Universidade do Minho

Escola de Engenharia

Departamento de Informática

Xavier Passos Rodrigues

Design a Market Hub Platform for Utilities

October 2018

Universidade do Minho

Escola de Engenharia

Departamento de Informática

Xavier Passos Rodrigues

Design a Market Hub Platform for Utilities

Dissertation Report

Master's degree in computer science

Dissertation supervised by:

António Nestor Ribeiro

Hugo José Pereira Pacheco

Rui Miguel Silva Couto

October 2018

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project «POCI-01-0145-FEDER-006961», and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia as part of Project «UID/EEA/50014/2013».

Acknowledgments

Começo por agradecer ao Professor Nestor, ao Hugo e ao Rui pela disponibilidade, pelo excelente trabalho de orientação e pelo constante incentivo para a realização deste trabalho.

Uma palavra de apreço aos 3 "heróis" da minha vida, os meus pais e irmã, porque sem eles nada disto era possível.

Agradeço à minha Cata, pelo carinho, pela motivação, por me ter aturado quando estava rabugento, mas sobretudo porque mesmo longe tentou estar sempre presente.

Um obrigado aos meus "irmãos" António e Tiago pelas tertúlias de sábado à noite, que serviram como momentos de animação e descompressão.

Agradeço também a malta do laboratório, Rui, Miguel e Marco, pela "motivação" e pelo ambiente de descontração que proporcionaram.

Por último, um grande obrigado a todos os amigos que levo destes maravilhosos 6 anos na melhor academia do pais.

Abstract

Software Engineering has been contributing, over the years, to a better and more efficient production of software. Through its methodologies and processes, it has been used to increase the assurance that the software produced is robust, of quality, easy to update, and above all that, conforms to the requirements identified by the stakeholders.

With the growth of data sharing, collection and storage in the utilities sector, there is also an urgent need to maintain and use the available information in a useful way. This requires the adoption of strategies and infrastructures that can help leveraging this form of treatment to make it possible to improve the quality of certain utility sectors (gas, electricity, water, internet, communications). However, much of this process is still nowadays controlled solely by the production and distribution companies, preventing the all other users to participate in the process.

To try to undo the supremacy of the production and distribution companies have on the utilities panorama, and in support of the European Commission vision, the energy sector is trying to move towards a liberalized market and, with this, it aims to enable all entities such as consumers, retailers, producers, distributors, to contribute to the management of the network and for new business models to emerge.

To sustain this ecosystem, where these entities can communicate and share data, it would be advantageous to have a platform that would allow the communication of such data between all users.

In this project, and through the application of SE techniques, we will develop, step by step, a model for a modular, scalable and integrated environment to enable demand response, data exploration, storage and fulfill all the European Union-wide Data Protection Regulation (GDPR).

Resumo

A Engenharia de Software tem contribuído, ao longo dos anos, para uma melhor e eficiente produção de software. Através de metodologias e processos, é possível produzir software produzido robusto, com qualidade, confiável, que possa ser atualizado e, acima de tudo, que respeite os requisitos identificados pelas partes interessadas.

Com o crescimento da partilha, coleta e armazenamento de dados no setor de serviços públicos, existe uma necessidade urgente de manter e usar as informações disponíveis de maneira útil, com que se consiga extrair conhecimento. Isso requer a adoção de estratégias e infraestruturas que possam apoiar essa forma de tratamento para que, a partir da coleta de informações, seja possível melhorar a qualidade de certos setores de serviços públicos (gás, eletricidade, água, internet, comunicações). No entanto, e nos tempos que correm, grande parte desses processos é controlada exclusivamente pelas empresas de produção e distribuição, impedindo que os demais utilizadores participem do processo.

De forma desfazer a supremacia que as empresas de produção e distribuição têm, e com apoio e motivação da Comissão Europeia, o setor energético tem vindo a tentar implementar um mercado de energia liberalizado e, com isso, permitir que todas as entidades como, consumidores, retalhistas, produtores, distribuidores, possam contribuir para a gestão da rede e na criação de novos modelos de negócio.

Para sustentar este ecossistema, onde todas as entidades podem comunicar e compartilhar dados, seria útil e vantajoso ter uma plataforma que permitisse a comunicação de tais dados entre todos os utilizadores.

Neste projeto, e através da aplicação de técnicas de SE, pretenderemos mostrar passo a passo um método para construir um ambiente modular, escalável e integrado para permitir resposta a procura, exploração dos dados, armazenamento e que cumpra a nova lei da União Europeia respeitante à Proteção de Dados (GDPR).

Summary

List of Figures

List of Tables

List of Abbreviations

CSV	Comma-separated values
DSO	Distribution System Operator
ETL	Extract, Transform and Load
GDPR	General Data Protection Regulation
IDP	Identity Provider
JSON	JavaScript Object Notation
MHP	Market Hub Platform
RUP	Rational Unified Process
SE	Software Engineering
UI	User Interface
UML	Unified Model Language
USA	United States of America
TSO	Transmission System Operator

1. Introduction

It's remarkable the importance of Software Engineering and the role that it plays in our society. Every day, around the world software is being made in order to help people in their daily tasks, and if we think on a large scale we can say that software plays an important role in the way the world is managed. National infrastructures, banks and utilities are in some way using or being controlled by software, which helps them to be more efficient. Therefore, Software Engineering is a natural part of the functioning of national and international societies.

Software Engineering, perhaps for its cross-area nature and for being a fairly recent engineering discipline, has not the same maturity level as other classical engineering disciplines, as is the case of Civil Engineering or Mechanical Engineering. Thus, it is not based on calculations or physical laws, but rather on a more abstract, intangible and sometime complex nature.

To cope with this nature, software engineers have long been developing processes, methodologies and software development patterns, to warrant better guarantees that the final software product was well-produced, compliant with the requirements and delivered within the stipulated deadlines.

Through the application of such methods for the creation of software, like requirements gathering, specification, modelling, implementation and optimization, it is possible to create a piece of software capable of fulfilling its objectives.

The creation of SE, in the early 70s, was due in part to the need to solve certain problems in the area of computer science. Its application and understanding was only dominated by some, which made this discipline restricted.

Over the years and with the emergence of new processes, software engineers realized that software engineering could be applied to other areas and therefore solve problems that were at the time out of reach.

One of the problems that would speed up, were processes that until that moment were done by humans and now were replaced by software. That resulted in the development of techniques and the reduction of costs. A clear success story from the production sector was the usage of software in the monitoring of industrial environments: software can help anticipating errors before they happen and alert in case any fault occurs.

More recently, Software engineering began to be applied in services that are provided to the general public, like water supply, communications, gas, electricity, internet, cable television, public transport, among others. These services are called utilities.

These utilities share the same underlying principle: there is a producer/distributor of a given utility, who manages a distribution network and then we have the consumers. In some cases, there are intermediary entities, such as retailers, that make up the utility's ecosystem.

On the consumer side, with the advent of IoT, there have also started to appear Smart Devices that can monitor consumer's consumption. Distributors have also began to apply software to monitor the network and perform some statistics regarding the consumption and resource management.

However, innovation in these sectors is still beyond expectation, the way to extract the maximum benefit from them still needs to be covered, and it is the purpose of this project. With the choice of good technologies, practices and software construction processes, we will try to add something useful and beneficial to these developing sectors.

1.1 Motivation

For several years, the entities that regulate utilities sectors, such as electrical power, gas supply and network providers, simultaneously controlled the distribution and commercialization of their goods, something that was difficult to change due to the technological limitations. This behaviour led to a big gap, between consumers/clients and all the entities that regulate those sectors.

Due to the technological limitations and use of closed processes to other agents, the regulators of each sector lacked the tools or infra-structures to help them on energy grid management and, with that it was difficult to ensure continuity of supply, efficiency and quality.

All these problems related to grid management, and also motivated by European laws, have been pressing the liberalization of some sectors. With that liberalization, a global market at European level was born and the paradigm where a company assumed a central role ceased to exist.

Thus, not only can several companies play an active role in supplying, but also enables various stakeholders (customers, smart devices, market players, regulators or third-parties) to participate actively in the management of the distribution grid, to speed up information exchange and even to explore new business models.

In order to drive this change, it makes sense to create an integrated and generic platform that provides data exchange and functions in a neutral way to the various users, of the different sectors, access and share data, providing a transparent access to the information.

This change is already in motion. During the last years and with the support of the European Commission, in the case of Europe, and the White House, in the case of the USA, several projects, especially in the area of electricity, have arisen in order to provide this change.

Projects like **Flexiciency**[1], **Upgrid**[2], **Green Button Initiative**[3] have emerged as an attempt to liberalize the energy sector and promote access to data. More recently started the **Integrid**[4] project, in which this project is integrated.

1.2 Objectives

The purpose of this project is to design and implement a data exchange platform for the utilities sector having as the main objective to provide an ecosystem so that both providers and other interested parties (customers, system operators, retailers or third-party service providers) have the possibility of participating and contributing to a better management of the ecosystem, through the services of data exchange and exploration.

By sharing the data on consumption and flexibilities, it is possible to better manage the different networks and resources, as well as to make each of the stakeholders start to gain the interest of participating in the management of each of the networks.

Because it is a neutral platform, which will deal with a big volume of data and various users, it is important that we build a platform that will guarantee a safe, confidential, standardized communication and always be considering that all users must be authenticated and be given a private session.

The Market Hub platform will be built using the best practices and recommendations of SE. Hence, the architecture will be composed by an application server, which will handle requests, and produce the outputs for the users. The application server will rely on a local storage, in order to store data related to the users.

In order for the platform to meet what has been mentioned so far, it will be built in a modular, scalable and extensible way, supporting plug and play services. This must be a cross-platform application that can be migrated, if necessary, to another server and ensure that it follows the European guidelines and assure that the General Data Protection Regulation is followed (GDPR).

But before we begin to define the implementation of the platform, it is important to first become aware of the current state of the utilities, trying to find problems in these sectors that can be solved with this project. We try also to learn from other projects or initiatives from one of those sectors, techniques and knowledge that we can follow and adapt to this project.

On the other hand, it is also necessary that we know methodologies and processes of modelling and prototyping of Software Engineering so that when we start working on the problem we know when and how the techniques we should use.

In short, the objectives that we intend to achieve in this project are:

- Define the requirements for a generic platform that can be used in data exchange for utilities;
- Validation of methodologies and processes, in order to find the most appropriate and suitable for the design and implementation of this project;
- Design a generic and neutral data access platform that should be modular, scalable, secure, ensures data protection policies and allows interaction between various types of users;
- Implement the designed approach as a proof of concept.

1.3 Document Organization

This document starts with an **Introduction** chapter where a description, motivation and objectives that we would like to achieve with this project is presented. Then, in the **Context,** chapter 2, an overview of the current state of the utilities sector and some inherent projects is provided.

To support the chosen approach for this project, we include a **Processes and Methodologies, in** chapter 3 which discusses processes and methodologies of SE with a comparison between then, as well as the methodology proposal for this project.

In the **Generic Market Hub,** chapter 4, the purpose and requirements of the platform are explained. The process and decisions taken throughout the implementation phase are described at **Design of/a Generic Market Hub,** chapter 5.

The implementation phase is exposed in the **Development,** chapter 6. This chapter contains the technics and the technologies chosen. At the end of this chapter, a case of study is presented.

Finally, a **Conclusion**, chapter 7, which contains a critical analysis of all the work carried out as well as proposals for future work.

2. Context

We live in a society were some public services, utilities, are extremely consumed, such as electricity, gas, water, internet, communications, cable tv, among others. The consumption of these services has been increasing, which has caused many companies from the various sectors to start looking at all the data resulting from this service and to use it to improve the service quality, always with the aim of increasing the profits of the company.

However, much of this data is retained on the side of the service provider, passing just a few information to the customer side, normally just the one we see in the invoices or in the customer area.

This approach worked well for a while, as people consumed, paid, and a lot of them were neither interested at all in knowing how the whole business was processed, nor why sometimes the price of a certain service increased.

However, that changed, and people are more and more educated, possessing a generic knowledge about how the whole service rendering process unfolded, and became interested in participating more actively in these business models.

From this, arose the need to create a platform, a Generic Market Hub, allowing the exchange of data independently of the sector and making it possible for all the parties that compose the business model of the sector to consult the inherent data and to participate in the business, translating into an optimization and development of the business model itself.

For the Generic Market Hub to be a plus, we need to look at how it will be implemented. Taking advantage of this new concept of cloud and the possibility of being accessible anywhere, be a flexible system and able to recover in case of failure. We have, for example, data storage systems in the cloud, some more robust and safe than others (amazon drive, google drive, one drive, megacloud), but which have proven to be reliable solutions that bring characteristics such as good usability, portability and efficiency.

Hereupon and taking into account that several leading companies in the IT sector, like Google, Microsoft and Facebook, are heavily using cloud computing to develop and make their services available, we will opt for a solution that runs and stores data in the cloud, enabling easy, fast and controlled access to the data.

After researching leading companies in cloud computing, names such as SAP and Oracle have stood out, not only for the quality of the business management products they have developed over the years but also for the maturity of their cloud computing services[5].

Another care that we must have in the construction and subsequent development of the Market Hub is what techniques and methodologies to use, so that at the end the resulting product meets the requirements and fulfils the functions for which it was designed. It is therefore necessary to design models and diagrams so that we can understand the problem as a whole, validate each requirement, perceive each component that constitutes each part of the Market Hub, which will serve as a guide in the development phase and, in the case of the Class Diagram, it will be possible to extract code from it.

This concern about creating a neutral platform of data access in order to facilitate and liberalize access to data, has already been gaining form in some utilities, particularly in the public sectors that involve electric energy. For instance, the European Commission has been funding projects aimed at liberating the data of the utilities sectors.

Smart energy grids are enabling electricity producers and consumers to effectively change their electricity generation and usage in ways that can contribute to the system optimization while also benefiting themselves and helping to reduce environmental pollution[4].

During the past years, both the European Commission and the United States Government have been contributing and promoting a change in the liberalization of the data inherent to the utilities sector as well as the participation in the management of these sectors, which could contribute to better management of smart energy grid and in some cases the birth of new business models.

This revolution has been taking shape in Europe and the USA, as a result of the projects that have been carried out in some utilities sector, manly in the energy sector. In Europe, the projects Upgrid[2] and Flexiciency[1] stand out, and in the USA, the White House created the Green Button Initiative[3].

All those these have contributed positively to a decentralization of energy grid management, with the possibility now of all entities contributing to it. There was better management, consumption data, which until then had been restricted to consumers, could now be consulted [6].

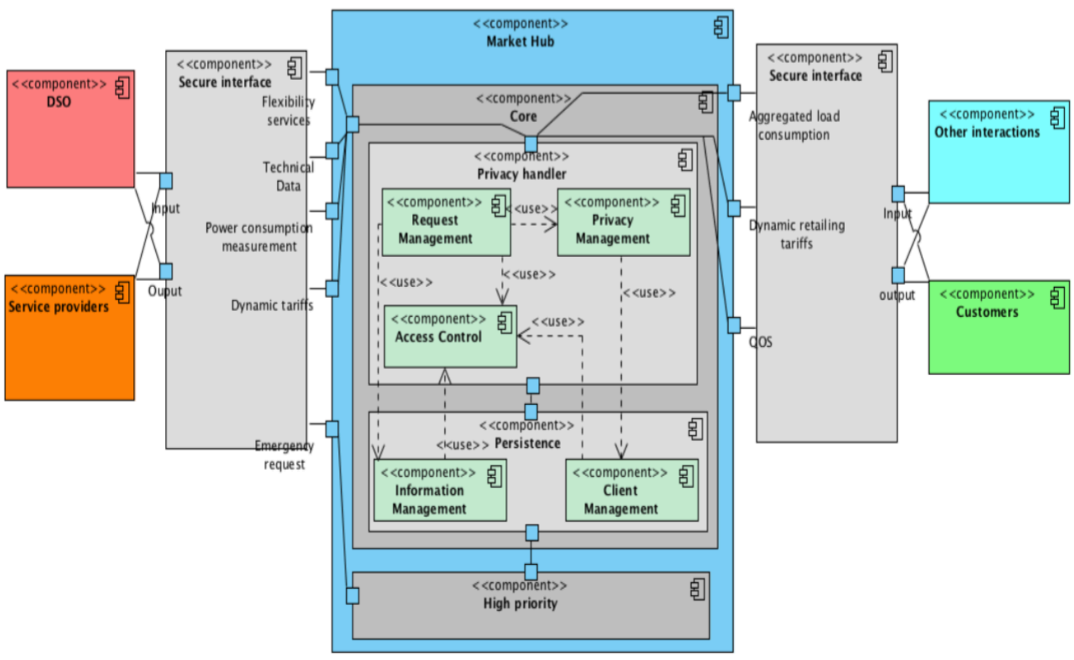
Next, follows a description of the three referred projects, in order to perceive the contribution, they have made to the liberation of the energy network.

2.1. Upgrid

Upgrid (ref. 646.531) is a European Project that started in the beginning of 2015 under the H2020 Program and it was developed by a European consortium, composed of 19 partners from 7 European Countries: Spain, Portugal, Poland, Sweden, United Kingdom, France and Norway.

With the objective of improving the monitoring and controllability of low voltage (LV) and medium voltage (MV) grids, making it possible to predict network energy peaks and problems associated with the large-scale integration of DER (Distributed Energy Resources), leaving the end users closer to the system operation and planning [2].

From the Upgrid project, we can extract the architecture design of a platform data exchange and service delivery, oriented to the energy sector, with several types of users that interact with the platform.



<<component>> DSO
<<component>> Service providers
Input
Ouput
<<component>> Secure interface
Flexibility services
Technical Data
Power consumption measurement
Dynamic tariffs
Emergency request
<<component>> Market Hub
<<component>> Core
<<component>> Privacy handler
<<component>> Request Management
<<use>>
<<component>> Privacy Management
<<use>>
<<use>>
<<use>>
<<component>> Access Control
<<use>>
<<component>> Persistence
<<use>>
<<component>> Information Management
<<component>> Client Management
<<component>> High priority
<<component>> Secure interface
Aggregated load consumption
Dynamic retailing tariffs
QOS
Input
output
<<component>> Other interactions
<<component>> Customers

Figure 1: Upgrid Market Hub Architecture

The Figure 1, showcases the modular components that compose the platform and how these relate to each other, focusing on dependency relationships. In particular, it depicts how the platform as a whole can interact with other market agents.

Let's move now, for a brief explanation of each of the components that are part of the core of Market Hub.

ACCESS CONTROL

This module contains the rules and access policies to be applied to users accessing system objects or service authentication handles. It needs to maintain a registry of user roles along with the permissions and capabilities these entail along with maintaining the registry of which roles are assigned to each user.

CLIENT MANAGEMENT

This module maintains a registry of client identifiers, as well as a mapping between each client identifier and the identifier of the contracted retailer. This module uses the Access Control module to ensure changes to the registry are only made by authorized users.

PRIVACY MANAGEMENT

This module is responsible for filtering response data to comply with rules and data privacy policies. In particular, it uses the Client Management module to ensure each retailer is only privy to its clients' data.

REQUEST MANAGEMENT

This module handles all operations pertaining to requests from the actors, namely DSO, Customers and Service Providers. It uses the Access Control module to verify that the user initiating a request has the required permissions as well as the Privacy Management module to ensure requests are forwarded to the appropriate recipients and that response data complies with the system's privacy policies.

INFORMATION MANAGEMENT

This module handles general information, i.e., which is not parameterized by client identifiers. View tariffs and announced network constraints (View constraints). Also, this module uses the Request Management module to obtain consumption profiles from the DSO.

Hereupon, we are dealing with component-based architecture which is one of the classic methodologies of software engineering.

Through the properties of the components, it is guaranteed that they can be reused or exchanged for other similar components and independent from each other. By choosing a component-based architecture is guaranteed that the code will be modulate and easy to maintain, easy in maintenance, because there are no dependencies, and room for evolution. [5 ,6]

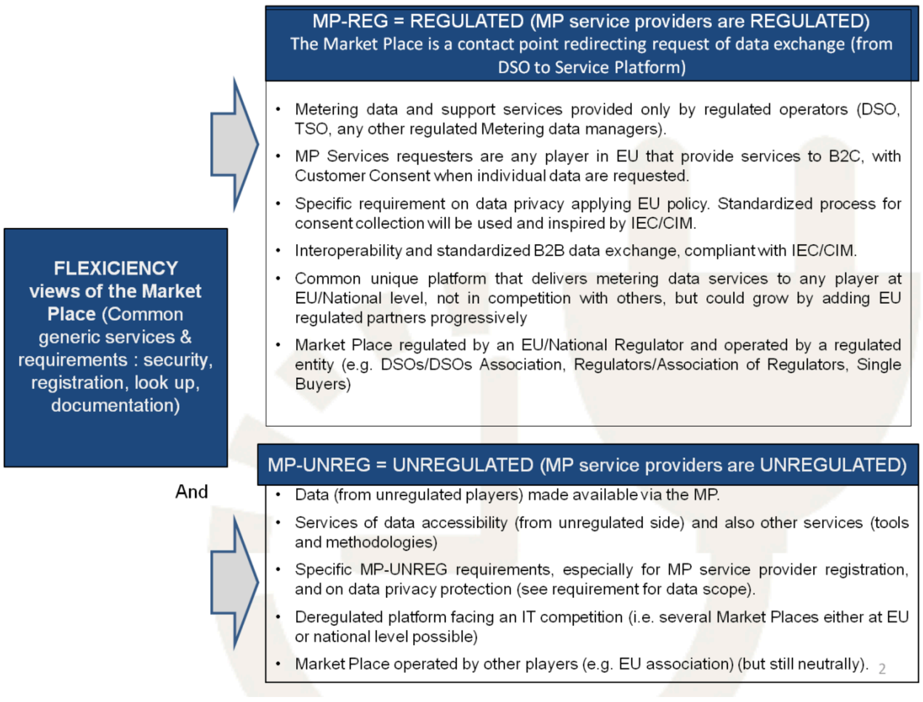
2.2. Flexiciency

Flexiciency is a European Project (ref. 646482) which has started in the in the beginning of 2015 and expected to end in January 2019. The project is promoted by a consortium of four major Distribution System Operators (DSOs) from Italy, France, Spain and Sweden [1].

The aim of the project is a creation of European Market Place (EU MP) that facilitates the interaction among all the electricity stakeholders. These interactions will be made through the provision of services - reduction in electricity consumption, load management, reduction in peak time consumption, potential consumer flexibility – and accessibility of metering data, close to real time, made available by the DSOs [1] .

Something that we can withhold from this project is that its architecture is being built to be flexible and modular, in order to keep up with the changes that may occur in the energy sector.

This flexibility is given to the system to be better leveraged after the project for further exploitation needs. These evolutions are likely to have different orientations for 2 major actor sets: regulated and unregulated. Therefore, the whole system must be structured into regulated and unregulated independent environments, building two modular configurations, according to their business responsibilities and implementation needs.



MP-REG = REGULATED (MP service providers are REGULATED)
The Market Place is a contact point redirecting request of data exchange (from DSO to Service Platform)

- Metering data and support services provided only by regulated operators (DSO, TSO, any other regulated Metering data managers).
- MP Services requesters are any player in EU that provide services to B2C, with Customer Consent when individual data are requested.
- Specific requirement on data privacy applying EU policy. Standardized process for consent collection will be used and inspired by IEC/CIM.
- Interoperability and standardized B2B data exchange, compliant with IEC/CIM.
- Common unique platform that delivers metering data services to any player at EU/National level, not in competition with others, but could grow by adding EU regulated partners progressively
- Market Place regulated by an EU/National Regulator and operated by a regulated entity (e.g. DSOs/DSOs Association, Regulators/Association of Regulators, Single Buyers)

FLEXICIENCY views of the Market Place (Common generic services & requirements : security, registration, look up, documentation)

And

MP-UNREG = UNREGULATED (MP service providers are UNREGULATED)

- Data (from unregulated players) made available via the MP.
- Services of data accessibility (from unregulated side) and also other services (tools and methodologies)
- Specific MP-UNREG requirements, especially for MP service provider registration, and on data privacy protection (see requirement for data scope).
- Deregulated platform facing an IT competition (i.e. several Market Places either at EU or national level possible)
- Market Place operated by other players (e.g. EU association) (but still neutrally). 2

Figure 2: Regulated and Unregulated environments in the Flexiciency Project (taken from [1])

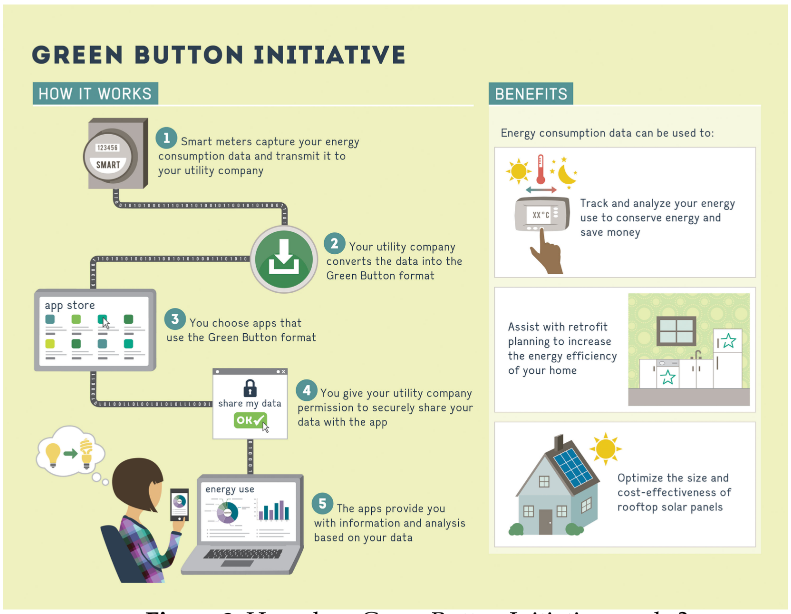
This system of environments, as specified in Figure 2, allows the existence of common utilities and services, several levels of security, as well as different types of procedures, namely related to data processing and authorizations, in both environments.

