriate threads:

- the first is **created contextually to the receipt of the StartInterventionNotice** data with the intent of signaling the start of the intervention to the CTS;
- the **second is created contextually with the receipt of the EndInterventionNotice data** with the intent of signalling the end of the intervention to the CTS. When the EndInterventionNotice is sent, the operations described by the logic diagram are performed in parallel.

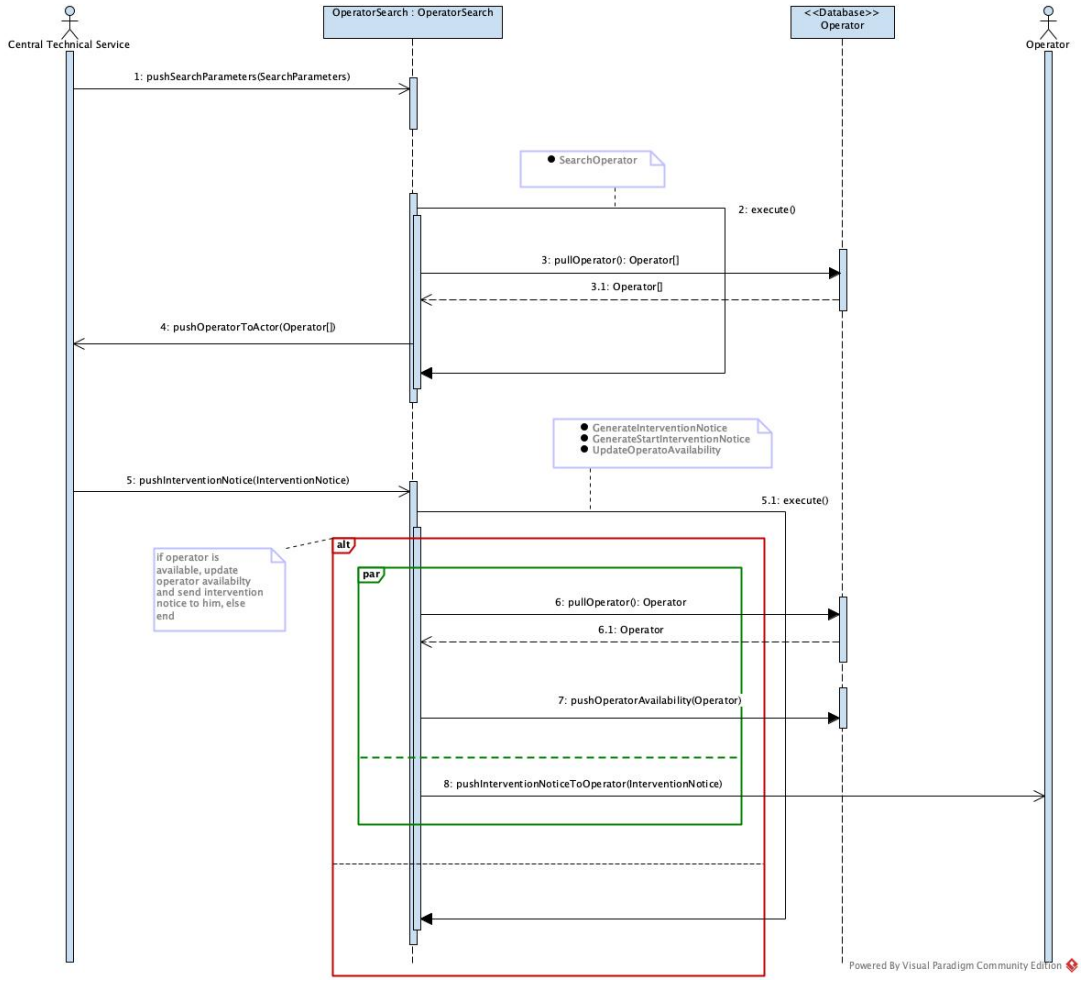
The component “sends” data to the Operator datastore (I12) of and the CTS actor.