While service providers are separated in 2 different environments, any B2B service requester has a unique access to Flexiciency platform and could get different services from both regulated and unregulated sides, as if it was in the same platform.

2.1. Green Button Initiative

The Green Button Initiative is a White House initiative which aims to provide consumers with their consumption data, in a digital and readable format, so that later they can analyze their consumption and take measures in what it takes to their energetic consumptions.

The Green Button Initiative was launched in January 2012 and has a total of 50 utilities and electricity suppliers signed on to the initiative. In total, these commitments ensure that over 60 million homes and businesses will be able to securely access their own energy information in a standard and readable format [3].


GREEN BUTTON INITIATIVE
HOW IT WORKS
1 Smart meters capture your energy consumption data and transmit it to your utility company
2 Your utility company converts the data into the Green Button format
3 You choose apps that use the Green Button format
4 You give your utility company permission to securely share your data with the app
5 The apps provide you with information and analysis based on your data
app store
share my data
OK
energy use
SMART
BENEFITS
Energy consumption data can be used to:
Track and analyze your energy use to conserve energy and save money
Assist with retrofit planning to increase the energy efficiency of your home
Optimize the size and cost-effectiveness of rooftop solar panels
Figure 3: How does Green Button Initiative works?

From the beginning the Green Button Initiative took special care of how it handles the data of each consumer, being that the data does not contain Personal Identifiable Information (PII), something that in Europe has always been a concern. A proof of that concern is the relatively recent GDPR, which aims at the protection and the way personal data are used.

This topic of data protection and how it should be handled is something that we should take a consideration since we have seen a problem of the same nature, personal data.

When dealing with personal data, attention must be paid to who owns the data, in this case of the Green Button Initiative project, who is responsible for the consumer's. To answer this question, the Green Button Initiative attempt to get consumers access to their energy usage easily and conveniently. This information is currently in the hands of the utility.

The motivation behind the Green Button Initiative to data privacy, is that with access to energy usage data, consumers will be able to better understand their usage and hence make better decisions.

For this scenario to happen, it was necessary to implement a system of authorizations in which each consumer managed the data by assigning authorizations to each one of them. With these authorizations, consumers were able to decide how and to whom to share the data as well as having knowledge to whom they were shared and for what (see Figure 3, step 4).

3. Methodologies and Processes

Before proceeding to a detailed explanation of the problem and the selected approaches to try to solve it, let's get acquainted with certain knowledge about processes, methodologies and good practices of software development.

In the early days of software development, was the programming that reigned and usually used as a sequence of instructions with the aim of solving a problem. These problems, most of the time, were solved by a single person, who, by knowing the nature of the problem, made the implementation/solution of the problem, having it a personal touch, thus varying from individual to individual.

Over time, programming enthusiasts and computer users began to grow and with it came the separation between end users and developers. Users felt the need to pass on what they wanted, and developers would have to pick up that information and convert it on to code. Software began to become increasingly complex and communication was increasingly difficult, it was necessary to find a method or a set of methods that would ensure that the software would be produced according to the requirements of the end user, that the development was carried out in the stipulated time and he final product complied with the requirements initially proposed. There was then a need to create an area to study this problem and to find a solution or set of solutions to solve it [9].

Since the creation of the term Software Engineering in the 1960s and due to the extreme demand in software production and the risk that software production presents, this area of computing found itself obliged to improve and create new methods and processes for the creation of reliable, secure, quality software that fulfilled its objectives.

According to the Standish CHAOS Summary Report 2016[10], in 2015 only 29% were successful, of which 29% were mostly small projects. In this report, we can also find what makes a project conducive to success. User Involvement, Executive Management Support and Clear Statement of Requirements are the three reasons that can dictate the success of the project [10].

With software becoming increasingly critical, needing to be delivered ever faster and costing huge failures, it was necessary to create a compilation of knowledge that a software engineer should have in order to produce a correct, reliable, robust and user-friendly software. This compilation of knowledge was given the name of Software Engineering Body of Knowledge (SWEBOK) and is divided into the following areas of knowledge:

- Software requirements

- Software design
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering tools and methods
- Software quality

Let´s now have a look into particular area of knowledge, Software engineering process, and get to know better about the concepts of software processes and software process models.

3.1. Software processes

A software process is a set of related activities that lead to the production of a software product. From the application of the processes, we can create software from scratch or so by extending and modifying an existing system or by configuring and integrating off-the-shelf software or system components [11].

There are many different software processes, like Waterfall[11], Agile[12], RUP[11], but they should all include four activities that are considered fundamental:

- *Software specification* – where all the functionalities of the software and constraints on its operation should be defined.
- *Software design and implementation* – where the software is produced.
- *Software validation* – after implementation phase, the software must be validated to ensure that it does what the customer wants.
- *Software evolution* – the software must be capable of change to meet customer needs.

Software processes are categorized either as plan-driven or agile process. Plan-driven processes are processes where all processes activities are planned in advance and a comparison is made with this plan. In Agile processes, planning it´s incremental and it is easier to change the process to reflect changing requirements [11].

Although there is no ideal process, it is important that we choose an approach that is suitable for our needs. In order to choose the best model for the development of our software and consequently, the success of this thesis, we will come to a brief explanation of the most used Software process models, concluding with a brief comparison between them.

3.2 Software process model

A software process model is a simplified representation of a software process. Each model represents a process from a particular perspective, and thus provides only partial information about that processes.

These models are not strict recipes that we should use, but rather, abstractions of the process that can be used to explain different approaches to software development. The process models that we will cover here are the: *Waterfall model, RUP, Agile and Spiral development.*

3.2.1 Waterfall model

The Waterfall model is a sequential development approach, in which development is seen as waterfall because of flowing steadily downwards through several phases, typically: *Requirements Definitions, System and Software Design, Implementation and Unit Testing, Integration and System Testing, Operation and Maintenance.*

The waterfall model is an example of a plan-driven process – first, developers must plan and schedule all process activities and only then, we can start work on them. This model is illustrated in Figure 4.

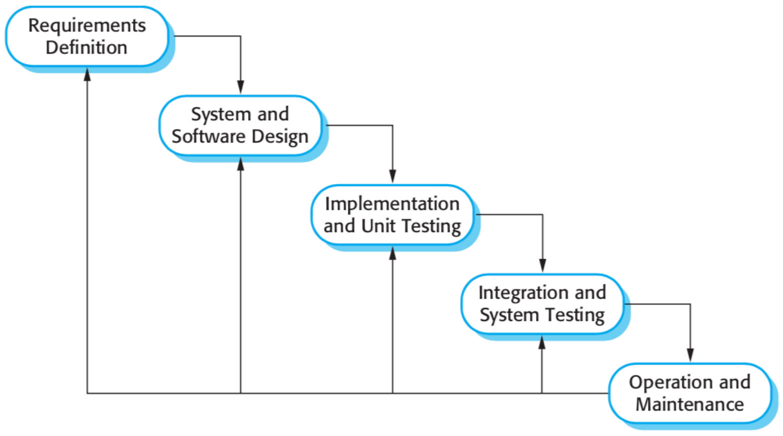

Requirements Definition
System and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure 4: The waterfall model (taken from [11])

The result of each phase is one or more documents that must be approved, that is, we can only proceed to the next phase if the antecedent phase has been duly approved and documented. In practice, these stages overlap and feed information to each other.

The waterfall model is consistent with other engineering process models. This makes the process visible, so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages [9].

3.2.2 The Spiral model

The Spiral model is a risk-driven process framework. Based on the unique risk patterns of the project, the Spiral model guides a team to adopt elements of one or more process models such as incremental or waterfall.

In Figure 5, we can see that software process is represented as a spiral, rather than a sequence of activities, without neglecting one activity to another. Each loop in the spiral model represents a phase of the software process. [5, 7]

The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

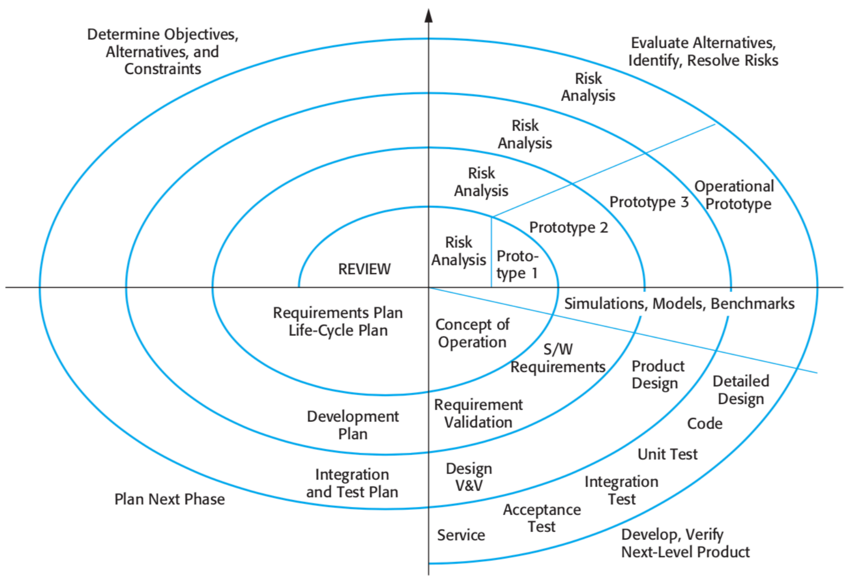


Figure 5: Boehm's spiral model of the software process (taken from [11])

3.2.3 Rational Unified Process – RUP

The RUP is an iterative software development process framework which provides a disciplined approach to assigning tasks and responsibilities to every developer. Its main goal is to ensure the production of high-quality software that meets the needs of its end-users. The RUP is a guide for how to effectively use the Unified Modeling Language (UML) [13].

Through the provision of 6 key practices, *Develop software iteratively, Manage requirements, Use component-based architectures, Visually model software, Verify software quality* and *Control changes to software,* each team member should be able to produce a piece of software according to the requirements of its end-users.

Figure 6 depicts the process described in two dimensions, organization along content and iterations / phases.

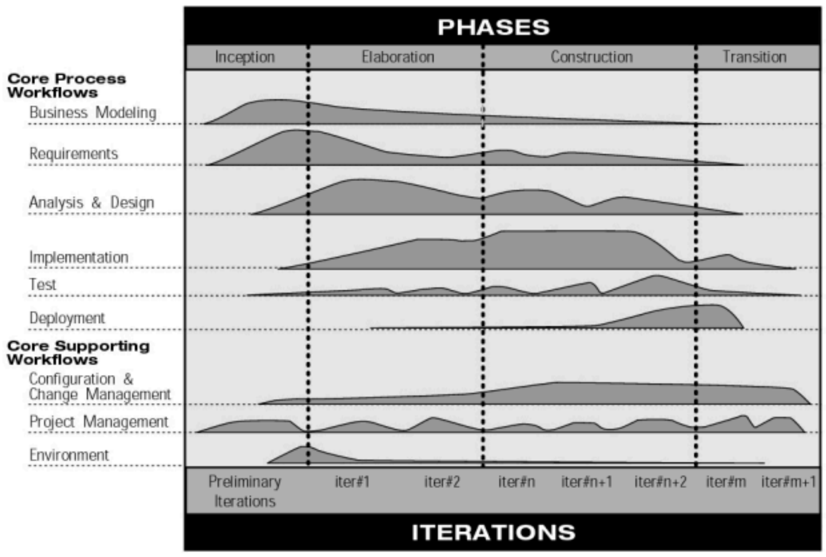


Figure 6: RUP Model graph (taken from [13])

The software lifecycle is broken into cycles, each cycle working on a new iteration of the product development.

3.2.4 Agile development

Agile relies on an incremental approach for software specification, development and delivery. It is suited for an application development where the system requirements usually change rapidly during the development process.

The goal is delivering working software quickly to customers, who can then propose new and changed requirements to be included in future iterations of the process.

The philosophy behind agile methods is reflected in the Agile manifesto [12] that was agreed on by many of the leading developers of these methods. This manifesto states:

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools;*
- *Working software over comprehensive documentation;*
- *Customer collaboration over contract negotiation;*
- *Responding to change over following a plan;*

That is, while there is value in the items on the right, we value the items on the left more".

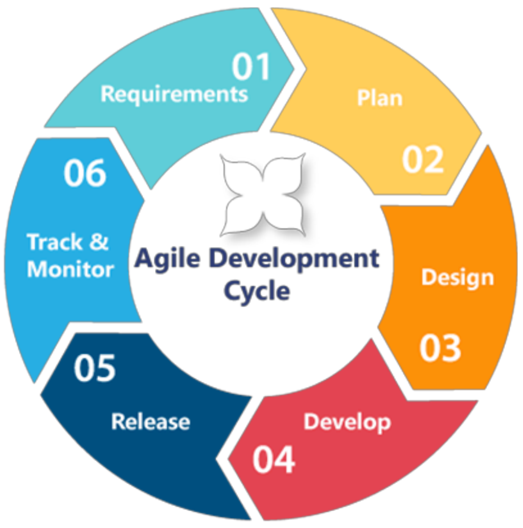


Figure 7: Agile development cycles (taken from http://www.7eleventech.com/methodology/index.html)

Agile processes are driven by customers descriptions of what is required, recognizes that plans are short-lived, develops software iteratively with a heavy emphasis on construction activities, delivers multiple software increments at the end of each cycle (see Figure 7) and adapts as changes occur.

3.2 Model comparison

Now that we've covered some of the most referenced software development processes, it's time to make a comparison between them, highlighting post and cons.

The purpose of this comparison is to attempt to establish a rationalization phase for the following steps. This was necessary in order to sustain the approach choice which will be followed to carry out this project.

	Waterfall[14]
PROS	• The client knows what to expect. They'll have an idea of the size, cost, and timeline for the project. • In the case of employee turnover, waterfall's strong documentation allows for controlled project impact.
CONS	• Once a step has been completed, developers are not allowed to go back to a previous stage and make changes. • If a requirement error is found, or a change needs to be made, it is necessary to re-evaluate requirements. • The whole product is only tested at the end. If bugs are written early, but discovered late, their existence may have affected how other code was written.

Table 1: Waterfall Pros/Cons

	Spiral model
PROS	• This model is good for large and critical projects. • The high amount of risk analysis hence, avoidance of possible risk is certainly reduced. • In the spiral model, additional functionality can be added at a later phase.

CONS	• It is not appropriate for low-risk projects. • The success of the entire project is dependent on the risk analysis phase thus, failure in this phase may damage entire project. • The big risk of this methodology is that it may continue indefinitely and never finish.

Table 2: Spiral Pros/Cons

	RUP[13]
PROS	• Enables managers to examine the relationship between tasks and external pressures. This creates more realistic projects. • The development time required is less due to reuse of components. • Removes project risks linked with client evolving needs.
CONS	• Needs excessively expert software developer. • Challenging to put into practice. • The development process in this methodology is very complex and not exactly organized.

Table 3: RUP Pros/Cons

	Agile[14]
PROS	• Allows for changes to be made after the initial planning. • The testing at the end of each sprint ensures that the bugs are caught and taken care of in the development cycle

	• At the end of each sprint, project priorities are evaluated. This allows clients to add their feedback so that they ultimately get the product they desire.
CONS	• The project can become an endless loop of code sprints. • The project is likely to come in late and over budget. • As the initial project doesn't have a definitive plan, the final product can be grossly different than what was initially intended.

Table 4: Agile Pros/Cons

	Good Documentation	Easy to apply / use	Change of requirements	Aim for deliver on time
Waterfall	+ + +	+ +	− − −	+ +
Spiral	+	+	−	−
RUP	+ + +	−	+ +	+ +
Agile	−	− −	+ +	− − −

+ - positively characterized by; − - negatively characterized by

Table 5: Comparison of methodologies

From the comparative table, it is possible to infer that none of the methodologies compared is completely negative nor completely positive. We cannot discard any of the methodologies from the beginning. First, it is necessary to know the project and its requirements. Only after this first perception, we can decide which methodology to follow.

This analysis of the methodologies is susceptible to subjectivity because it depends on the person who applies them and his/her knowledge on the subject. In this context, this person should have knowledge of software development methodologies and UML.

First, we must consider the nature and size of the project, check if there are aggregate risks to the project, the duration of the project and if the customer knows what he wants, which is never the case.

Aware that the project has a considerable dimension, which involves some types of users, requirements that can be refined until the final version and deadlines that need to be met, there are certain aspects of the methodologies that we can put aside and others that could be a plus:

- In the Waterfall methodology, we can ignore the fact of having a general planning, due to changes that may be made. On the other hand, it can be important to have a preview of the final product.
- In the Spiral methodology, the fact that a risk analysis is done to the project is always welcome, another positive aspect for the project is the possibility of being flexible when it comes to changes. A negative aspect of this methodology, and that can put at risk the success of this dissertation, is that control of project management can be lost, jeopardizing the delivery deadlines that need to be met.
- In RUP, we could use UML so that in the development phase we can automatically extract much of the code.
- In Agile methodology, the fact that changes can be made at any stage of the project becomes a plus since the project may have to undergo such changes. However, the part of project management in terms of deadlines is an aspect that stands out negatively, and in the case of this dissertation, it is important to pay close attention to this

3.3 Proposed Methodology

After seeing how important it is to choose a methodology to ensure the success of a project, always considering the objectives, deadlines, requirements, among others, we will explain the methodologies followed and the reason for those choice.

The methodology followed in the project of this dissertation, in the entire process of requirements analysis and platform modeling, is based on the RUP methodology (Figure 8) with iterations.

In this first phase, we start by analyzing all utilities and try to understand how they work, find aspects and procedures which extend to all. Then, the UML was used to model the entire generic system in order to understand better all requirements of the platform and, in the implementation phase, this modelling was used as a guide and from some diagrams we were able to extract some code.

On the other hand, we knew that the requirements would take time until they were defined and to safeguard from that aspect it became crucial to choose a methodology that managed to deal with these constant changes.

It became necessary to follow some procedures of the Agile methodology, in order to deal with the subtle changes that might occur during the course of the project. Also, in the implementation phase, it was decided at the end of each cycle/sprint to test what was implemented in order to detect errors and correct them in a timely manner.

In Figure 8, you can see, step by step, the entire process, from the requirement analysis phase, through modeling until completion of implementation.

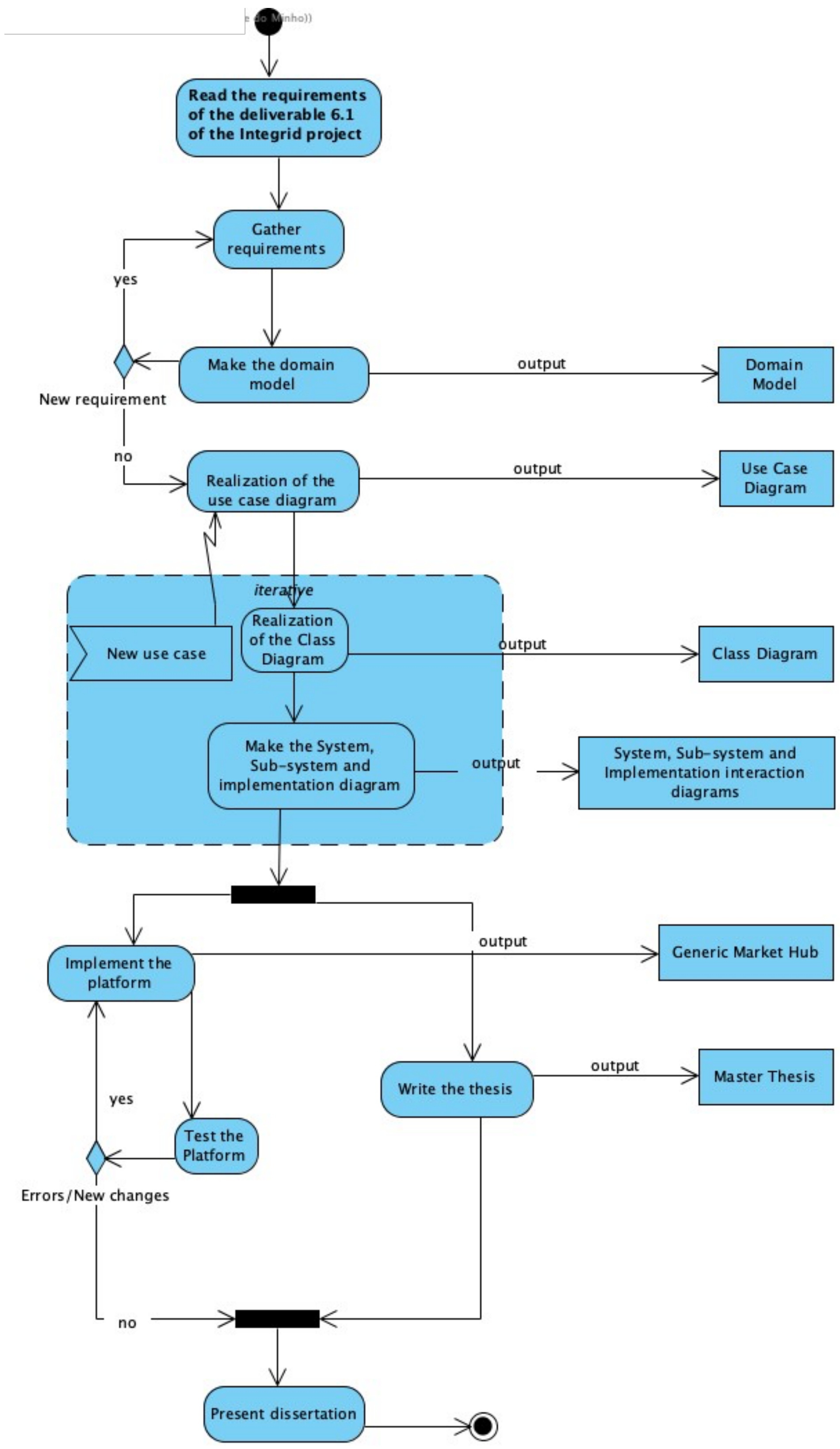


Figure 8: Proposed methodology

The first phase of the project and considering what is proposed by the RUP methodology, we start with the requirement collection and analysis phase, passing to business modeling as the requirements appear by reading the deliverable D1.2 Read Use Cases and Requirements from the Integrid Project [10] and to try to gather the first requirements.

Through interactions and new inputs, these requirements were worked out not only for being applied to the energy sector, but also to cover all utilities, such as water supply, gas, communications, internet and cable tv.

With this, and with the requirements being more or less defined, we started with the design of the domain model in order to be the "glossary of the project", making it possible for us start to decipher the terms used and represent the relations between these terms.

The domain model was not ready in its first iteration, as they were changes over time. Such was one of the main reasons why we have chosen RUP and Agile methodologies to guide us throughout the design, modeling and implementation processes of the platform. That is, it was necessary to choose a methodology or methodologies that did not compromise the success of the project when we change the requirements.

With the entities already defined, we began to outline the diagram of use cases in order to define which iterations with the system is that each entity/user should have. This phase and the following, class diagram and system diagram, subsystem and implementation, needed more time and the iterations between them were some. The search for use cases that covered all the utilities mentioned was difficult and made it necessary for us to redo all the diagrams as they were withdrawn or added new use cases.

Lastly, we have the implementation and testing phases, as well as the writing of this thesis. For the implementation phase, it was applied an Agile methodology, where the entire implementation was based on a succession of code sprints. At the end of each sprint, tests were performed and in case of errors, these were solved.

4. The Generic Market Hub

The Generic Market Hub is a web-based platform with the purpose of speeding up and moderating the exchange of data among various types of users. Through this data exchange, it is intended that all the intervening parties can participate in the management of the network and contribute towards its evolution.

Through the application of good software design practices, it is intended to develop a generic platform, with the properties of being robust, modular, scalable and easily maintained.

In order to achieve a solution that will suit almost all utilities, we use as starting point the utilities such as electrical energy, water and gas. Through a research, we were able to perceive how each one worked, its business models, entities involved but, above all, the data that supported these sectors.

4.1 The Generic Domain Model

With this and through a process of iterations, we managed to get a domain model for the Generic Market Hub (Figure 9).

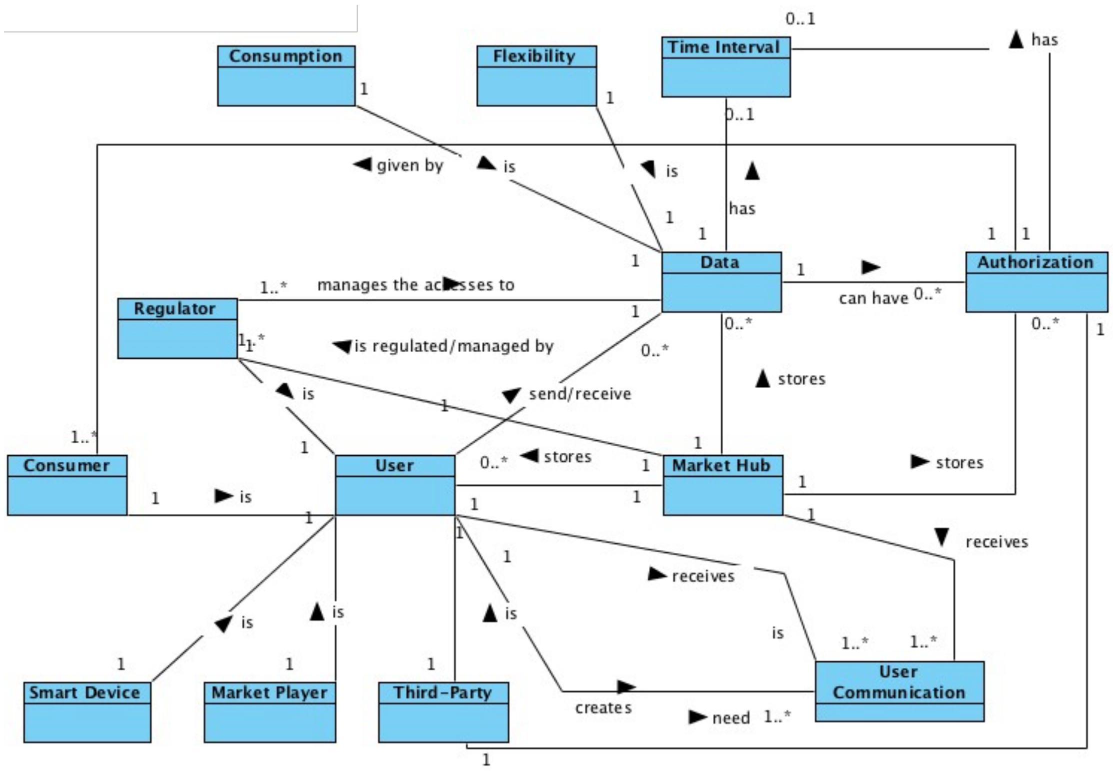


Figure 9: Domain Model for Generic Market Hub

Following, there is an explanation about all the logic behind the diagram exposed in Figure 9.

Market Hub has three types of Users (`User`). They can be a `Consumer`, a `Market Player`, `Regulator` or a `Third-Parties`, as shown in Figure 10.

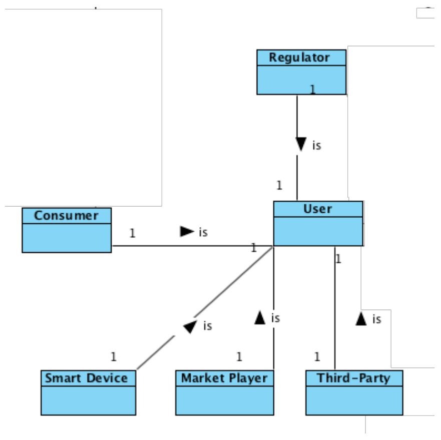


Figure 10: Types of Users

The `Consumer` represents the final customer/consumer. The `Market Player` represents an intermediary entity or entities within the specific application domain in which the architecture is instantiated, and that usually participates in the exchange and validation of the data.

Then, there is the `Smart Device`, an entity responsible for collect data and share it, or receive data and proceed according with the data received.

The whole ecosystem is managed by the `Regulators`, which are responsible for ensuring that all data exchanges are made in compliance with all safety rules, specified in the European Data Protection regulation of the domain.

The Third-Parties represent all the entities related to the exploitation of data, with the aim, for instance, of carrying out studies to develop new business models of the domain.

These entities can communicate (`User Communication`) either via `Market Hub` or choose a direct communication, as shown in Figure 11. Through these communications channels, they may require/share `Data`. The `Data` corresponds to relevant information to be shared between the entities that make up the **Market Hub** (e.g. energy flexibility, network utilization).

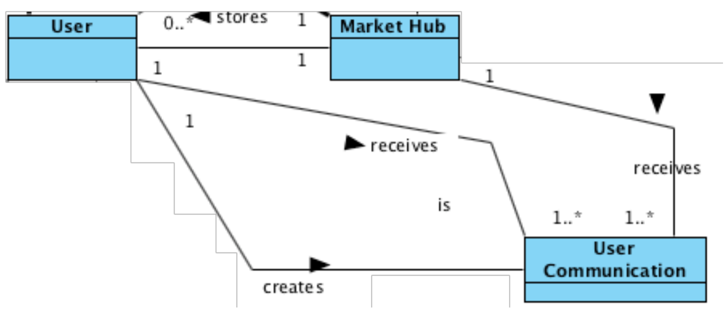


Figure 11: Communication on Generic Market Hub

This `Data`, that can be stored in the `Market Hub`, can have a `Time Interval` and an `Authorization`. The Authorization includes all the information about how and who might use the data, and might also have a `Time Interval`, required for its proper use. Time Interval specifies the Begin Date and the End Date.

The `Data` can be either related to `Consumption` or `Flexibility`. `Consumption` corresponds to data related to the utilization of a particular good by the customer, while `Flexibility` represents the amount of a good that a user may relinquish or need.

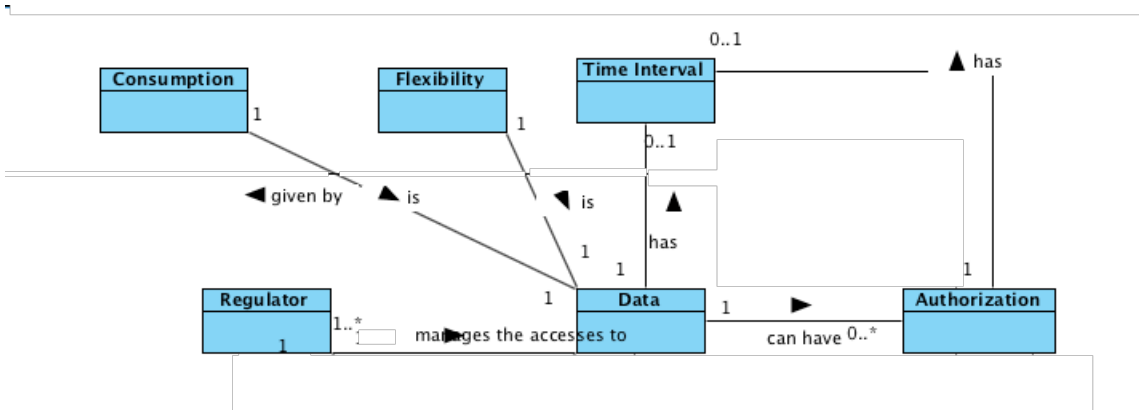


Figure 12: Data relationships

In order to comply with these standards, all third parties must have `Authorization` given by the data owner (`Consumer`). All authorizations management is in charge of the Consumers, the data owner's.

This detail of the **Authorization** was inspired from the **Green Button Initiative** project, discussed in chapter 2. The authorization also serves to ensure that the platform complies with the GDPR.

This was our vision on how the Generic Market should be designed, in order to fulfil the presented objectives and be able to be instantiated to any domain.

4.2 Refinement for concrete models

Following, lets analyze three domain diagrams, which represent three different utilities and compare them with the model for the Generic Market Hub. In this comparison, we highlight all the details that are common to all diagrams.

For constructing the diagrams, we had to abstract ourselves from certain aspects to make each diagram smaller, easier to perceive and focus only on actions involving the utility grid management data which would be supported by the Generic Market Hub.

With all this information gathered, we designed three different domain model diagrams.

Figure 12 shows all entities, roles, datatypes present in the **Water Market Domain**. Figure 13 depicts all entities, roles, datatypes present in the **Gas Market Domain** and, finally, Figure 14, shows all entities, roles, datatypes present in the **Electricity Market Domain**.

With this comparison, we intend to find the aspects that apply to all domains and compare them with the aspects represented in the Generic Market Hub.

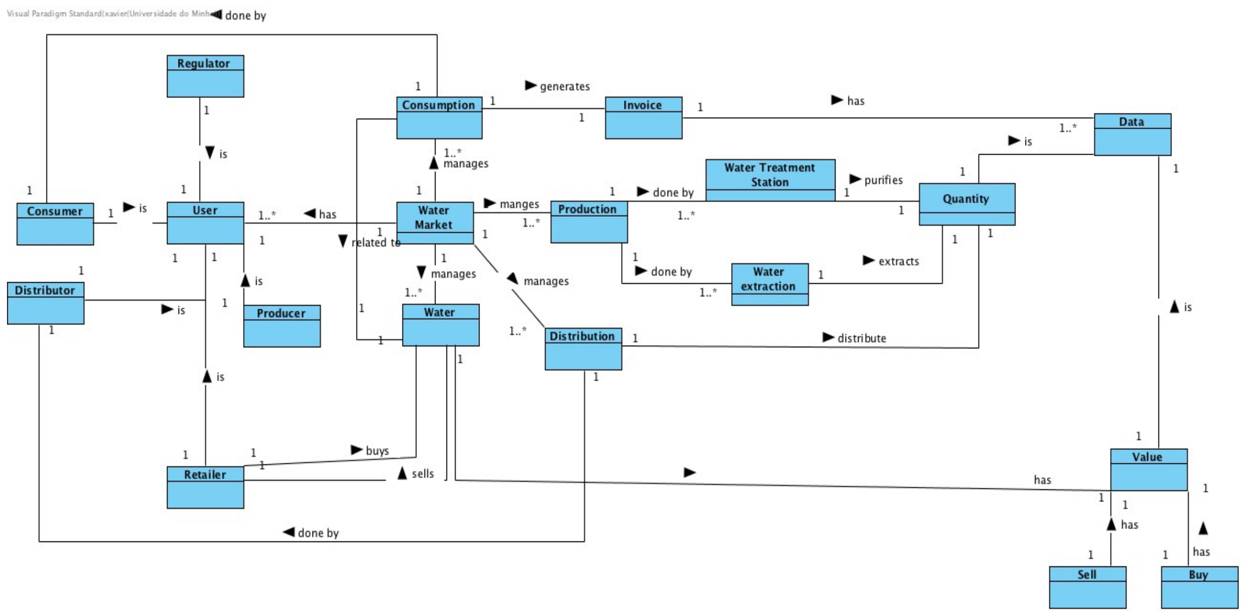


Figure 13: Domain Model of Water Market

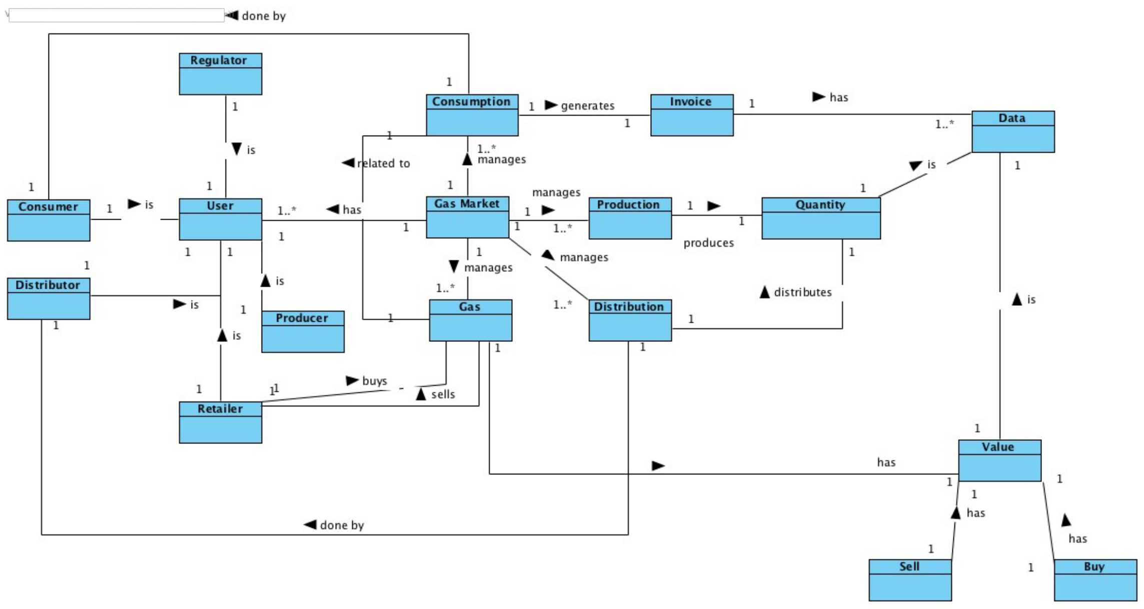


Figure 14: Domain Model of Gas Market

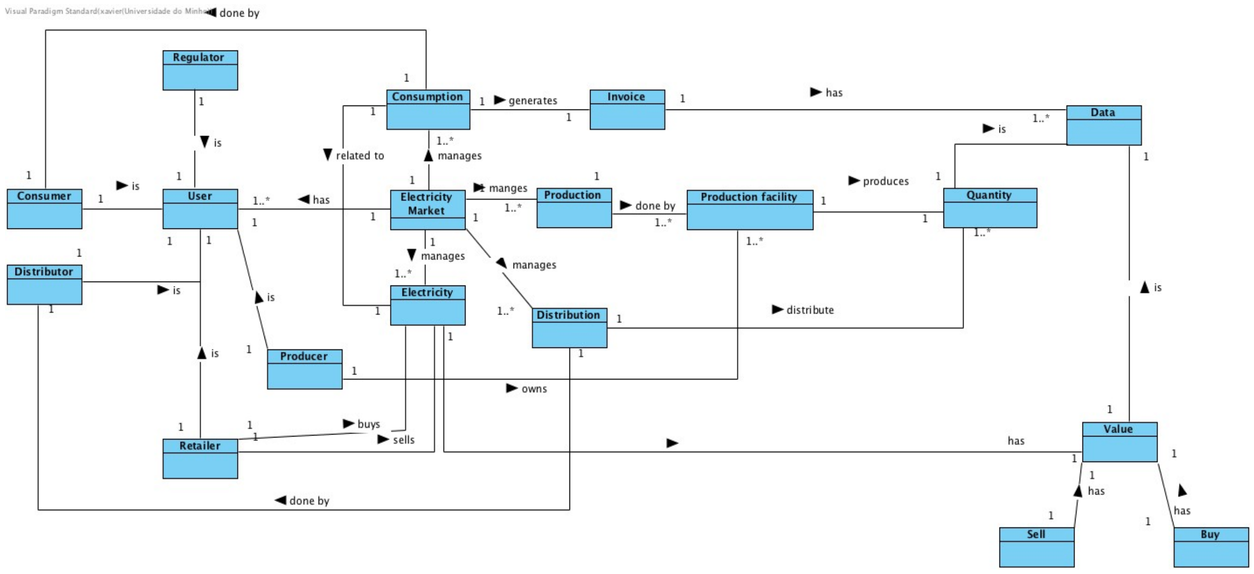


Figure 15: Domain Model of Electricity Market

These three diagrams (Figure 13, 14 and 15) show the entities/users that participate in each of the utilities, as well as the actions from which there are important data to be exchanged and extracted, which can be communicated via Generic Market Hub. We will now analyze them together, trying to collect aspects that are common to all.

Regulator, Consumer, Distributor, Retailer and Producer are types of Users (User) that constitute all utilities described. Each type of user has its functions as well as access to different types of data.

Comparing with the domain model of the Generic Market Hub (Figure 9), it is possible to find similar types of users such as Regulator, Consumer. Then, we have the Market Player and the Third-Party. Both types of Users are representing entities that compose and integrate the utility sector, usually with more specific names.

Entities such as Producer, Retailer and Distributor might be represented by the Market Player entity. On the other hand, Third Player can represent the types of users interested in using the data for studies and statistical purposes.

Actions such as Consumption, Production and Distribution, which are usually carried out by a type of User (e.g. Consumption -> done by -> Consumer) and related with a utility (e.g. Consumption -> related to -> Gas), are actions of which there is associated data, and which will be important to store or exchange.

An Invoice is generated from the Consumption. In that Invoice we have Data, such as Value of Buy, the amount consumed and the date range to which the invoice relates.

In these three models there are another type of interesting Data to be collected, such as the Quantity produced and distributed. Then and from Buy and Sell values, we can get Data regarding the process of buying and selling a utility, Gas, Electricity or Water, by the entity Retailer.

In these models (Figure 12, 13 and 14), the data access Authorization is not referred. Such is due to the fact that each domain has data access policies restricted to only 2 types of users, normally to Regulators and Distributors.

Therefore, to ensure that the liberalization of access to data by all users goes with the meeting standards and laws, it was necessary to include this detail in the Generic Market Hub.

4.3 Comparation with InteGrid's Project Market Hub

A specific instance of the Generic Market Hub model can be defined for the energy sector, specifically in the context of the InteGrid project.

InteGrid vision is to bridge the gap between citizens and technology/solution providers such as utilities, aggregators, manufacturers and all other agents providing energy services, hence expanding from DSOs distribution and access services to active market facilitation and system optimization services while ensuring sustainability, security and quality of supply.

InteGrid is developing a grid and market hub (gm-hub) that provides the integration link between the stakeholders of the project (e.g. service providers, service users) to build an integrated environment to enable demand response, smart grid functions and storage.

InteGrid will develop and test a **cloud-based** gm-hub solution to support the provision of services in a neutral standardized way between customer **relationship manager** or **neutral market facilitator** (primary roles of this central platform) and stakeholders like electricity retailers, transmission system operator (TSO), aggregators, group of consumers and energy services providers (e.g. Energy Services Company, data analytics companies).

The main objective is to **facilitate market access allowing new business models and services** while ensuring efficient and secure network operation as well as highest standards of data security.

The gm-hub **operates in a regulated domain**, thus all the embedded services are regulated and subjected to a suitable regulatory framework for data management and exchange. Furthermore, this central platform should be perceived as an **enabler of third-parties' services** that can emerge in the gm-hub ecosystem.

In Figure 16, depicts the domain model of the gm-hub of the InteGrid project. Next, we will try to understand and justify why our generic market hub is a viable solution for this problem.

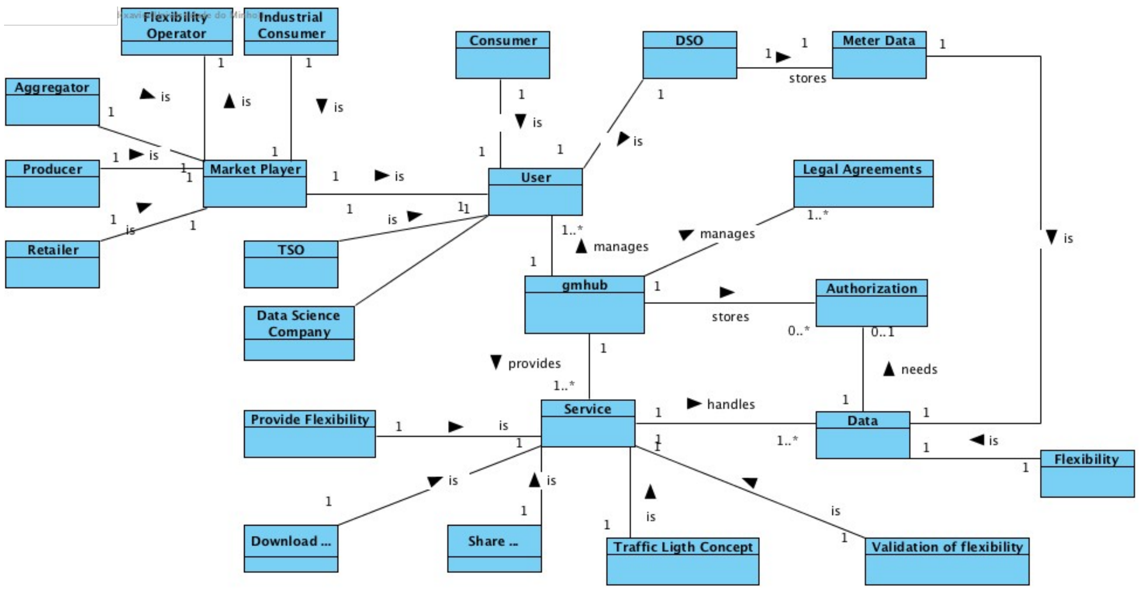


Figure 16: InteGrid's Domain Model

As in the domain diagrams of the energy, water and gas sectors, also in gm-hub is perceptible the existence of several users. These users will use gmhub as a proxy so that they can communicate through it and use the services provided.

Among these types of users, we have `Transmission Control Protocol` (TSO), `Consumer`, `DSO`, `Data Science Company`, `Market Player`. The `Market Player` can be divided into 5 types of user: `Retailer`, `Producer`, `Aggregator`, `Flexibility Operator` and `Industrial Consumer`.

In the context of this project, the DSO is in charge of the management of the platform as well as to moderate the access to consumption data of each of the consumers (**Meter Data**). Then there is the whole bureaucratic part of **Legal Agreements** that each user must agree in order to use gmhub.

gmhub will have as core features the provision of services like **Provide Flexibility**, **Download Meter Data**, **Share My Data**, **Traffic Light System**, **Validation of Flexibility**. In all these services there is a need for Data exchange, this data that can be Meter Data or **Flexibility**. To access some data, it may be necessary to have an **Authorization** to do so. Authorizations are managed by gmhub.

Figure 17 presents the use case diagram, which explains the core features of the gmhub.

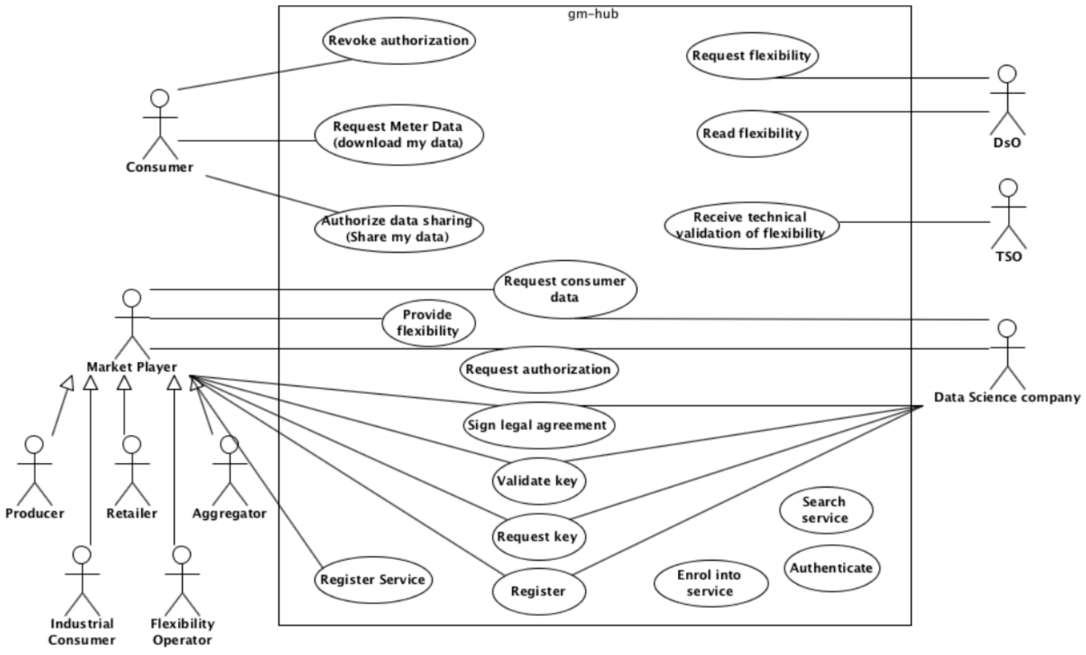


Figure 17: InteGrid's Use Case Diagram

This diagram is a concrete application of the use case diagram of the Generic Market Hub. In this case, the diagram was applied to a project that has as utility the electric power.

In addition to the typical use case of registration and authentication, we also have sector -specific use cases.

In Figure 17, we can see the distribution of use cases by each type of user. Following, there is a brief explanation of each use case.

- **Consumer**: can perform two main actions, **request meter data** and **authorize data sharing**. The first use case corresponds to download its own data in a readable format. The second one corresponds to authorize third parties to use their data.
  These two features correspond, respectively, to the green button initiative use cases **download my data** and **share my data**. Finally, the consumer can revoke read authorizations at any time.
- **Market player**: apart from the **registration**, this group of users can request consumption data or provide information of flexibility. In order to perform the former, the user should **request an authorization**, which the customer needs to allow.

In addition, the market player can **sign the legal agreements**, required to access the customer data. In the case of the gm-hub acting as an intermediary, users can also **request** and **validate** the keys to access user data.
Finally, the user can **advertise services** in the gm-hub, available to all stakeholders.

- **DSO:** this actor can perform two main use cases, **request flexibility** and **read flexibility**. The first one corresponds to request flexibility that the Transmission Control Protocol (TSO) has activated in order to perform a validation. The second use case corresponds to the DSO being able to read the flexibility that was activated after been evaluated and resubmitted.
- **TSO:** this actor has access to one use case, **receive technical validation of flexibility**. This use case allows to receive the flexibility that has been evaluated by the DSO and that will be activated.
- **Data Science Company:** this actor has the well-defined role of performing analytics. This group of users makes use of the already described use cases that are related to data access (request consumer data, request authorization and sign legal agreement, request key and validate key).

The **search service** and **enroll into service** use cases are common to all roles, therefore considered system use cases.

In the case of gmhub, the Generic Market Hub would be a solution since it can be used by several types of users and allows data exchange, even in a more generic context. There is also concern about how data is managed and used through the use of authorizations.

The Generic Market Hub by itself does not provide any type of services. However, in the context of gmhub, the provided services are nothing more than a sequence of data exchange between various types of users, action that can also be done in the Generic Market Hub via use cases such as send data, receive data and request data.

4.4 Use Case Diagram

Since the beginning of this chapter, we have been stressing that the core features from the Generic Market Hub are the exchange and sharing of data used to and from all types of users. These data will be used, most often, to promote the participation of all stakeholders in the management of the utility itself.

In Figure 18, we have the use cases of each of the types of platform users. In this diagram there are no reference to use cases such as authentication and registration of all types of users. This is because at the moment that information is redundant to the purpose of what will be the Generic Market Hub.