PolicyManagement



The following component has been implemented as a "**One Shot**" executor. The reason behind this choice is since the operations the component performs are executed **infrequently**, as they are done every 30 days, as specified by the annotation, and **the creation does not require an excessive overhead compared to the duration of the execute**.
PolicyManagement retrieves the data for the definition of a new policy via a pull-type information flow and sends a notification containing the statistics generated if a new policy has been identified for implementation.

OperatorSearch



The following component has been implemented as a "**Sequential**" executor. The reason behind this choice, besides the fact that operations are **handled sequentially** by the same process, is that they perform short tasks with a **fair frequency**. In fact, this component is responsible for (1) **managing the search for available technicians to troubleshoot an anomaly** when the CTS requests it and (2) **notifying that technician of the need for intervention**.

The (1) first phase of the process deals, in detail, with retrieving all the Operators that have a match with the search parameters sent by the CTS (SearchParameters - I11). The second phase (2) of the process deals, following the CTS's selection of the Operator, with sending the intervention notification and updating the Operator's availability field (stored in the datastore I12).

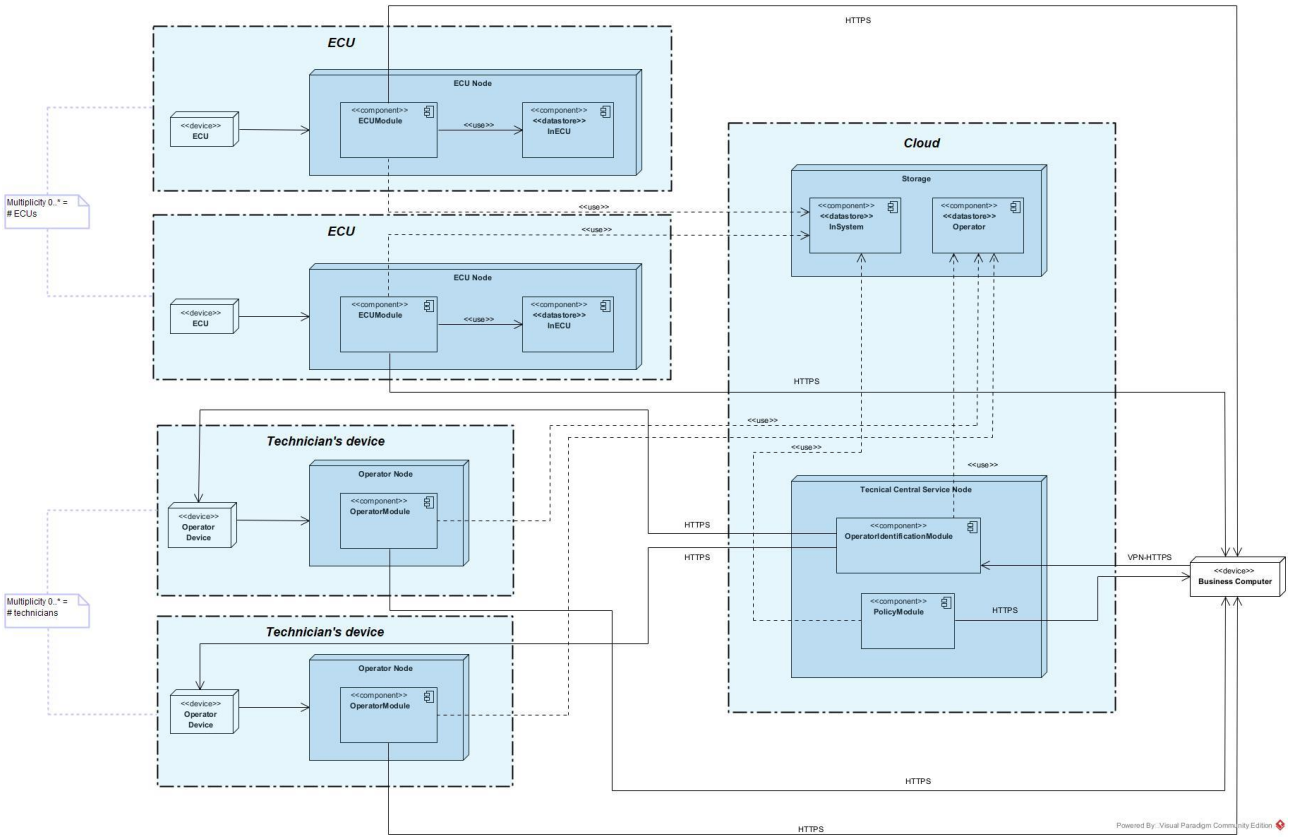
Deployment Architecture

Introduction - a bit of theory

The goal of the deployment architecture is to **define and identify the computational nodes needed for our solution.** Also at this stage we need to **estimate the costs** (hardware and people) as well as **allocate component instances in the computational nodes**.

Our Solution

link [here]



The image shows the **deployment** solution from our system. **Communication between components and actors** is via the **HTTPS protocol**. This protocol is mainly used to ensure secure/cypher communications. The **advantages** of using the HTTPS protocol are as follows:

- **Data Encryption.** It is essential when transferring sensitive data;
- **Authentication.** It provides a mechanism to ensure that the user is communicating with the actual website and not a copycat;
- **Protection Against Man-in-the-Middle Attacks.** Helps protect user privacy by encrypting transmitted data, preventing malicious parties from intercepting and reading the data exchanged between the user and the website;
- **Data Integrity.** Ensures that the data sent has not been altered or corrupted during transfer, guaranteeing the integrity of the data.

To ensure a certain level of security, it was decided that CTS operators' devices must communicate with the system via VPN, this is because they have higher-level permissions. The use of VPNs in an enterprise context is a common practice. The advantages of implementing a VPN are as follows:

- **Increased security**. VPNs encrypt network traffic, protecting corporate data from eavesdropping and attacks;
- **Delocalized work**. I allow employees to securely access company resources remotely.

While **AcquisitionAndAnomalyManagement** is deployed on each ECU and **ManagementAnomalyResolution** is deployed on each technicians' device, **OperatorSearch** and **PolicyManagement**, as well as the database, are **deployed on a cloud provider**. Using cloud computing for applications and databases offers a number of **advantages** and **disadvantages**. Some of the main points to consider are listed below:

Advantages:

- **Scalability**. Cloud solutions offer the ability to flexibly scale resources according to the needs of the application or database. This allows you to handle peak loads without having to invest in additional hardware on a preemptive basis;
- **Costs**. Using the cloud can reduce the upfront costs associated with purchasing and maintaining hardware. In addition, many cloud platforms offer flexible payment models based on actual resource consumption;
- **Accessibility and flexibility**. Remote access to cloud resources enables teams to work collaboratively from different geographic locations. In addition, resources can be accessed at any time and from any device with an Internet connection.
- **Backup and recovery**. Most cloud providers offer automated backup services and the ability to restore data in case of system loss or failure.
- **Automatic Updates**. Cloud platforms often manage software updates and patches automatically, ensuring that applications are always up-to-date with the latest security fixes.

Disadvantages:

- **Dependence on Internet Connection**. Access to cloud resources requires a reliable Internet connection. In the event of network outages, temporary unavailability of resources may occur.
- **Long-Term Costs**. While cloud use may initially be cost-effective, long-term costs can accumulate. It is important to carefully monitor and optimize resource use to avoid excessive costs.
- **Limitations on Customization**. Some cloud platforms may have limitations on resource customization, which could be a problem for highly specialized applications or advanced configurations.

In general, however, the **disadvantages of Cloud**, have less impact than the advantages for our solution.

Conclusion

The team

This report is maintained by the **InforMAT "Software Architecture" course group**, whose members are:

- MATtia **Piazzalunga** - 851931;
- MATteo **Severgnini** - 851920.

We really care about the success of this project, so for any problems/understandings, we are available (we provide a contact email: m.piazzalunga2@campus.unimib.it).