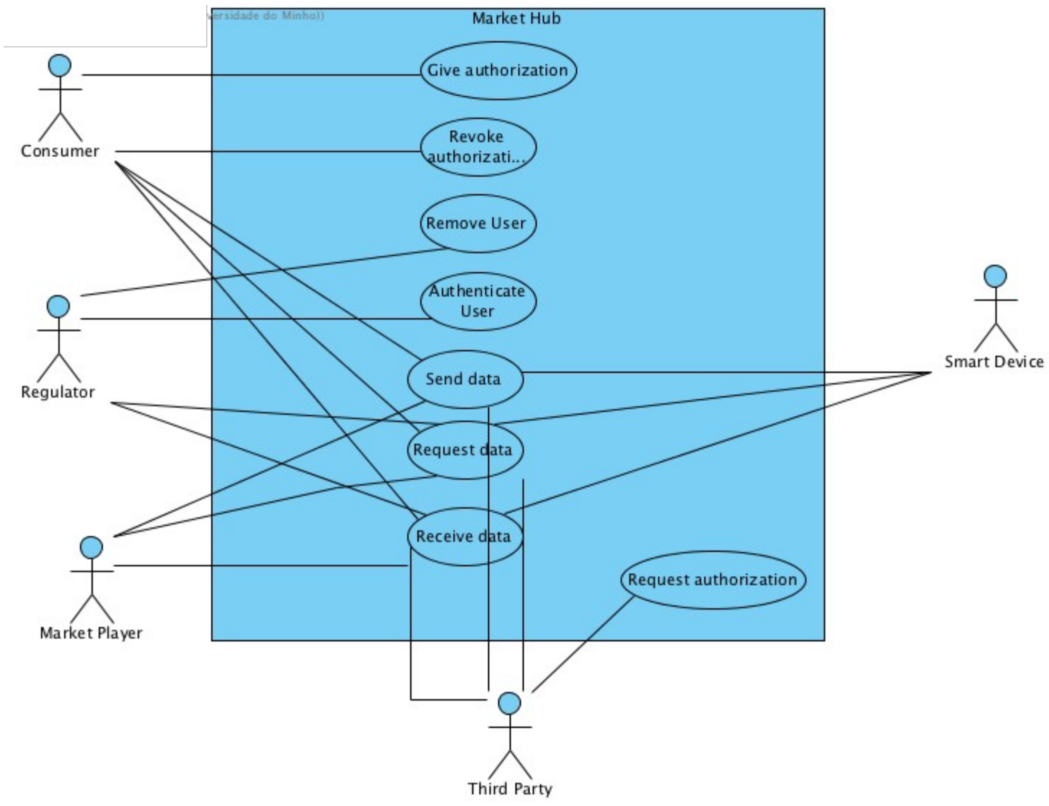


Figure 18: Use Case Diagram

Consumer:

- **Give Authorization** – This use case refers to the action of the consumer gives permission for his data to be used for purposes that himself can dictate.
- **Revoke Authorization** – This use case refers to the action of the consumer to revoke an authorization that he gave so that his data could be accessed.

Regulator:

- **Remove User** – This use case refers to the action of the Regulator, entity that manages the entire platform, to remove the user from the system. The next time that this user wants to access the platform he will not be able to do it anymore.

- **Authenticate User** – This use case refers to the action of a Regulator to accept the registration of a new user. After the execution of this use case, the user can proceed to use the platform.

Third-Party:

- **Request Authorization –** This use case refers to the action of a Third-Party user request authorization for use the Consumer's data. After receiving the authorization, the Third-Party is allowed to request data.

Common use cases:

- **Send data –** This use case refers to the action of any type of User send data to another user.
- **Request data-** This use case refers to the action of any type of User request data from another user. In the case of Third-Party and in some cases, to activate this use case he will need an authorization for request the data.
- **Receive Data -** This use case refers to the action of any type of User receive data from another user.

5. Design of the/a Generic Market Hub

In this chapter, the entire process and decisions taken throughout the implementation phase will be explained. With the use of UML diagrams and concepts of software architecture, we will present all the processes and artifacts that supported the design of the platform.

The whole design of the Generic Market was planned using good practices and methods of software design. Therefore, there was always a concern in design a platform that was scalable, modular, safe, easy maintenance and replication.

5.1 Package Diagram

In order to provide an overview of the developed system, at a high level of abstraction, we start by presenting a Package Diagram. According to the objectives proposed for this architecture, we resort to UML in order to create structural diagrams. By using these diagrams, we show the arrangement and organization of model elements which compose all system.

Figure 19 shows the overall package diagram. This diagram shows both structure and dependencies between sub-systems or modules, showing different views of a system.

The name assigned to each package and subpackage considers the analysis done to the problem domain, that is, it does not simply suggest a concrete domain but rather something generic. Hence, the given names generically represent the aggregate of entities and functionalities core of the generic platform. Then, we have the package Data Access Object (DAO), which will contain all objects that provide an abstract interface to some type of database or other persistence mechanism.

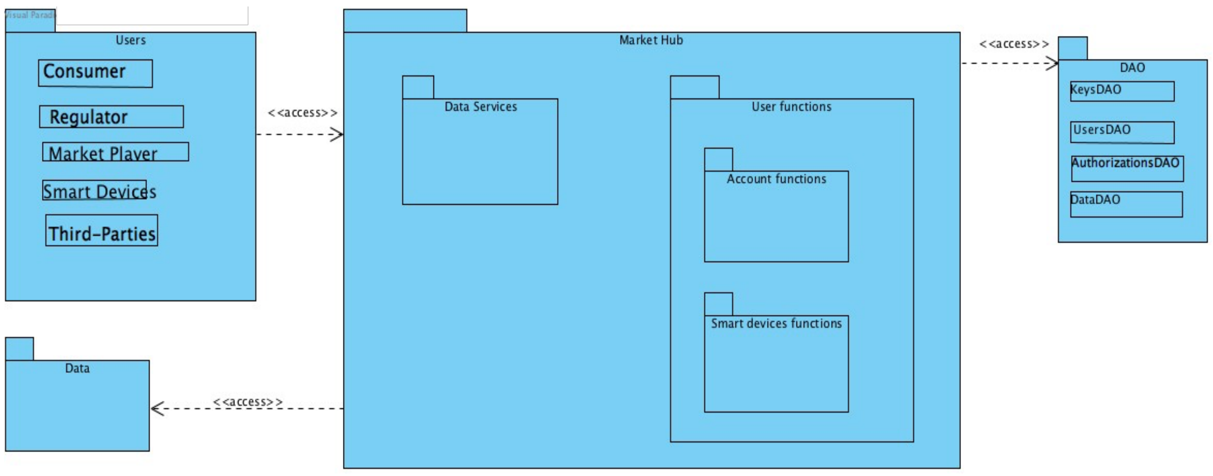


Figure 19: Generic Market Hub Package Diagram

The system was divided into four major packages, **User**, **Data**, **Market Hub**, **DAO**. Knowing that the platform will be used by several types of users, the package **User** will group all classes that belonged to this type and which represent a type of user. The **Data** package is related to all classes that compose this type. Then, there is the Market Hub package which is composed of two sub-packages, **Data Services** and **User Functions**.

All core functions, which will be related to the data communication, will be placed within this package, the Data Services package.

User functions, in order to have a better organization and perception, is subdivided into 2 sub packages, **Account functions** and **Smart Devices functions**. The first one contains all classes related to account management by human users. While the second concerns to all the management and configuration functions of all machine users.

Finally, we have the **DAO** package, which will contain all the classes that will manage all the persistence of the objects in the database.

5.2 Class Diagram

After this first structural view of how the platform will be built, the next step is to create a diagram capable of showing the types being modeled within the system. In Figure 20, the class diagram of the Generic Market Hub is presented. It is a specification class diagram because it specifies, at a low level, the entire implementation details.

In this diagram, we can see the classes that make up the system, how they interact and their distribution by the packages. In each class we have its attributes as well as the methods that characterize it.

In the User package, we used inheritance to achieve a better organization. We have a User class that will be the superclass and the remaining classes, except for Smart Devices, will be subclasses. These subclasses will inherit attributes and methods from the User superclass. On the other hand, and in order to distinguish each type of user, each of these types will have different variables.

Through this diagram we can also extract the data model. For this, we labeled as Entity Bean all the classes that will be persisted in Database. All this persistence will be managed by the classes contained in the DAO package.

All classes that will encapsulate the business logic that can be invoked programmatically by a client over local, remote, or web service client views were labeled as session beans.

Finally, and to make the diagram more complete, we added a new package, Servlets, which contains all classes that will make the respective classes accessible via browser, with the respective methods **doGet()**, **doPost()**, **doDelete()** and **doUpdate()**.

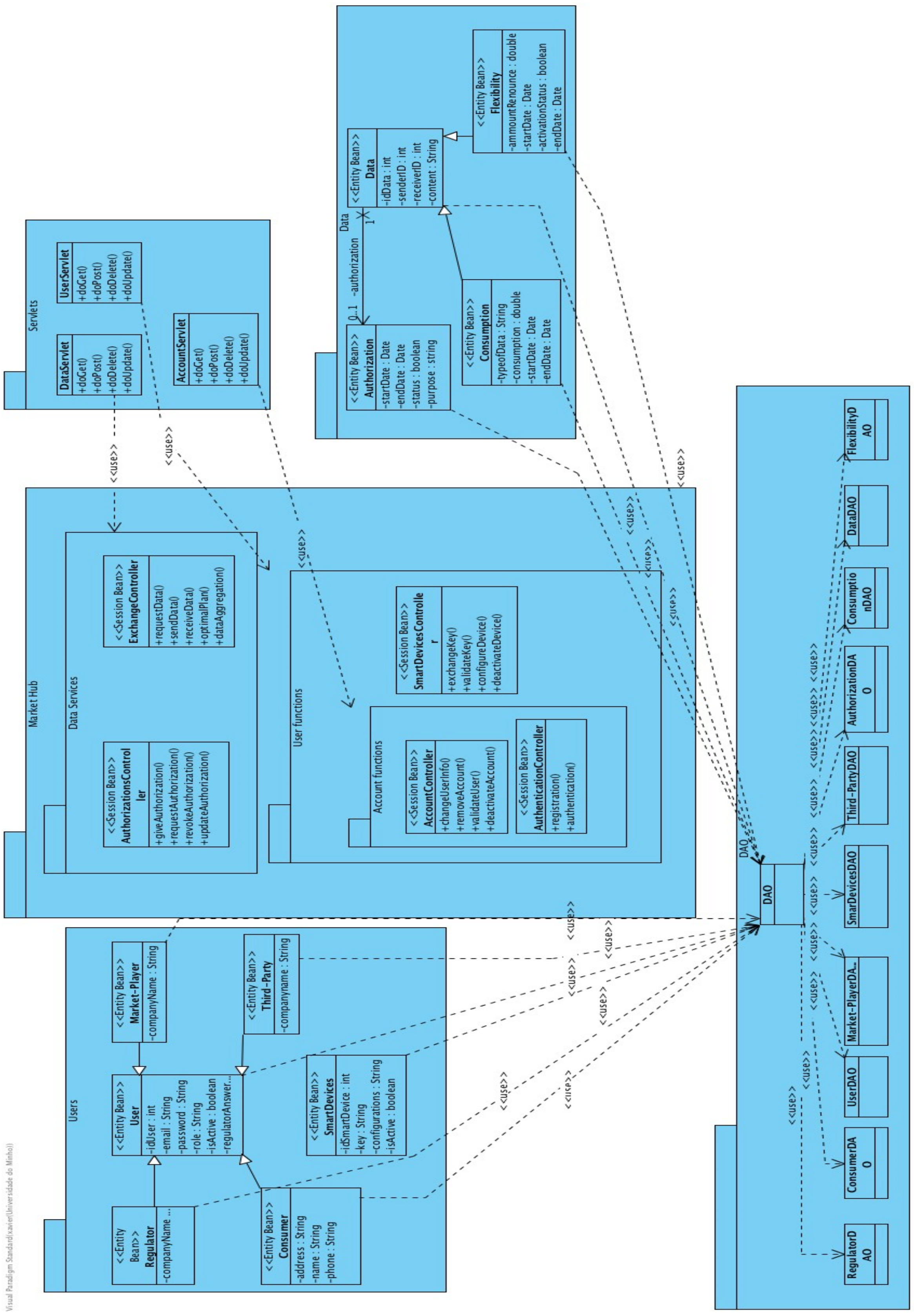

Servlets
DataServlet
UserServlet
AccountServlet
Data
<<Entity Bean>> Authorization
<<Entity Bean>> Data
<<Entity Bean>> Flexibility
<<Entity Bean>> Consumption
Market Hub
Data Services
<<Session Bean>> ExchangeController
<<Session Bean>> AuthorizationsController
User functions
<<Session Bean>> SmartDevicesController
Account functions
<<Session Bean>> AccountController
<<Session Bean>> AuthenticationController
Users
<<Entity Bean>> User
<<Entity Bean>> Market-Player
<<Entity Bean>> Third-Party
<<Entity Bean>> SmartDevices
<<Entity Bean>> Regulator
<<Entity Bean>> Consumer
DAO
FlexibilityDAO
DataDAO
ConsumptionDAO
AuthorizationDAO
Third-PartyDAO
SmarDevicesDAO
Market-PlayerDA...
UserDAO
ConsumerDAO
RegulatorDAO
<<use>>

Figure 20: Class Diagram

Next, there is a brief explanation of each class:

- **User** – this class will generally represent all types of users. Contains all the attributes and methods that will be common to all types of users. Because it is an entity that we will need to persist, it has been tagged as an entity bean.

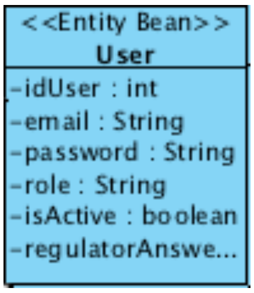


Figure 21: User class

- **Market Player** – this class will represent Market Players types of users. Market Players are entities that belong to the specific sector and are linked to the exchanges of data made. This class will be a subclass of User and will inherit all variables of that same class. Because it is an entity that we will need to persist, it has been tagged as an entity bean.

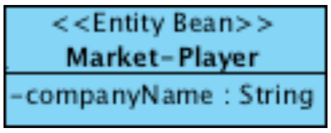


Figure 22: Market Player class

- **Consumer** – this class will represent Consumers types of users. This class will be a subclass of User and will inherit all variables of that same class. Because it is an entity that we will need to persist, it has been tagged as an entity bean.

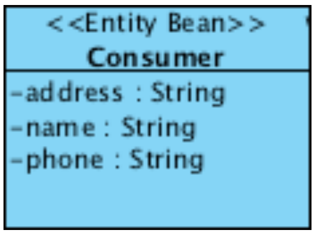


Figure 23: Consumer class

- **Regulator** – this class will represent Regulator type of users and will have as main function the management and mediation of the data exchange in the platform. This class will be a subclass of User and will inherit all variables of that same class. Because it is an entity that we will need to persist, it has been tagged as an entity bean.

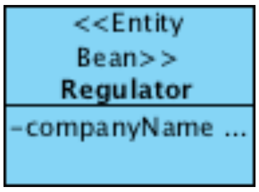


Figure 24: Regulator

- **Smart Devices** – this class will represent all types of machine and devices users. Because it is a user type different from the human user, it has no hierarchical relationship with the User class. Because it is an entity that we will need to persist, it has been tagged as an entity bean.

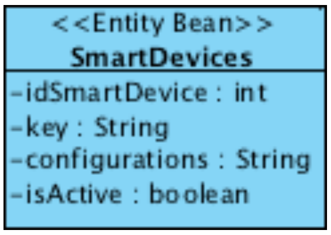


Figure 25: Smart Devices class

- **Data** – this entity represents all types of data which can be exchanged on the platform. This class will be the parent class of the classes of precise data types, Consumption and Flexibility. Because it is an entity that we probably will need to persist, it has been tagged as an entity bean.

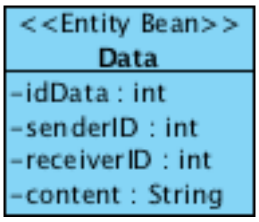


Figure 26: Data class

- **Consumption** – represents all the information relative to consumptions of a determined good. Inherits all variables and methods of the class superclass Data. Because it is an entity that we probably will need to persist, it has been tagged as an entity bean.

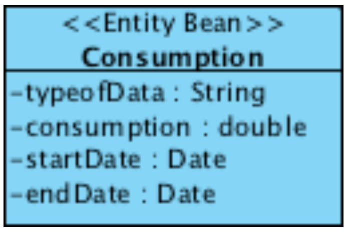


Figure 27: Consumption class

- **Flexibility** - represents all the flexibilities to be liberalized from a particular good. Inherits all variables and methods of the class superclass Data. Because it is an entity that we probably will need to persist, it has been tagged as an entity bean.

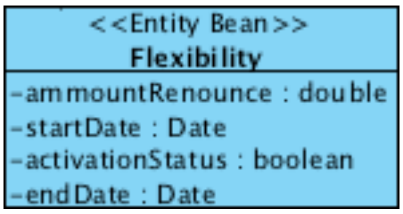


Figure 28: Flexibility class

- **Authorization** - this class represents the authorization associated with the use/access to a certain set of data. Because entity that will have to be used for several access validations, it has been tagged as an entity bean and therefore can be persisted.

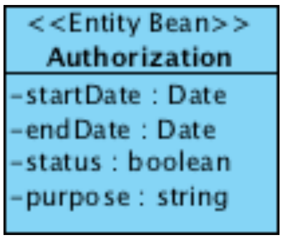


Figure 29: Authorization class

- **AuthorizationsController -** this class contains all methods which serve to manipulate the class authorization. Is tagged as a session bean in order to encapsulate all business logic behind all the methods contained in this class. This way, the session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

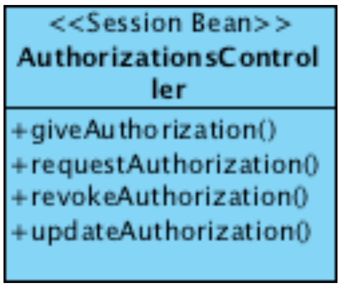


Figure 30: AuthorizationsController class

- **ExchangeController-** this class contains all methods which serve to manipulate the class Data and all the methods that makes possible to share and transfer it from one entity to the another. This class contains 2 methods (**dataAggregation** and **optimalPlan**) that perform data analysis and then present this analysis so that the user can easily interpret them. The first compiles a dataset statistically and return the same compilation. The second takes on all the consumptions and flexibilities of a consumer's, stored in the database, normally about a time interval (a month) and returns an optimal plan for that month.
  Is tagged as a session bean in order to encapsulate all business logic behind all the methods contained in this class. This way, the session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

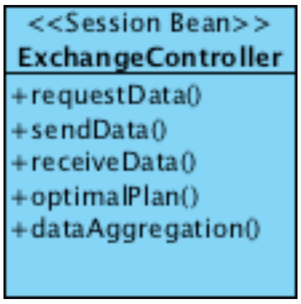


Figure 31: ExchangeController class

- **SmartDevicesController-** contains all the methods to configure and manage all platform-connected machine devices. Is tagged as session bean in order to encapsulate all business logic behind all the methods contained in this class.

This way, the session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

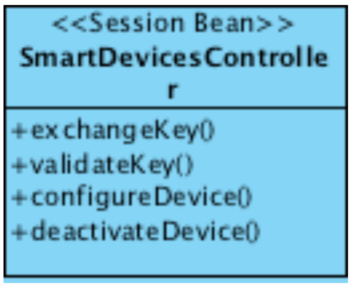


Figure 32: SmartDevicesController class

- **AccountController-** this class contains all methods which enable a human user to manage their user account settings and information. Is tagged as a session bean in order to encapsulate all business logic behind all the methods contained in this class. This way, the session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

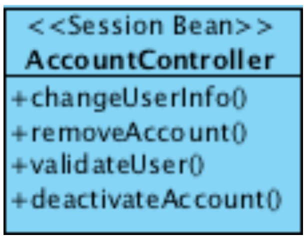


Figure 33: AccountController Class

- **AuthenticationController-** this class contains all methods for registration and login in the platform. Is tagged as a session bean in order to encapsulate all business logic behind all the methods contained in this class. This way, the session bean performs work for its client, shielding it from complexity by executing business tasks inside the server.

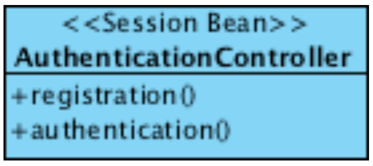


Figure 34: AuthenticationController Class

5.3 Data Model

The foundation for the conceptual data model in Figure 35, is the information required for a supported functioning and thorough record of data communication/sharing provided by the Generic Market Hub, which does not comprise confidential information of any user of the platform.

The core data structure of the Generic Market Hub records information related with users, either humans or machine users/small electronic devices, as well as all the information inherent to the process of data sharing, in this case the data access authorizations.

This data model was generated based on the entity beans of the class diagram (Figure 19) and according to the Object-relational mapping rules. Attending to the User class of the Class Diagram (Figure 20), we see that it contains the variables `idUser`, `email`, `password`, `role`, `isActive`, `regulatorAnswer`. Through the application of ORM rules, we obtain a table with the name `User`, with the primary key `idUser` and the other attributes that derived from the class `User`.

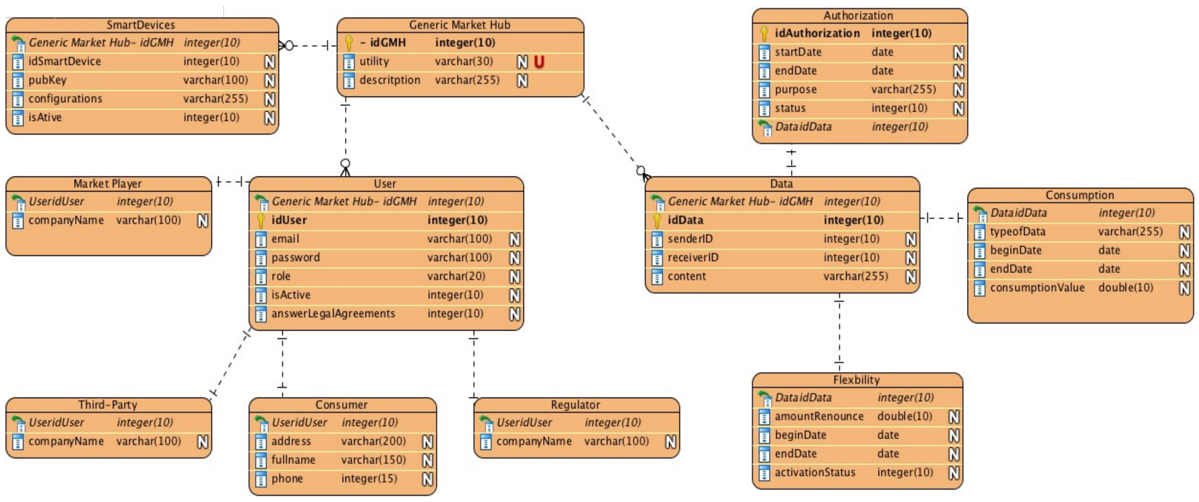


Figure 35: Generic Market Hub Data Model

5.4 Behavioral Modelling

The process started with the modelling of the platform at the structural level. In this level, variables and methods that constitute each class were specified. The following step was the behavior specification. At this modelling level each method presents in the classes **`AuthenticationController`**, **`ExchangeController`**, **`AuthorizationsController`**, **`SmartDevicesController`** and **`AccountController`** were specified.

For each method, we produced three types of refined version of a UML Sequence Diagram (System, Subsystem and Implementation), each one with a different abstraction level. Using the sequence diagrams, it was possible to represent the sequence of processes (more specifically, the messages passed between objects) in a computer program.

As we have many methods, it can be difficult to determine the overall behavior of each of one of them. Therefore, the sequence diagrams were used for a more simple and logical representation of all the actions that constitute the normal operation of the platform.

Following, we are going to use the method **`OptimalPlan`** from the class **`ExchangeController`** to demonstrate how the whole process of drawing diagrams was carried out. Diagrams of the remaining methods of all classes can be found in the Appendices.

The method **`OptimalPlan`** has as action of allowing a Consumer, through consumptions and flexibilities submitted during a certain time interval, to obtain an optimum consumption plan. This optimal plan will be submitted to the various market players that distribute and market the utility. Then each market player will bid a monthly cost for this same plan. At the end, the consumer will opt for the bidding that is more economical and efficient.

5.4.1 System Sequence Diagram

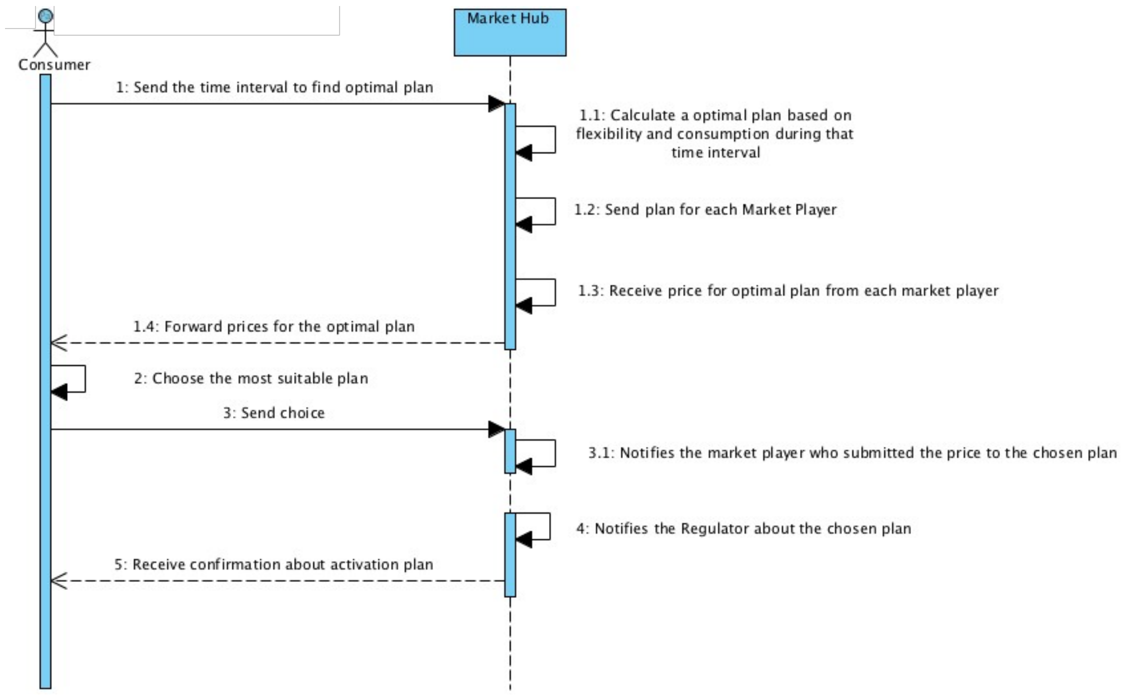
A System Sequence Diagram (SSD) is refined version of a UML Sequence Diagram, which supports the description of the progression of events over the time as sequence of steps describing the interaction between the User and the system.

Figure 36 demonstrates, step by step, the entire flow of messages shared by the Consumer and the Generic Market Hub (`Market Hub`).

The flow starts with a message coming from the actor `Consumer` (step 1). That message has a time interval (start date and end date) and has as receiver the `Generic Market Hub`. Inside the `Generic Market Hub`, a calculation considering the consumptions and flexibilities between the specified time interval is done (step 1.1).

The calculation will produce a plan. Then the plan will be forward to all `Market Player` (step 1.2) with the intention that each of them respond with a price for the submitted plan (step 1.3). These prices will be sent to the `Consumer` (step 1.4). This will opt for one price (step 2) and send it via `Generic Market Hub` (step 3).

Then the `Market Player`, to whom belongs the price that the Consumer chose, is notified (step 3.1). After that, a notification is sent to the `Regulator` (step 4). The flow ends with a confirmation message from the `Generic Market Hub` to the `Consumer` (step 5).



Consumer
Market Hub
1: Send the time interval to find optimal plan
1.1: Calculate a optimal plan based on flexibility and consumption during that time interval
1.2: Send plan for each Market Player
1.3: Receive price for optimal plan from each market player
1.4: Forward prices for the optimal plan
2: Choose the most suitable plan
3: Send choice
3.1: Notifies the market player who submitted the price to the chosen plan
4: Notifies the Regulator about the chosen plan
5: Receive confirmation about activation plan

Figure 36: System Sequence Diagram from OptimalPlan method

This diagram will serve as the basis for the subsystem diagram.

5.4.2 Subsystem Sequence Diagram

In the Subsystem Sequence Diagram (Figure 37), further detail about all processes that compose this method are specified. In addition to the actor `Consumer`, we have all subsystems represented in the process, i.e. `Generic Market Hub`, `Market Player` and `Regulator`.

In this diagram, we start with message sent from the `Consumer` to the `Generic Market Hub` (step 1). The `Generic Market Hub` calculates an optimal plan (step1.1). Then we have a loop which is responsible from sending the optimal plan for all the `Market Player` (step 1.2) and then, wait for a response from each one of them (step 2). After that, each response is forward to the `Consumer` (step 2.1) that will choose a submitted price for the optimal plan (step 3). That information will be forward, via `Generic Market Hub` (step 4), to the `Market Player`, who submitted it and then (step 4.1), the `Market Player` is going to activate it (step 4.1.1).

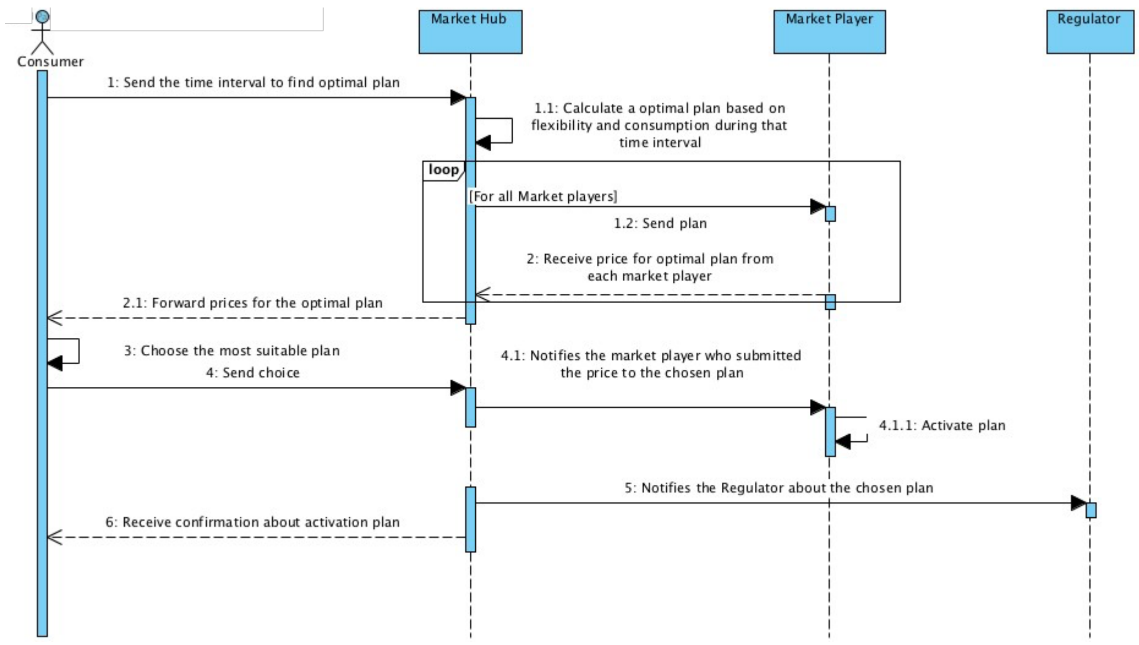

Consumer
Market Hub
Market Player
Regulator
1: Send the time interval to find optimal plan
1.1: Calculate a optimal plan based on flexibility and consumption during that time interval
loop
[For all Market players]
1.2: Send plan
2: Receive price for optimal plan from each market player
2.1: Forward prices for the optimal plan
3: Choose the most suitable plan
4: Send choice
4.1: Notifies the market player who submitted the price to the chosen plan
4.1.1: Activate plan
5: Notifies the Regulator about the chosen plan
6: Receive confirmation about activation plan

Figure 37: OptimalPlan Subsystem Sequence Diagram

Then a message will be sent for the `Regulator`, notifying him about the chosen plan (step 5). This process ends with a confirmation message from the `Generic Market Hub` to the `Consumer` (step 6).

5.4.3 Implementation Sequence Diagram

In the Implementation Diagram (Figure 38), it is presented a description of the process, in an abstraction level closer to the implementation. In this diagram, the auxiliary functions that compose the method appear. For each of these functions it is possible to infer the input/output of each one of them.

Although, final implementation may vary with what is described on this diagram. During implementation, this version can suffer minor changes due to the use of concrete technologies. Nevertheless, the whole process detailed here will serve as guide in a first implementation.

The process starts with a function, `optimalplan`, that has two parameters (`startDate` and `endDate`). Then, inside the `Generic Market Hub` is invoked a function, `calculatePlan`, that generates a plan, receiving as parameters a start and end date.

After that, we proceed to a loop cycle where the collection market players are iterated. At each iteration, a sequence of three functions are invoked. The first one is responsible for communicate the plan to the Market Player, the second one receives the plan and calculate a price for that plan, the third one communicates that price to Generic Market Hub.

As plans are received in the `Generic Market Hub`, they are stored in a collection. As soon as the loop ends, the collection is forward to the `Consumer` through the function `sendPrices`. Then, the `Consumer` will invoke the function `choosePrice`, which results in the choice of a price. This price is communicated to the Generic Market Hub by the function `sendActivatePrice`. The `Generic Market Hub` will forward a request for activate the plan (`activatePrice`) to the `Market Player` who submitted that price and he will activate that plan.

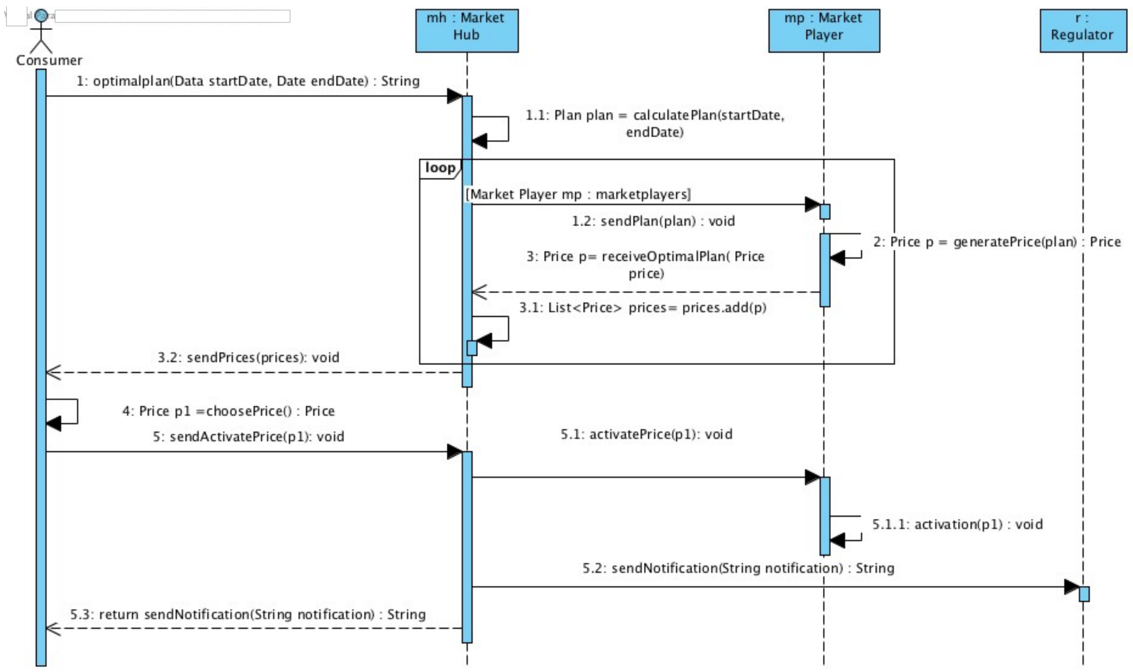

Consumer
mh : Market Hub
mp : Market Player
r : Regulator
1: optimalplan(Data startDate, Date endDate) : String
1.1: Plan plan = calculatePlan(startDate, endDate)
loop
[Market Player mp : marketplayers]
1.2: sendPlan(plan) : void
2: Price p = generatePrice(plan) : Price
3: Price p= receiveOptimalPlan(Price price)
3.1: List<Price> prices= prices.add(p)
3.2: sendPrices(prices): void
4: Price p1 =choosePrice() : Price
5: sendActivatePrice(p1): void
5.1: activatePrice(p1): void
5.1.1: activation(p1) : void
5.2: sendNotification(String notification) : String
5.3: return sendNotification(String notification) : String

Figure 38: OptimalPlan Implementation Diagram

To conclude the process, the `Generic Market Hub` will send two notifications. The first one is for the `Regulator`, which will be informed about the plan activation. The second one is for the `Consumer`, which will be informed that the entire process concluded successfully, and the plan was activated.

6. Development

In this chapter, all steps taken during the development phase will be presented. We will start for explaining how each diagram drawn aided the development phase, i.e. a presentation of what each diagram designed contributes to the development and documentation of the platform.

After that, we will present the Design Patterns and Architectural Patterns which were applied to design and solve problems in the implementation of the platform. To complement the entire development, all the technologies and languages used to design the platform will be specified.

A concrete application of the Generic Market Hub will be depicted from the models to the source code. This application was born through a refinement to a precise utility.

6.1 Development Approach

In the development phase of the Generic Market Hub, we have followed model driven approach, so the code production is based on the diagrams (Packages, Classes, System, Sub-system and Implementation) produced. All diagrams produced were followed as a structure and guide for this implementation phase, as shown in Figure 39.

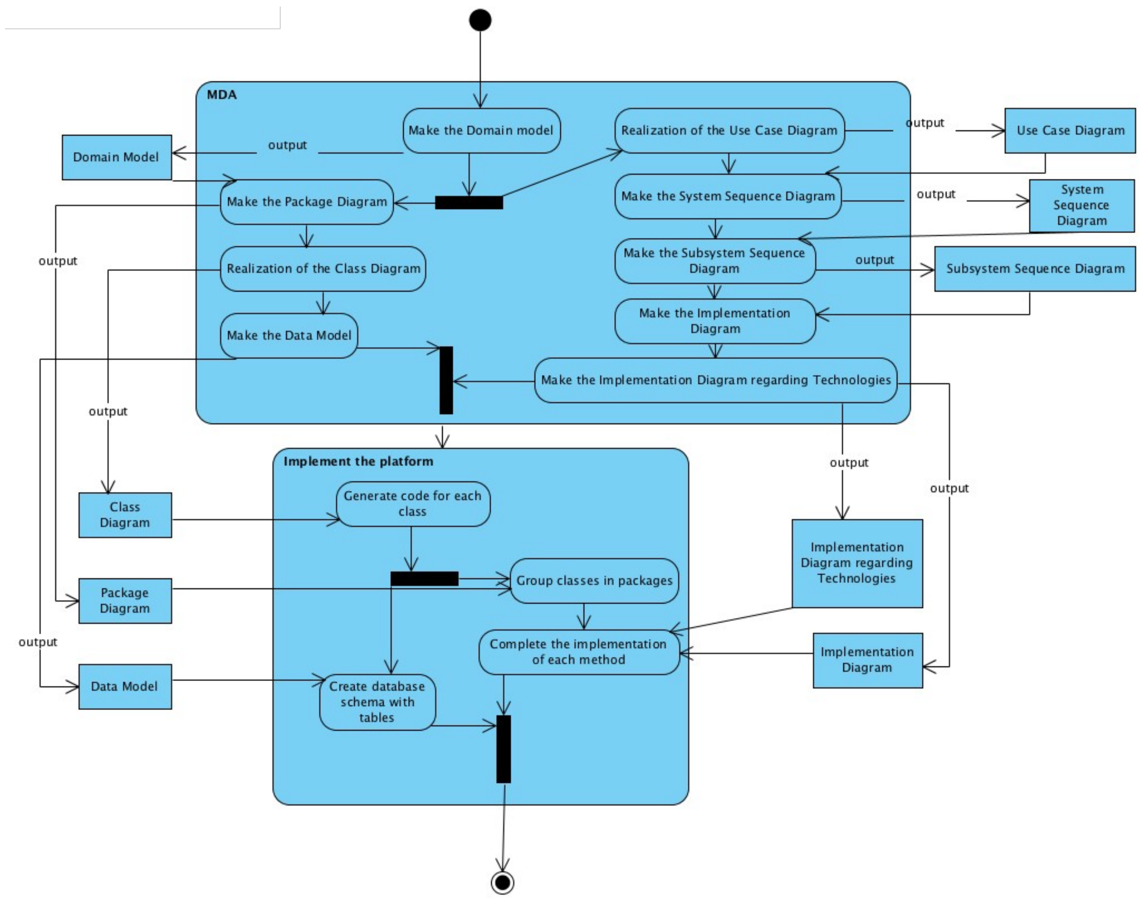

MDA
Make the Domain model
Realization of the Use Case Diagram
output
Use Case Diagram
Domain Model
output
Make the Package Diagram
Make the System Sequence Diagram
output
System Sequence Diagram
output
Realization of the Class Diagram
Make the Subsystem Sequence Diagram
output
Subsystem Sequence Diagram
Make the Data Model
Make the Implementation Diagram
Make the Implementation Diagram regarding Technologies
output
Implement the platform
output
output
Class Diagram
Generate code for each class
Group classes in packages
Implementation Diagram regarding Technologies
Package Diagram
Complete the implementation of each method
output
Data Model
Create database schema with tables
Implementation Diagram

Figure 39: Development approach diagram

The modelling process started with the Domain Model. Resorting to the details of concepts, roles, datatypes, individuals, and rules of the Domain Model, the Package and Use Case diagrams were drawn. With the structure and dependencies between sub-systems and modules represented in the Package Diagram and datatypes and individuals from the Domain Model, we draw the Classes Diagram. Using the Classes Diagram and applying ORM rules, we were able to extract the Data Model. With each use case from the Use Case Diagram, we drawn three types of refined version of a UML Sequence Diagram: System Sequence Diagram, Subsystem Sequence Diagram and Implementation Diagram.

For this development phase, and considering all diagrams drawn, it was possible to generate some code from those diagrams.

- From the Classes Diagrams, we extracted the code for all classes. This process accelerates the development as both variables from each class and the definition of each method are automatically generated.
- From the Data Model, we were able to generate a SQL script for create the database schema and all tables which set the schema.

- The code for each method was produced considering the implementation diagrams, normally the sequence diagrams. This way we ensured that the code was well documented and easy to maintain.

6.2 Architectural design

During the Architectural design, we applied the best practices regarding classes structure and dependencies. Those good practices were based on the use of Design Patterns. During the refinement of the Class Diagram, we highlight problems that would arise during development and for each problem we use a design pattern solution.

Following, there is an explanation about every Design Pattern used:

1. **Facade Pattern:**

Normally the database access logic is composed of many methods and many of them are rather complex. To abstract all this complexity, a Facade pattern was used. This way we are providing an interface for the classes that implement it. In our application, we used a Facade pattern in all DAO classes to hide the complexity behind the implementation of the DAO itself (Figure 40).

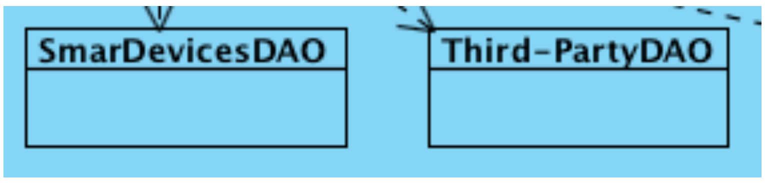


Figure 40: Facade Example

2. **MVC Pattern:**

This pattern was applied to separate all logic of the application in three layers, Model View and Controller. Some classes, like the ones presented in Figure 41, handle several layers at the same time.

For example, inside the Exchange Controller, the method `optimalplan` has to access the database base in order to fetch objects, these objects can be manipulated through methods and finally there will be a view to pass the object information to the end-user. All this workload must be separated in different layers. Therefore, in each controller we implemented each method using the MVC pattern.

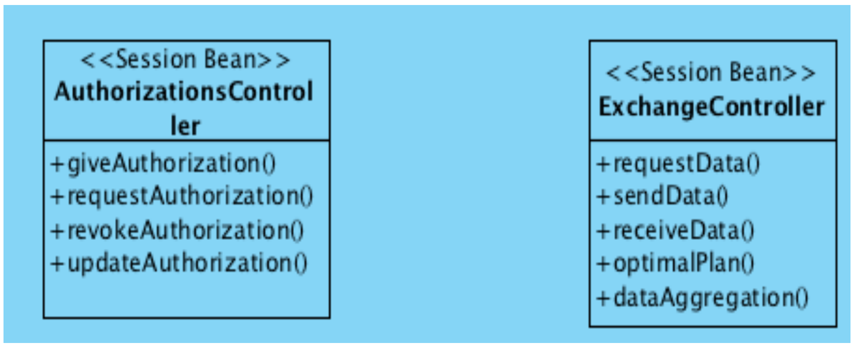


Figure 41: Example of MVC Pattern

3. DAO Pattern

As our platform will persist data, we need to have classes that perform those persistence functions. In these situations, it is necessary to have the separation level of the class that manipulates the model itself and class that deals with the persistence of this same model. To address this issue, a DAO design pattern was applied. This solution was applied for each model that composes the platform.

Figure 42 shows an excerpt of the application of the DAO design pattern.

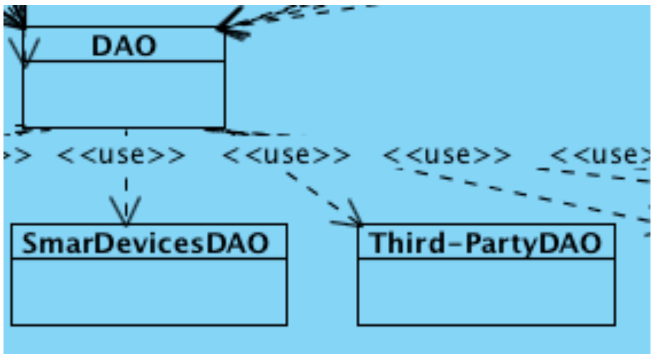


Figure 42: Example of DAO pattern

As for the software architecture, we used existing Architectural Patterns (or, Architectural Style) and apply them to this problem. Based on the entire structuring and planning of the platform, we choose **Layered Architecture** and **Model-View-Controller** (MVC) Architecture to sustain our problem.

This choice was due to the fact we wanted to divide the code of each classes by the functions that they perform and by the dependencies.

With this structuring, and in addition to the code being easier to read, it would be easier to keep it in the future without putting at risk the whole operation of the other groups of classes, layers, and the operation of the platform.

Thus, we divided the code into three group/layers of classes:

1. Classes that manipulated the models and dealt with its persistence in the database;
2. Classes that did all the intermediate logic and calculations;
3. Finally, classes that would deal with the communication (input/output) with the end-user;

6.3 Technological support

The language used for the development of the platform was **Java**. This choice was mainly due to the fact that after twenty-two years it remains a widely used language, programs can run on several different types of computer, has huge ecosystem of libraries and frameworks, and a super-optimized JVM runtime.

In order to develop faster, better and with certainty that we were developing an application that is in full compliance with the business rules, structured, and both maintainable and upgradable, we used the framework **Spring**[15]. The use of Spring brought simplicity and speed to the way the platform, being a web application, was built. In addition, it just needs a few lines of code to build a REST Application Programming Interface (API), Spring has a vast library that allows us to protect the application, interact with databases, allows to log in using other platforms, among others.

For the webservices, we opted for RESTful communication, in which HTTP/HTTPS is supported. One of the key advantages of REST APIs is that they provide a great deal of flexibility. Data is not tied to resources or methods, so REST can handle multiple types of calls and return different data formats.

This flexibility allows developers to build an API that meets all needs, meeting, also, the needs of very diverse customers. Data will be passed in JavaScript Object Notation (JSON) format.

In order to construct an efficient and documented API, we used **Swagger**. Swagger is a software tool framework backed by a large ecosystem of tools that helps developers design, build, document, and consume RESTful Web services.

Swagger has his own interface with which, and in a well-structured and documented way, exposes the whole API divided by the respective controllers. Is often used in large projects where it is necessary to make available the API by several development teams.

The **Swagger** can also be used as a way to test the proper functioning of the API.

For ORM, we use **Hibernate,** one of the most used ORM for Java, that allow us to create/map automatically the classes models with the data model that will be created in the database.

The class diagram presented in the Figure 19 is related to the generic platform. To be a solution, this platform needs to be instantiated to a concrete technology. In the context of this project **SAP** was chosen as a case study. With this decision we wanted to prove that our solution can be integrated into the systems of one of the companies that is the leader in the software services sector.

For this purpose, we used the SAP Cloud Platform with HANA database and the Identity Provider Service. The first service provides a deploy slot to launch a Web Application Resource (WAR). This WAR is running on an Apache Tomcat 8 server. This service guarantees that the application is always available, in a safe environment, able to meet several requests at the same time and in case of any failure, a process is triggered for which application to recover and become available again.

The second service, HANA[16] database, corresponds to an in-memory, column-oriented, relational database management service.

Its main function as a database server is to store and retrieve data as requested by the applications. In addition, it performs advanced analytics (predictive analytics, spatial data processing, text analytics, text search, streaming analytics, graph data processing) and includes ETL capabilities as well as an application server.

This service does its own management that ensures that the data is always delivered, the execution time of queries is very low and ensures that there is no loss of data.

The Identity Provider is a service responsible for managing all the registration and authentication features of the platform. It is composed by two components, User Management and Identity Lifecycle Management, this service has its own interface and a user data repository independent of the rest of the application. This service ensures that all bureaucracy around the User Personal Data (GDPR) is fulfilled.

For the User Interface, HTML and CSS was used. In order to maintain a structured and attractive interface, we use the **Bootstrap** framework. As for the logic and animation of the user inteface, we use Javascript. To simplify the code produced that will interact with the HTML, we used the library of functions **JQuery**.

After choosing the technologies, new Implementation diagrams were drawn, taking account technologies choices we have made. Following, in Figure 43, we have the example of an implementation diagram dependent of the technology of the `optimalPlan` method, presented in chapter 5, section 5.

The flow starts with a request method, `optimalplan` (step 1), which has as parameters two dates (`startDate` and `endDate`). Then, inside the `ExchangeController`, two requests are made to the mechanism `UserRepository`. The `UserRepository` allows a transparent access to ORM functionalities. The first request retrieves a collection of Consumptions of a User based on the `startDate` and `endDate` (step 1.1/2). The second one, a collection of Flexibilities of a User based on the `startDate` and `endDate` (step 1.2/3).

After retrieving both collections, a plan is calculated inside the `ExchangeController` (step 3.1). Then, we use the mechanism `UserRepository` to get a collection with all Market Players (step 3.2 and 4).

Following, we iterate through the Market Player collection sending a request to each one of them (step 4.1) through the function `sendPlan`. Each Market Player will send a price, through the function `receiveOptimalPlan` (step 5). Each Price received will be stored in a collection prices (step 5.1).

Those prices will be forward to the `Consumer`, over the method `sendPrice` (step 6). The `Consumer` will answer with the chosen price (step 7). Then, the Market Player that submitted that price will be notified step (7.1). The Regulator of the platform will be notified about this procedure through the function `sendNotification` (step 8).

To conclude the flow, the `Consumer` will receive a notification informing that the entire process was concluded successfully, and the plan was activated (step 9).

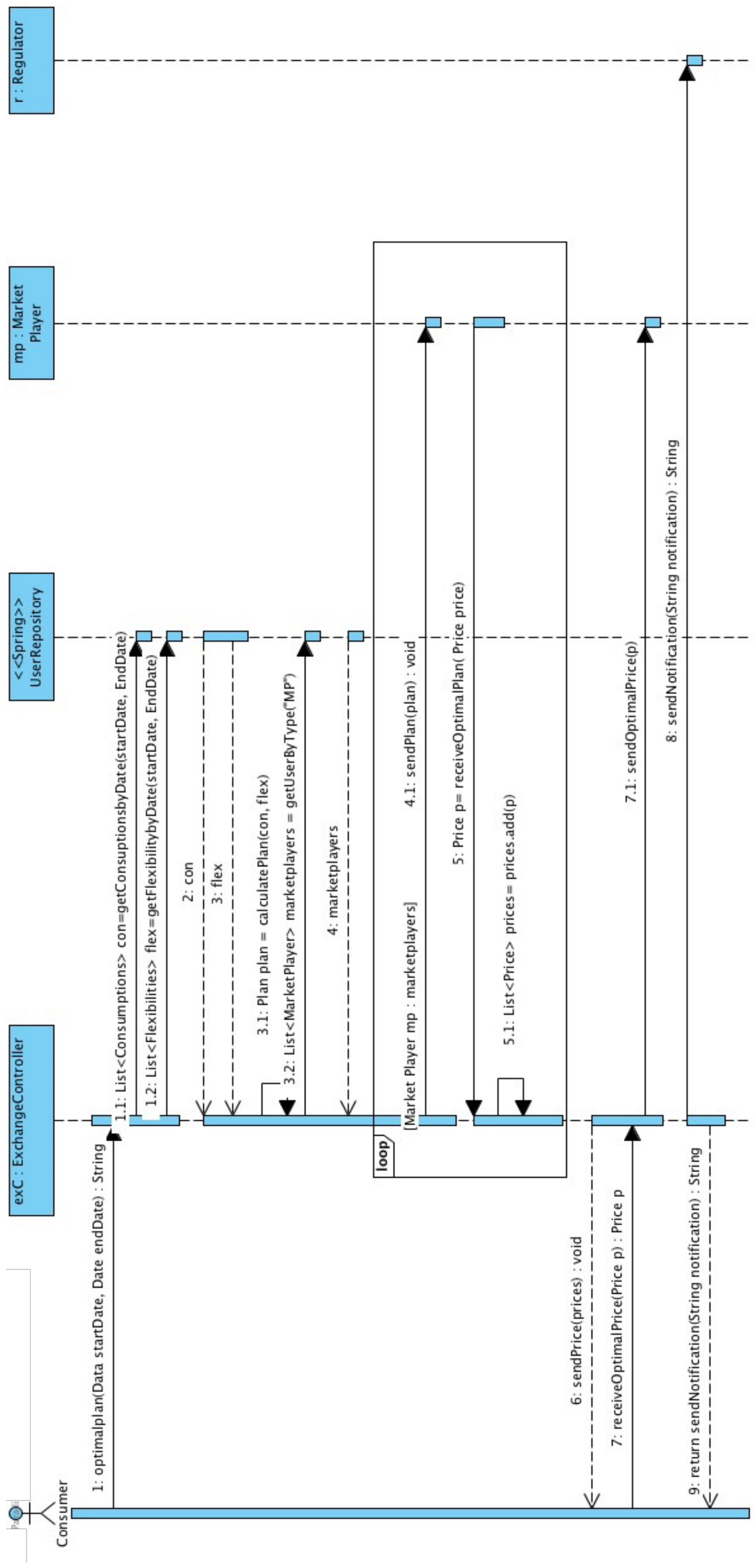
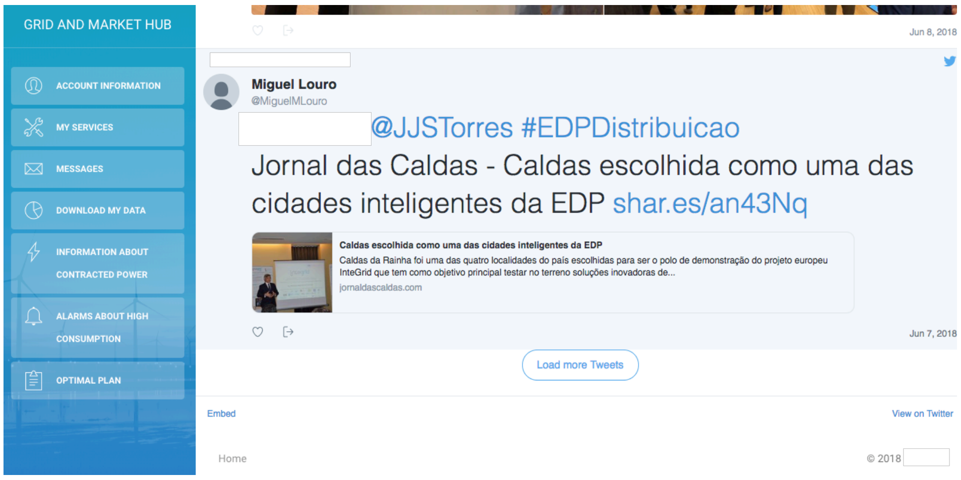

Consumer
exC : ExchangeController
<<Spring>> UserRepository
mp : Market Player
r : Regulator
1: optimalplan(Data startDate, Date endDate) : String
1.1: List<Consumptions> con=getConsuptionsbyDate(startDate, EndDate)
1.2: List<Flexibilities> flex=getFlexibilitybyDate(startDate, EndDate)
2: con
3: flex
3.1: Plan plan = calculatePlan(con, flex)
3.2: List<MarketPlayer> marketplayers = getUserByType("MP")
4: marketplayers
loop
[Market Player mp : marketplayers]
4.1: sendPlan(plan) : void
5: Price p= receiveOptimalPlan(Price price)
5.1: List<Price> prices= prices.add(p)
6: sendPrice(prices) : void
7: receiveOptimalPrice(Price p) : Price p
7.1: sendOptimalPrice(p)
8: sendNotification(String notification) : String
9: return sendNotification(String notification) : String

Figure 43: Implementation Sequence Diagram regarding the Technologies from OptimalPlan method

To interact with the **`OptimalPlan`** method the user, in this case a Consumer, will have an interface that allow him to use this method. After logging in, the consumer will have at his disposal a homepage (Figure 44) in which he can access various services and manage his personal data.

In this homepage and to interact with the **`OptimalPlan`** method, the consumer must click Optimal Plan in the services bar on the left side of the window. After clicking, a new window will appear (Figure 45) and at this point, the consumer has two date input. One is to specify the start date and the second is to specify the end date. Based on this time interval, this method will look at the consumptions made and flexibilities available in that timeframe and calculate an optimal plan.



Miguel Louro
@MiguelMLouro
@JJSTorres #EDPDistribuicao
Jornal das Caldas - Caldas escolhida como uma das cidades inteligentes da EDP shar.es/an43Nq

Caldas escolhida como uma das cidades inteligentes da EDP
Caldas da Rainha foi uma das quatro localidades do país escolhidas para ser o polo de demonstração do projeto europeu InteGrid que tem como objetivo principal testar no terreno soluções inovadoras de...
jornaldascaldas.com

Jun 7, 2018

Figure 44: Consumer homepage

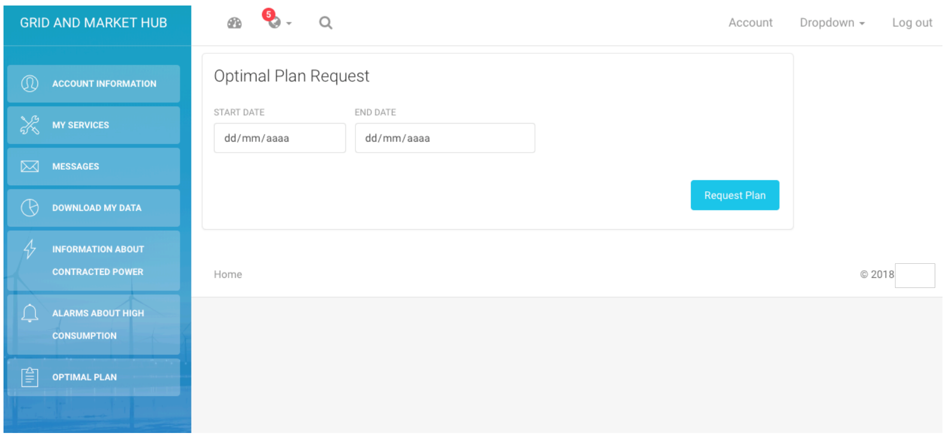


Figure 45: UI for request an optimal plan

6.4 Refinement of the class diagram

A concrete application of the Generic Market Hub is the gmhub of the European project Integrid. Figure 51 depicts gmhub class diagram which derived from a refinement of the Class Diagram of Generic Market Hub (Figure 20).

According to what has been presented in the class diagram of the Generic Market Hub regarding the several types of users, also in the gmhub application it is possible to see the necessity of having different types of Users.

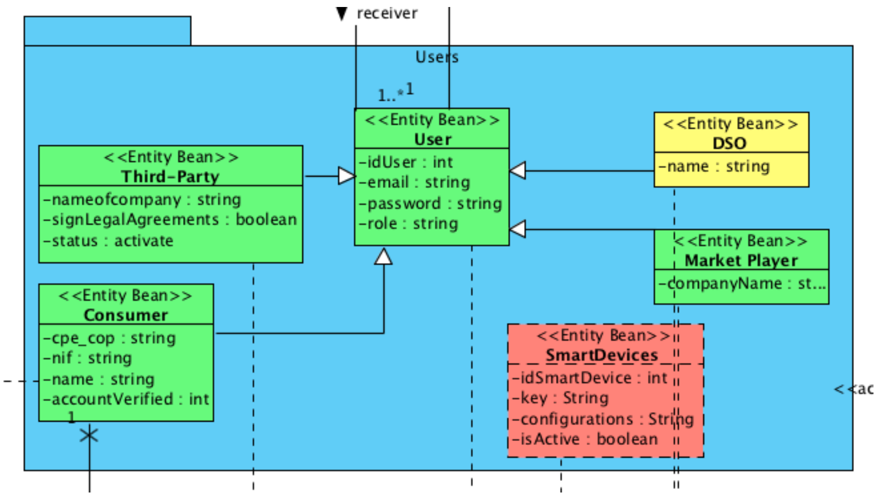


Figure 46: gmhub types of User

As shown in Figure 50, gmhub overall has the same types of Users as the Generic Market Hub. In green color, we have the types of users that are identical to the Generic Market Hub (`Consumer`, `Market Player` and `Third Party`). In yellow, we have the DSO class. This class represents an entity with the same role as the Regulator, in the Generic Market Hub, but with a specific name of the energy sector.

In this project there is no reference to the Smart Devices (red color). For this reason, it did not cross from the Generic Market Hub for the gmhub and, therefore, was removed.

Other aspect that is verified in both diagrams is the need to having an authorization (Figure 47) to access the data, that will be given by the Consumer.

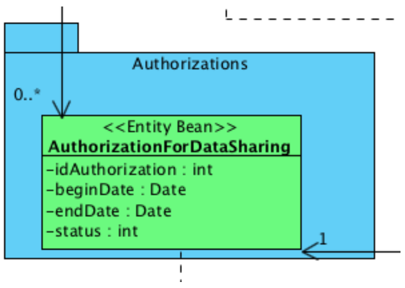


Figure 47: Authorization class from gmhub

Then, there are industry-specific services, such as Download My Data, Share My Data, Information feedback about contracted power, Alarms about high consumption and the possibility to create new service.

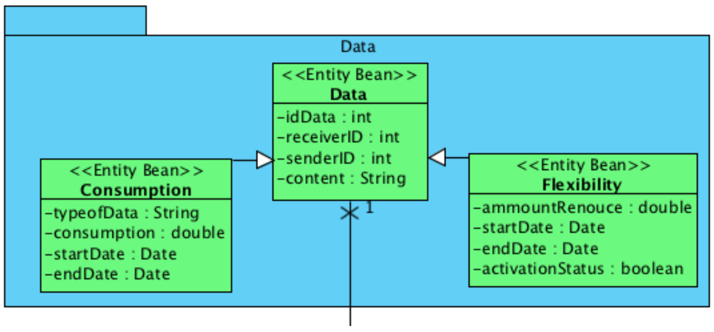


Figure 48: Data classes representation

These services (Figure 49) have as main function the exchange of specific data of the sector such as consumptions and flexibilities (Figure 48). Both datatypes transitioned from the Generic Market Hub Diagram.

- **Download Meter Data** – this service allows the Consumer to request the DSO his consumptions made in a specific time interval. The consumption will be presented in a CSV file.
- **Alarms about high consumption** – this service allows the Consumer to create an alarm which will analyze the consumption patterns and alert when there are big variances.
- **Feedback about contracted power** – this service allows the Consumer to have access to a dashboard in which various statistics regarding consumption and flexibilities are presented.
- **New service** – allows any type of user to create a new data service

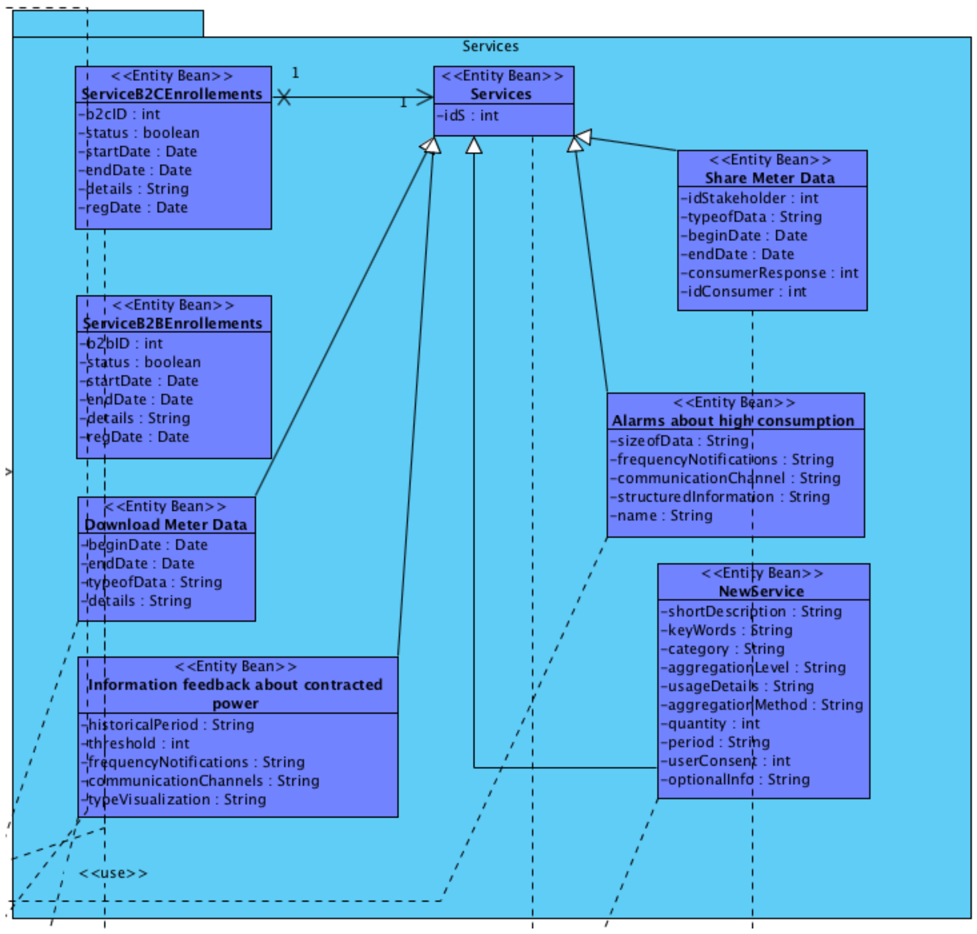


Figure 49: gmhub core services classes

To complete the functionalities of gmhub, there is the possibility of data exchange, via gmhub, between machine users. For such purpose there are the services **Traffic Light System** and **Flexibility Exchange to support grid operation**. Both have as core functions the exchange of data that allows to monitoring the grid status and to optimize it.

These services turn out to be a more concrete application and with accurate data types which use the methods `sendData()`, `receiveData()` and `requestData()` from the `ExchangeController` from the Generic Market Hub.

One service that would fit on gmhub platform would be `optimalPlan`. In this context, this service would allow the Consumer to find an optimal and cheap energy supply contract, based on its consumptions and flexibilities. One of the peculiarities of gmhub is the non-persistence of data referring to consumption, flexibilities or inherent to the management of the network.

Therefore, whenever it is necessary to consult these types of data, it is requiered to request them to the DSO. In case of some types of users, they will need an authorization to request it.

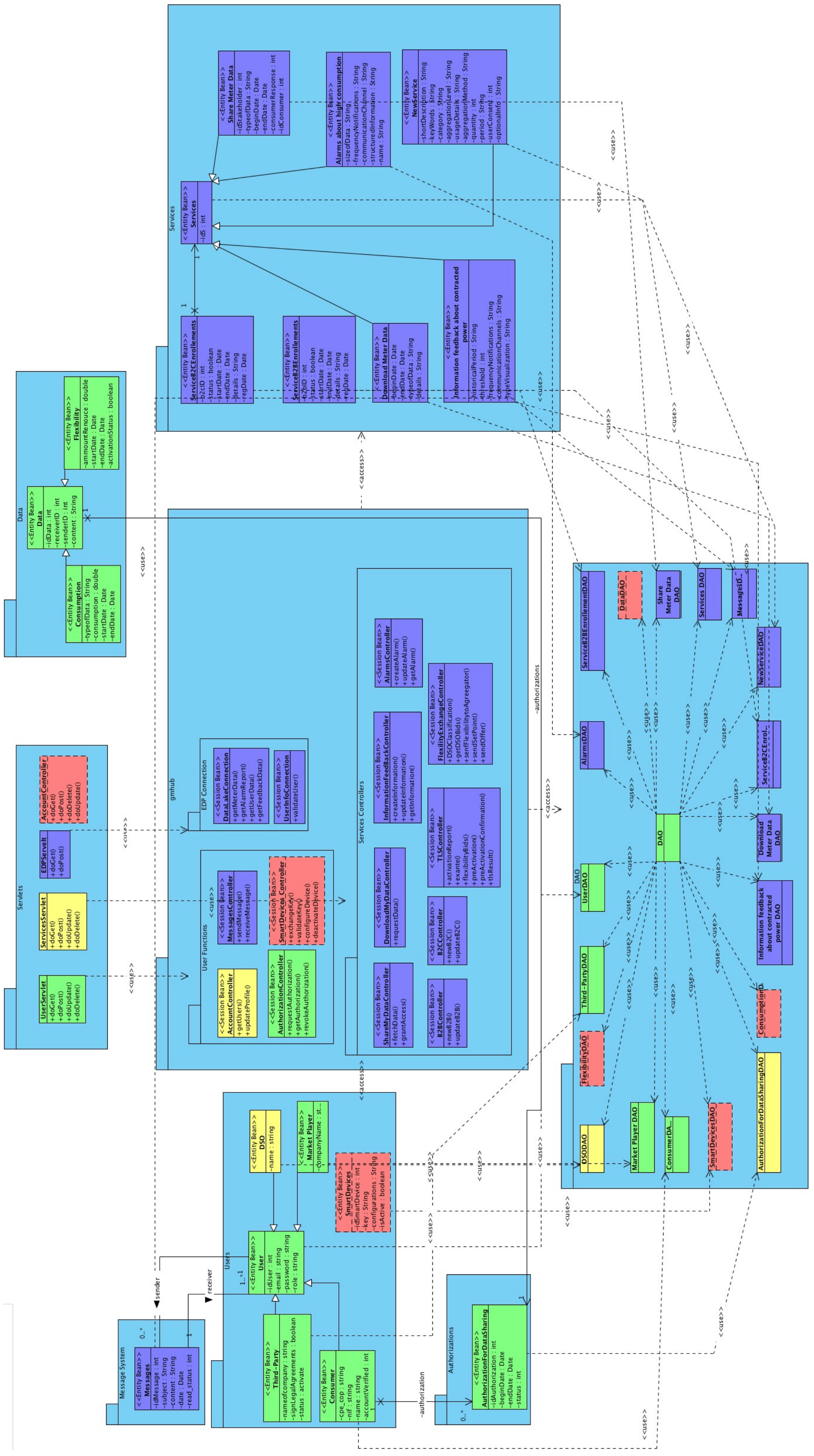
gmhub will record the services that were used, who has used, who sent the data, and who received the data. To complement this platform, there is an internal messaging system that allows communication between all users.

To better understand which classes have passed from the Generic Market Hub, which have been removed, which have undergone through changes and which are new, we have drawn a class diagram in which we illustrate the whole procedure.

Figure 46 depicts, with green color, all classes that transited from the class diagram of the Generic Market Hub. Then, with yellow color, we have classes that were extracted from the Generic Market Hub class diagram but have undergone some changes.

At red color, we have all classes that were in the Generic Market Hub class diagram, but which not transit to gmhub class diagram.

Finally, with purple color, we have all new classes that arose from the application of the generic model into a concrete domain.



Services
<<Entity Bean>> Services
<<Entity Bean>> Share Meter Data
<<Entity Bean>> Alarms about hight consumption
<<Entity Bean>> NewService
<<Entity Bean>> ServiceB2CEnrollements
<<Entity Bean>> ServiceB2BEnrollements
<<Entity Bean>> Download Meter Data
<<Entity Bean>> Information feedback about contracted power
Data
<<Entity Bean>> Data
<<Entity Bean>> Flexibility
<<Entity Bean>> Consumption
Servlets
UserServlet
ServicesServlet
EDPServlet
AccountController
gmhub
EDP Connection
<<Session Bean>> DataLakeConnection
<<Session Bean>> UserInfoConnection
User Functions
<<Session Bean>> MessagesController
<<Session Bean>> SmartDevices Controller
<<Session Bean>> AccountController
<<Session Bean>> AuthorizationController
Services Controllers
<<Session Bean>> AlarmsController
<<Session Bean>> FlexibilityExchangeController
<<Session Bean>> InformationFeedBackController
<<Session Bean>> TLSController
<<Session Bean>> DownloadMyDataController
<<Session Bean>> B2CController
<<Session Bean>> ShareMyDataController
<<Session Bean>> B2BController
DAO
ServiceB2BEnrollementDAO
DataDAO
Share Meter Data DAO
Services DAO
MessagesD...
NewServiceDAO
AlarmsDAO
ServiceB2CEnrol...
Download Meter Data DAO
UserDAO
Information feedback about contracted power DAO
Third-PartyDAO
ConsumptionD...
FlexibilityDAO
DSODAO
Market Player DAO
ConsumerDA...
SmartDevicesDAO
AuthorizationForDataSharingDAO
Users
<<Entity Bean>> User
<<Entity Bean>> DSO
<<Entity Bean>> Market Player
<<Entity Bean>> SmartDevices
<<Entity Bean>> Third-Party
<<Entity Bean>> Consumer
Message System
<<Entity Bean>> Messages
Authorizations
<<Entity Bean>> AuthorizationForDataSharing
<<use>>
<<access>>
-authorizations
-authorization
sender
receiver

Figure 50: From GMH to gmhub Class Diagram

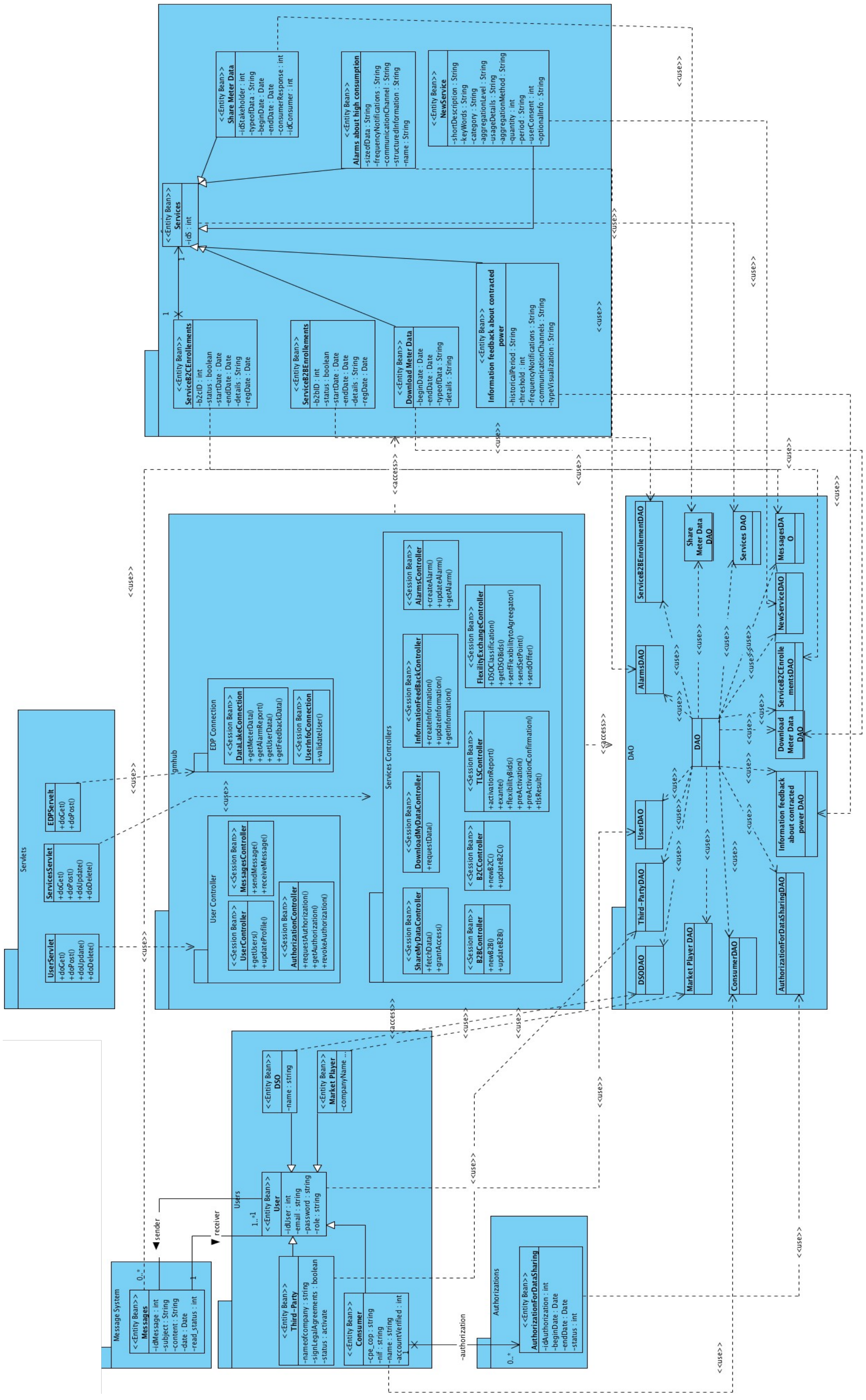


Figure 51: gmhub Class Diagram

6.5 Comparison between application cases

In both models (c.f. Figure 52 and 53) it is possible to see the existence of several types of users, such as Consumers, Retailer, Producer, Regulator and Distributor. The fact that the user's representation is kept between these two levels of abstraction, and consequently models, means that the concept of user initially specified is relevant to the platform. In fact, being Integrid a project of the energy sector, with specific requirements, it was necessary to add more types of users.

In gmhub, we added the Aggregator, Flexibility Operator, Industrial Consumer, Transmission System Operator (TSO) and Data Science Company type of User. The entity Regulator from the Energy Domain Model was replaced by the DSO in the gmhub domain model. The objective of the gmhub is that all participants involved in electricity trading, even those with less decision-making power, can participate in this trade.

When we moved from the energetic model, which represents a very closed type of commerce where only some type of user can participate, to the gmhub, there was a need to add new types of user.

Another difference is the concept of data in the Energy domain model. In this case, the data is still very abstract while in gmhub model domain begins to appear more concrete data such as Consumptions and Flexibilities.

For this data be shared securely and structured, the gmhub provides services such as Download My Data, Share My Data and Traffic Light System. These services classes are lacking in the Energy domain model (c.f. Figure 53) due to no concept of service.

As in the Energy domain model only a few stakeholders are involved in the data exchange, becomes futile the creation of services because data can be passed in a simpler way. For security reasons and for some services of the gmhub, a data authorization is needed. Therefore, the class Authorization in the gmhub domain model was added.

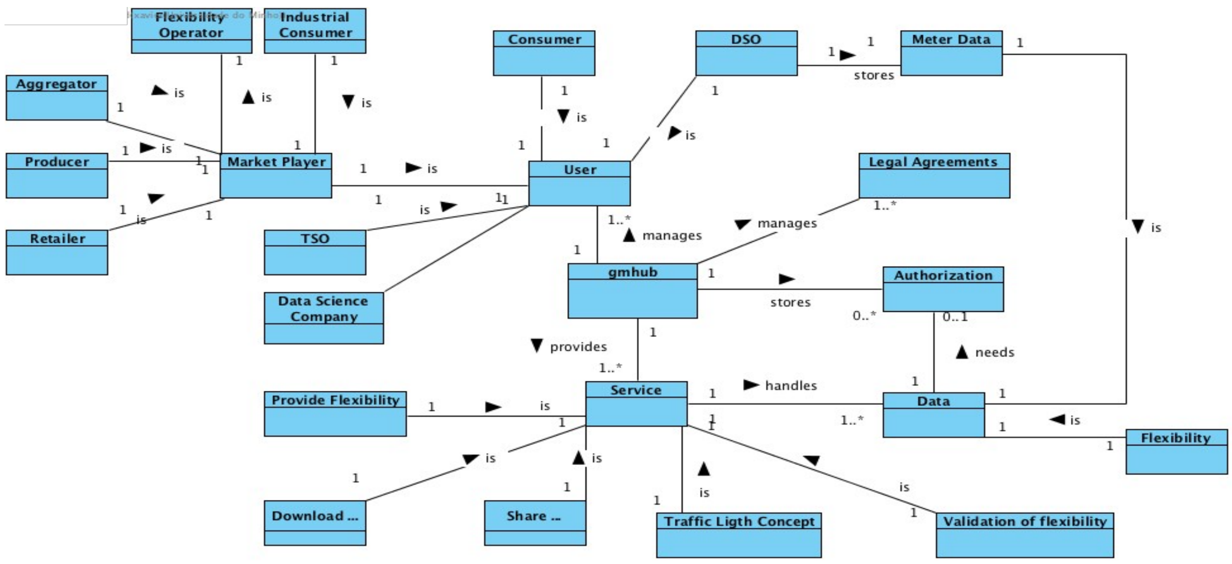


Figure 52: Integrid's gmhub domain model

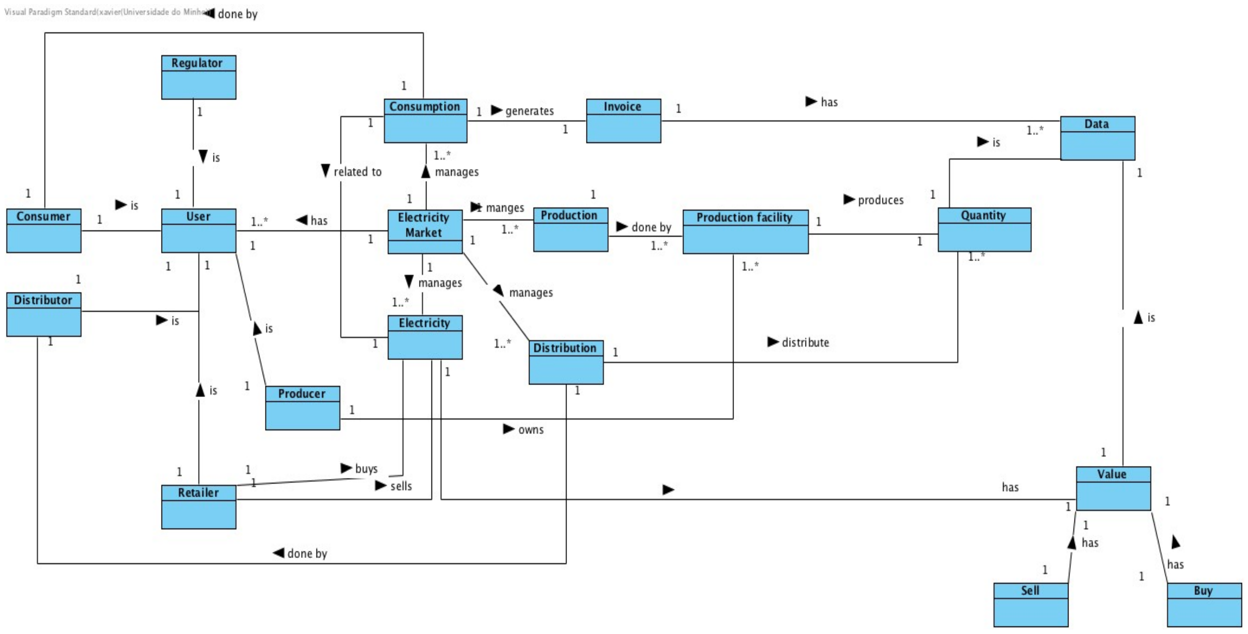


Figure 53: Energy domain model

As we saw, when we moved from the Energy domain model to gmhub domain model many changes had to be made and new classes emerged. This scenario has contributed to the development of the Generic Market Hub.

Our main objective was to fit the Generic Market Hub between the two models (c.f. Figure 52 and 53). This way we would ensure that the Generic Market Hub could be a solution for both cases.

Comparing both models, i.e. Energy domain model (Figure 53) and Generic Market Hub domain model (Figure 54), we see that both include various types of users. Some of these types are similar (Consumer and Regulator) but others arise from the need to use this platform as a solution for other utilities.

Smart Device, Market Player and Third-Party are types of users that were not in the Energy domain model but that were added so that this platform as heterogeneous as possible, making possible the representation of various types of user which compose each sector.

There is data communication in both models. In the Energy domain model, we have reference to Data regarding quantities and values. In the gmhub model, we have a general Data type which can represent various messages but also with very precise types such as consumptions and flexibilities.

As the Generic Market Hub promotes the exchange of data by all stakeholders, there was a need to include some protection over these data. Therefore, we need to include an authorization for data access. This authorization can be used or not, depending on the will of the data owner.

The authorization is lacking in the Energy model due to be a closed system where some relevant data is not shared. There are situations where data is not complete, there are details that are not available for the consumers in general and much of the data is restricted to the Regulator.

In short, there are some modifications between the Energy model and the Generic Market Hub. These modifications were done in order to generalize the platform to serve several utilities. Nevertheless, the Generic Market Hub is a modular solution that with minimal changes can fit in any solution. Proof of that is what we saw in chapter 6, section 6.4 where based on the Generic Market Hub class diagram, we were able to extend it to and create the gmhub for the Integrid project.

We can say that the Generic Market Hub can be a starting point for implementing solutions that involve the utilities sector.

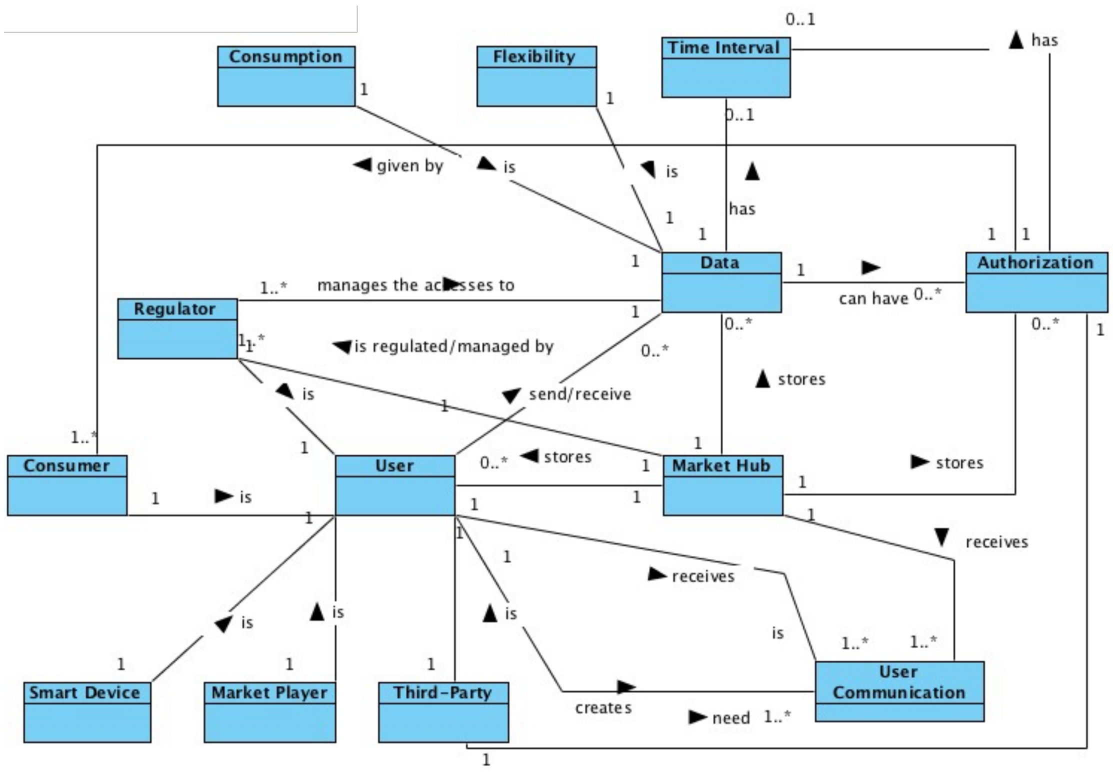


Figure 54: Generic Market Hub domain model

7.Conclusions

This chapter summarizes the work developed during this project. It finishes by presenting some ideas for future improvements and new developments that could be added to the Generic Market Hub.

In the context of the utilities sector and based on SE techniques, processes and methodologies, it was our goal to design, build and implement generic, modular platform that would take in consideration the concerns of the GDPR, namely the control of data access through the need of authorizations and user data protection through the use of the SAP IDP service. Another aspect of this generic platform is to allow several types of users.

The Generic Market Hub provides a safe and organized platform for data exchange. Using a software methodology and good practices of software construction, i.e. requirements analysis, UML modeling and development, we were able to design a platform modular, scalable, secure, with several types of user and able to ensure the exchange of data among several stakeholders of various utilities.

The main objective of this dissertation was to, based on domain models of several utilities and application of modeling and abstraction techniques, build a generic platform that can potentiate and streamline the exchange of data inherent to some utilities. At the end, we apply this solution to a real-world problem.

Currently, we have the entire design and implementation process of the Generic Market Hub. Based on a proposed methodology, we were able produce a set of models/diagrams with the intention of documenting and modeling all phases of the Generic Market Hub construction. The application of this cloud-based platform in the utilities sector will help the communication of data between all users. This platform was designed with rigorous methods.

This solution was applied in the design and development of the gmhub of the project Integrid. With this achievement, we could verify that the built generic platform can be the starting point to implement valid solution to support communication and data sharing among several users of a given utility.

During the realization of this project some limitations and problems were identified. At the time of modeling the three domains (water, light and gas), which served as the basis for the construction of the generic, it was difficult to get information to make those models as complete as possible. Hence, the generic model might have inherited some of the limitations.

It was difficult to find a data structure that represents consumption and flexibilities and that this structure could be used in all domains.

Finally, using SAP Cloud and its services resulted in an increased complexity during the implementation phase because the support and documentation given was not very explicit. Implementing in the SAP Cloud environment and using its services affected the extend to which the implemented platform can be considered generic.

This way, the implementation regarding some aspects like the use of the IDP and the database connection had to be done according to the norms and practices required by SAP. One of these requirements is the use of a specific database, SAP HANA, a relational database. For instance, a NoSQL database could be a viable approach but due to this restriction it was not possible to test in the context of this work.

7.1 Contributions

From this project, 2 scientific articles were produced. The first one was already published[6]. The second one was submitted and is awaiting final decision.

In short, the results achieved in the development of this project were:

- Definition of a methodology to follow;
- Domain analysis of 3 utilities (Water, Gas and Energy);
- Analysis of projects that aims to share data and the liberalization of sectors of utilities;
- Design a Generic Market Hub;
- Implementation of Integrid's gmhub;

7.2 Future Work

At the end of this project we assume that there is still some work to do and space to make some improvements to the platform. One of the next steps, and after finishing the implementation of the gmhub, is to further test the gmhub. So far everything that has been implemented has already been tested, both unitary and integration. However, the implementation is not finished yet and, in the end, it is necessary to make new tests including load tests. Through these tests we could confirm if the requirements were met and detect errors, if they exist.

It would be interesting to implement the Generic Market Hub and then test it in the domain of another utility, for example gas or internet. This way we would confirm if the Generic Market Hub fulfills one of the core requirements that is to be extensible for several domains.

Finally, and as a final test, check if the Generic Market Hub can make the exchange of data of several utilities at the same time. This way we could assure that the Generic Market Hub, even being a generic solution, can, in concrete cases, support the simultaneous communication of data.

Bibliography

[1] E. Union, “FLEXibility and energy effICIENCY based on metering data D6 . 0 – B2B Data Standard – EUMED,” pp. 1–65.

[2] “UPGRID | Real proven solutions to enable active demand and distributed generation flexible integration, through a fully controllable LOW Voltage and medium voltage distribution grid.” [Online]. Available: http://upgrid.eu/. [Accessed: 14-Jan-2018].

[3] “Green Button | Department of Energy.” [Online]. Available: https://energy.gov/data/green-button. [Accessed: 08-Jan-2018].

[4] “InteGrid - Smart Grid Solutions.” [Online]. Available: https://integrid-h2020.eu/. [Accessed: 14-Jan-2018].

[5] “Why Microsoft Is Ruling The Cloud, IBM Is Matching Amazon, And Google Is $15 Billion Behind.” [Online]. Available: https://www.forbes.com/sites/bobevans1/2018/02/05/why-microsoft-is-ruling-the-cloud-ibm-is-matching-amazon-and-google-is-15-billion-behind/#73874a291dc1. [Accessed: 21-Jun-2018].

[6] R. Bessa, F. COELHO Guido PIRES, P. G. MATOS Hossein SHAHROKNI Xavier RODRIGUES, and A. ALONSO Inês PRATES Aram MÄKIVIERIKKO Tiago SOARES EDP Distribuição KTH, “GRID AND MARKET HUB: EMPOWERING LOCAL ENERGY COMMUNITIES IN INTEGRID,” 2018.

[7] D. Garlan, “Software Architecture.”

[8] C. Szyperski, J. Bosch, and W. Weck, “Component-oriented programming,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999.

[9] E. M. Simão, “Comparison of Software Development Methodologies,” p. 110, 2011.

[10] S. R. Meier, “Technology Portfolio Management for Project Managers 28 Feb 17 PMI Tysons Corner Chapter,” pp. 1–31, 2017.

[11] I. Sommerville, *Software engineering*. 2007.

[12] The Agile Manifesto, “Manifesto for Agile Software Development,” *The Agile Manifesto*, 2001. [Online]. Available: http://agilemanifesto.org/.

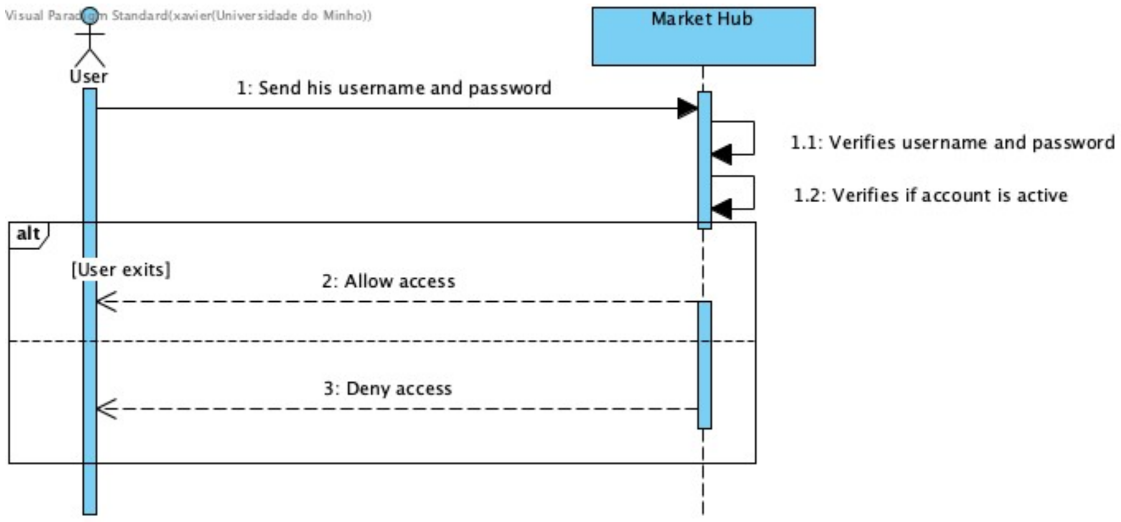
[13] Paper Rational Software White, "Rational Unified Process Best Practices for Software," *Development*, pp. 1–21, 2004.

[14] M. S. Palmquist, M. A. Lapham, S. Miller, T. Chick, and I. Ozkaya, *Parallel Worlds: Agile and Waterfall Differences and Similarities*, no. October. 2013.

[15] "Spring." [Online]. Available: https://spring.io/. [Accessed: 25-Oct-2018].

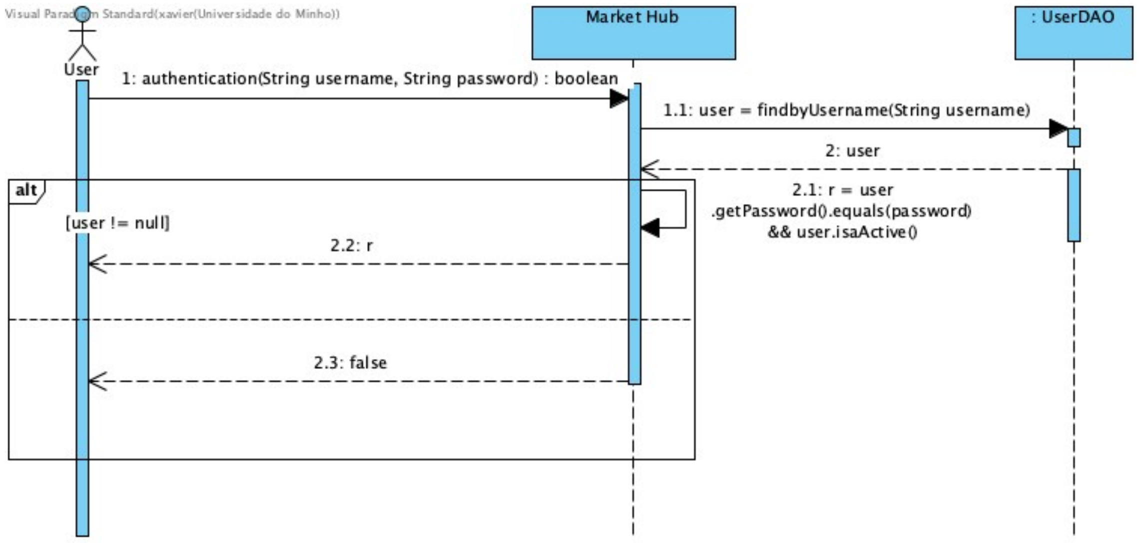
[16] "SAP HANA In Memory Database | Parallel Processing and RDBMS." [Online]. Available: https://www.sap.com/products/hana/features/in-memory-database.html. [Accessed: 23-Oct-2018].

Appendices



Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Send his username and password
1.1: Verifies username and password
1.2: Verifies if account is active
alt
[User exits]
2: Allow access
3: Deny access

Figure 55: Login System Sequence Diagram



Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: UserDAO
1: authentication(String username, String password) : boolean
1.1: user = findbyUsername(String username)
2: user
2.1: r = user .getPassword().equals(password) && user.isaActive()
alt
[user != null]
2.2: r
2.3: false

Figure 56: Figure 52: Login Implementation Sequence Diagram

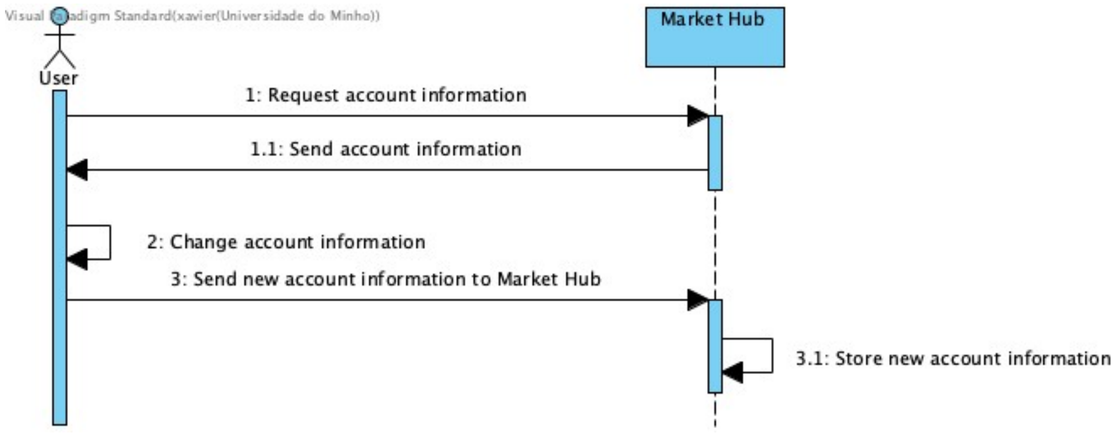

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Request account information
1.1: Send account information
2: Change account information
3: Send new account information to Market Hub
3.1: Store new account information

Figure 57: Update Profile System Sequence Diagram

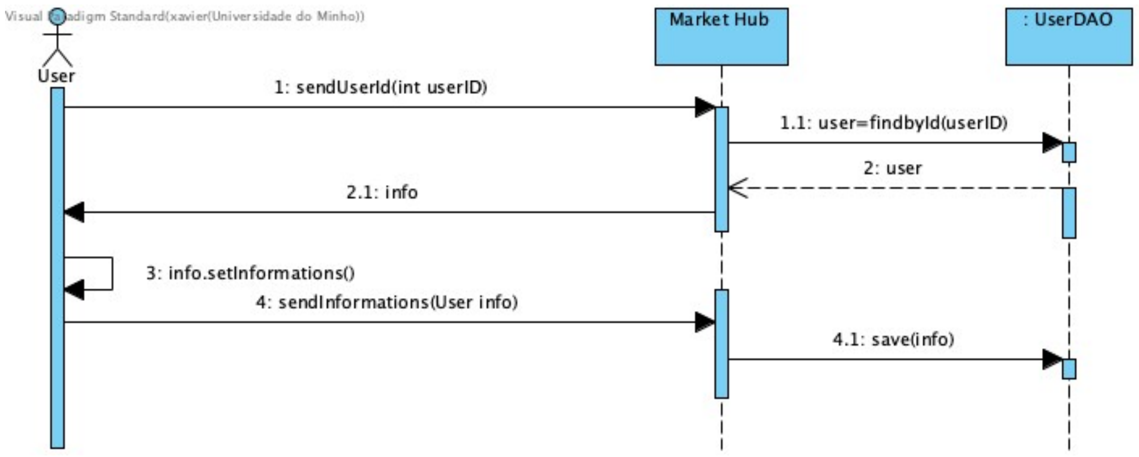

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: UserDAO
1: sendUserId(int userID)
1.1: user=findbyId(userID)
2: user
2.1: info
3: info.setInformations()
4: sendInformations(User info)
4.1: save(info)

Figure 58:Update Profile Implementation Sequence Diagram

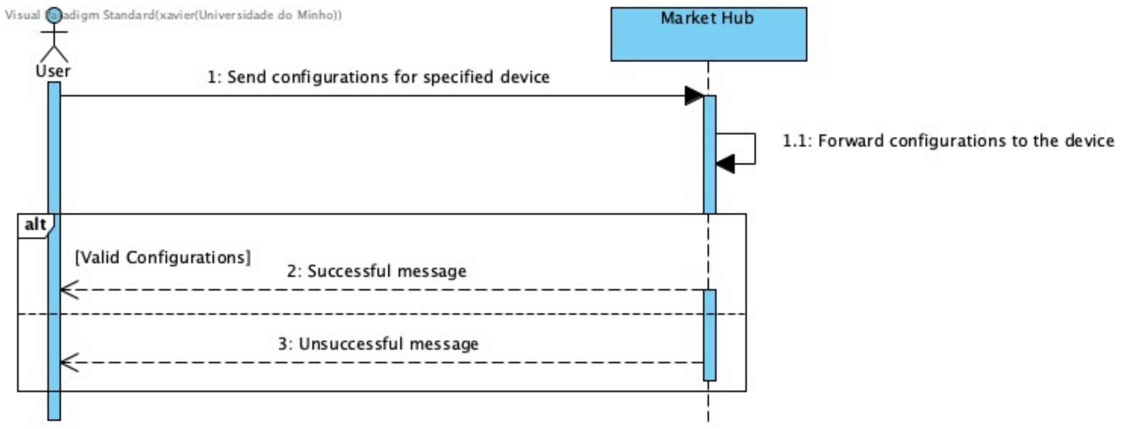
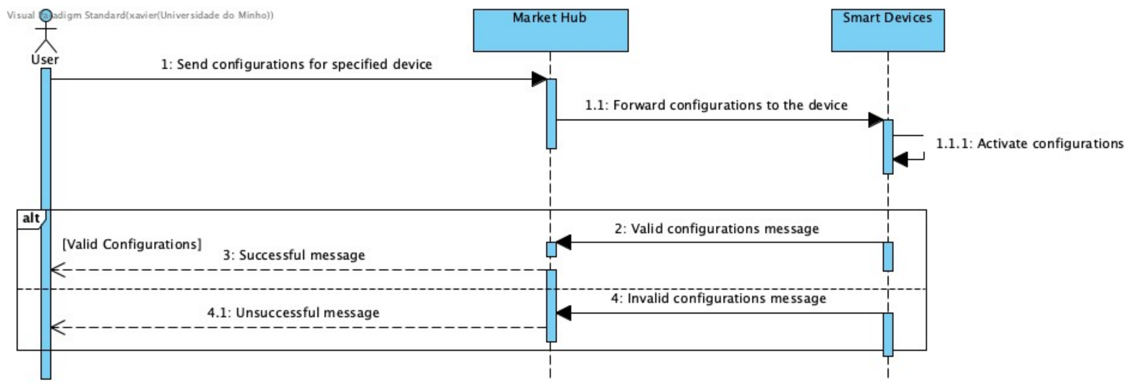
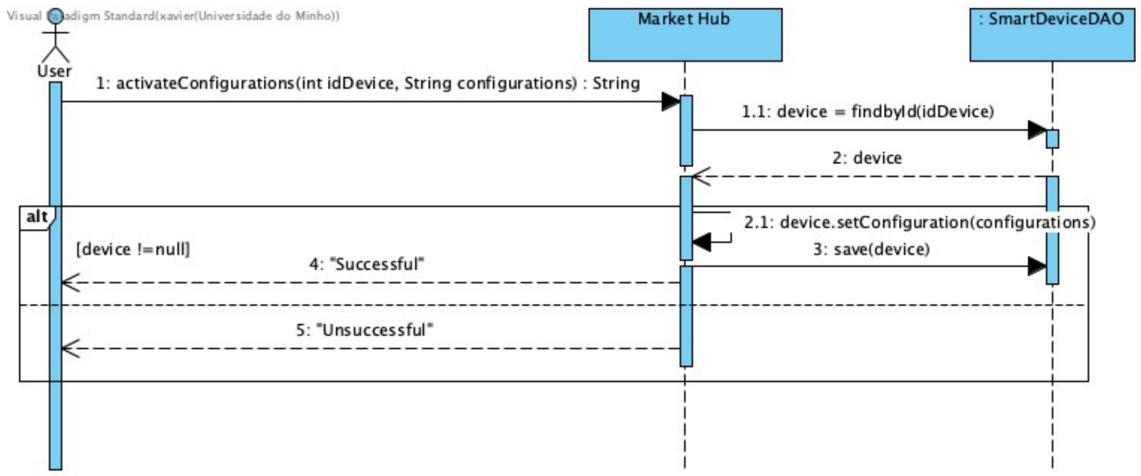

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Send configurations for specified device
1.1: Forward configurations to the device
alt
[Valid Configurations]
2: Successful message
3: Unsuccessful message

Figure 59: Configure Device System Sequence Diagram



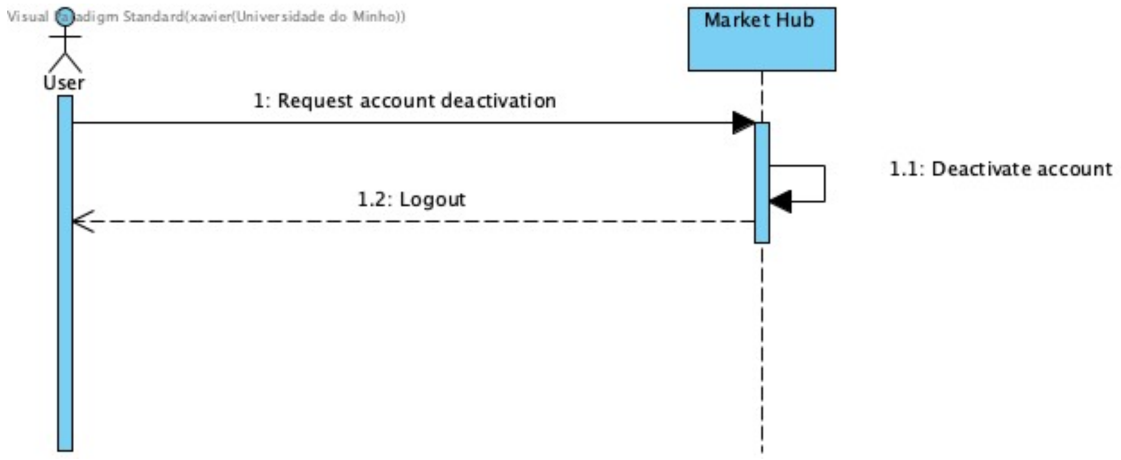
Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
Smart Devices
1: Send configurations for specified device
1.1: Forward configurations to the device
1.1.1: Activate configurations
alt
[Valid Configurations]
2: Valid configurations message
3: Successful message
4: Invalid configurations message
4.1: Unsuccessful message

Figure 60: Configure Device Subsystem Sequence Diagram



Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: SmartDeviceDAO
1: activateConfigurations(int idDevice, String configurations) : String
1.1: device = findbyId(idDevice)
2: device
alt
2.1: device.setConfiguration(configurations)
[device !=null]
3: save(device)
4: "Successful"
5: "Unsuccessful"

Figure 61:Configure Device Implementation Sequence Diagram



Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Request account deactivation
1.1: Deactivate account
1.2: Logout

Figure 62: Deactivate Account System Sequence Diagram

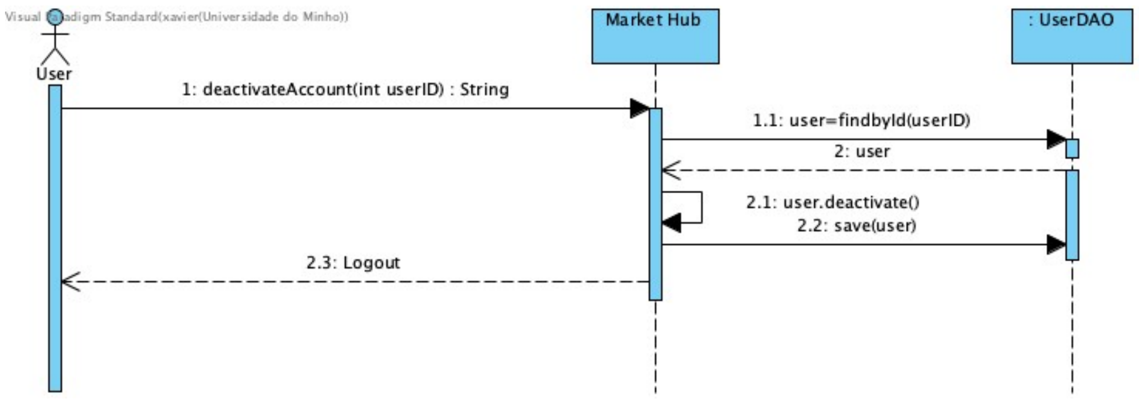

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: UserDAO
1: deactivateAccount(int userID) : String
1.1: user=findbyId(userID)
2: user
2.1: user.deactivate()
2.2: save(user)
2.3: Logout

Figure 63: Deactivate Account Implementation Sequence Diagram

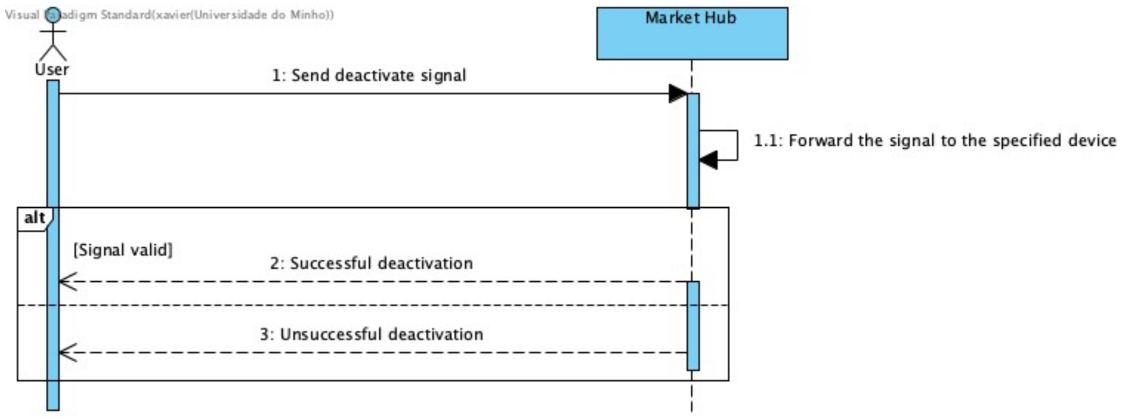

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Send deactivate signal
1.1: Forward the signal to the specified device
alt
[Signal valid]
2: Successful deactivation
3: Unsuccessful deactivation

Figure 64: Deactivate Smart Device System Sequence Diagram

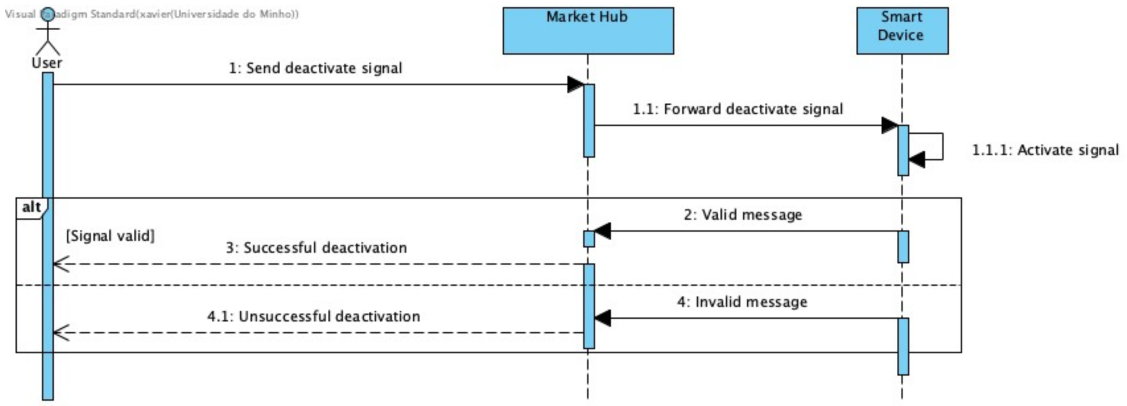

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
Smart Device
1: Send deactivate signal
1.1: Forward deactivate signal
1.1.1: Activate signal
alt
[Signal valid]
2: Valid message
3: Successful deactivation
4: Invalid message
4.1: Unsuccessful deactivation

Figure 65: Deactivate Smart Device Subsystem Sequence Diagram

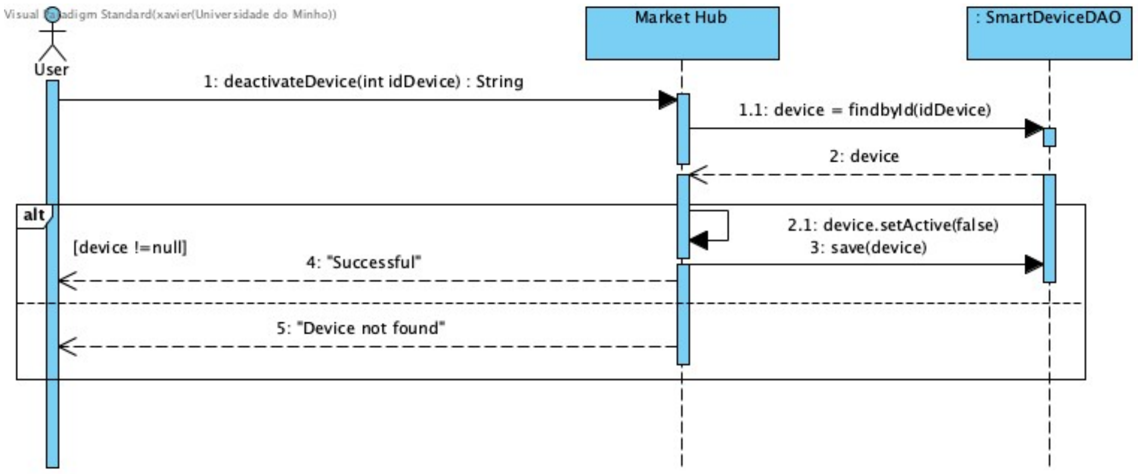

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: SmartDeviceDAO
1: deactivateDevice(int idDevice) : String
1.1: device = findbyId(idDevice)
2: device
alt
[device !=null]
2.1: device.setActive(false)
3: save(device)
4: "Successful"
5: "Device not found"

Figure 66: Deactivate Smart Devices Implementation Sequence Diagram

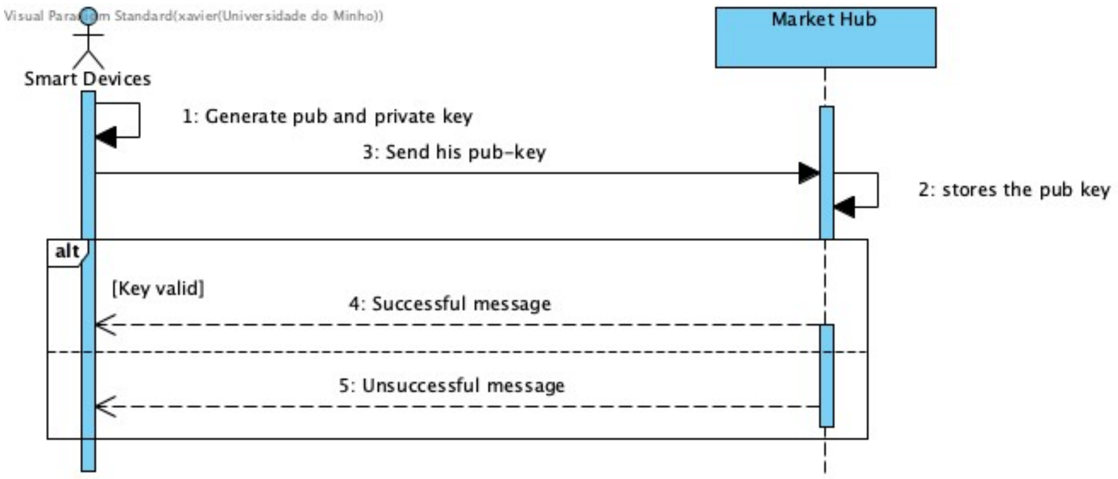


Figure 67: Exchange Smart Devices Keys System Sequence Diagram

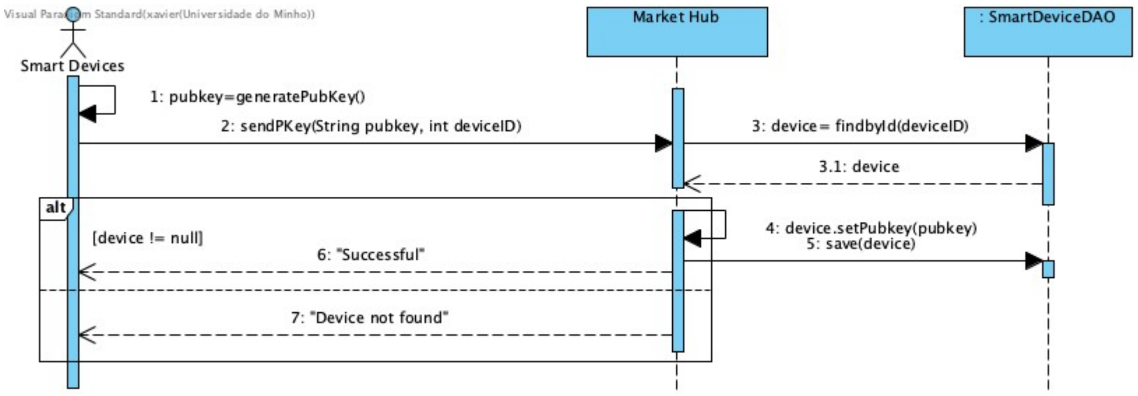

Visual Paradigm Standard(xavier(Universidade do Minho))
Smart Devices
Market Hub
: SmartDeviceDAO
1: pubkey=generatePubKey()
2: sendPKey(String pubkey, int deviceID)
3: device= findbyId(deviceID)
3.1: device
alt
[device != null]
4: device.setPubkey(pubkey)
5: save(device)
6: "Successful"
7: "Device not found"

Figure 68: Exchange Smart Devices Keys Subsystem Sequence Diagram

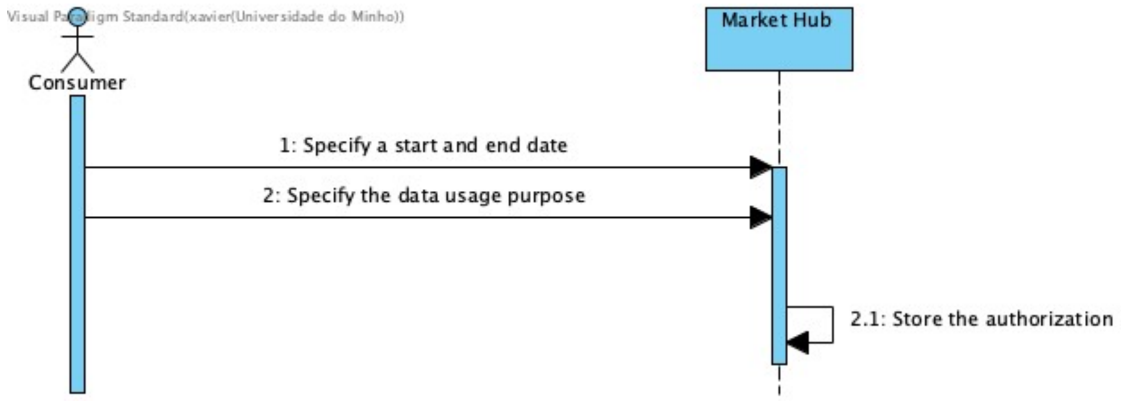

Visual Paradigm Standard(xavier(Universidade do Minho))
Consumer
Market Hub
1: Specify a start and end date
2: Specify the data usage purpose
2.1: Store the authorization

Figure 69: Give Authorization System Sequence Diagram

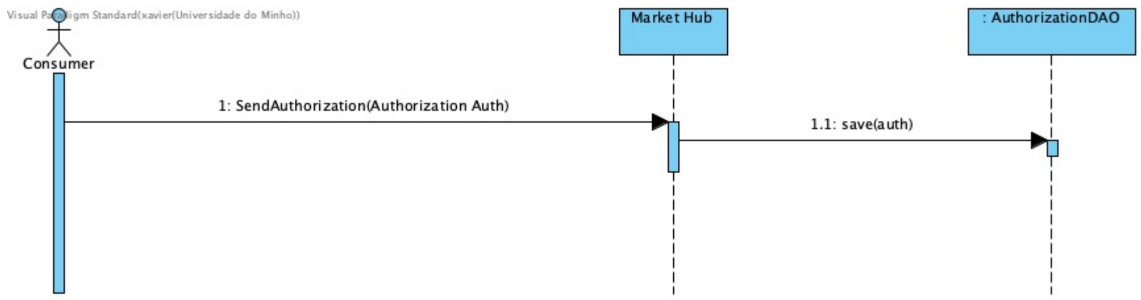

Visual Paradigm Standard(xavier(Universidade do Minho))
Consumer
Market Hub
: AuthorizationDAO
1: SendAuthorization(Authorization Auth)
1.1: save(auth)

Figure 70: Give Authorization Implementation Sequence Diagram

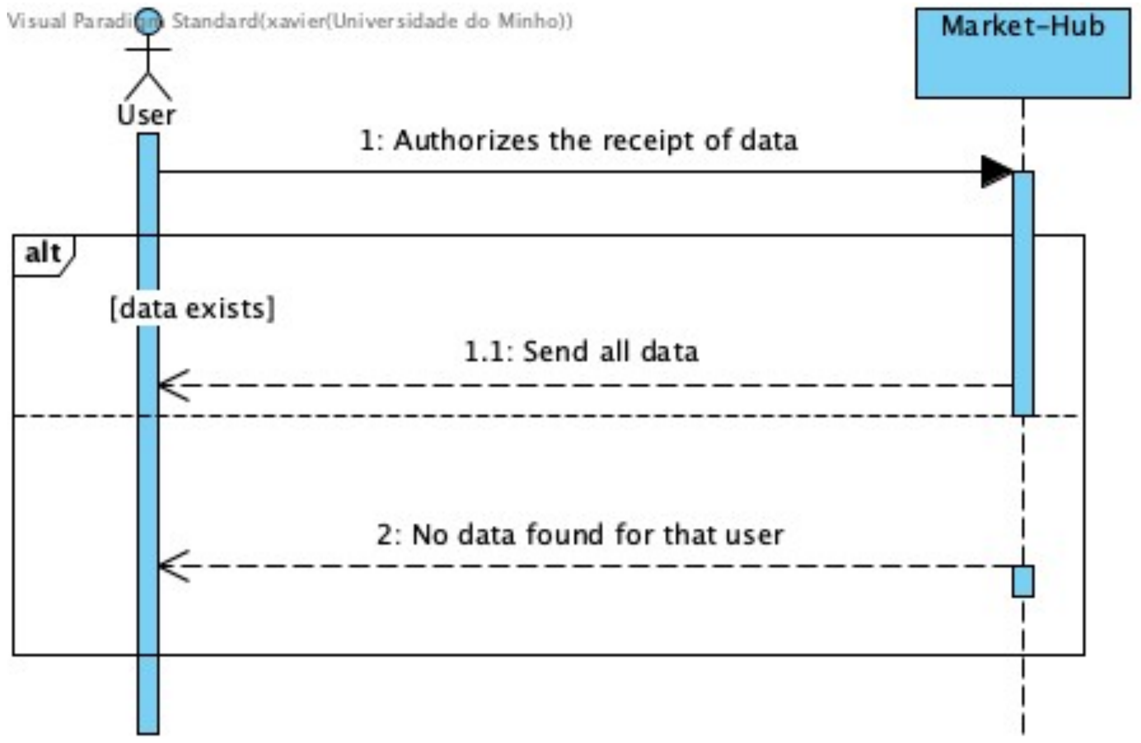

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market-Hub
1: Authorizes the receipt of data
alt
[data exists]
1.1: Send all data
2: No data found for that user

Figure 71: Receive Data System Sequence Diagram

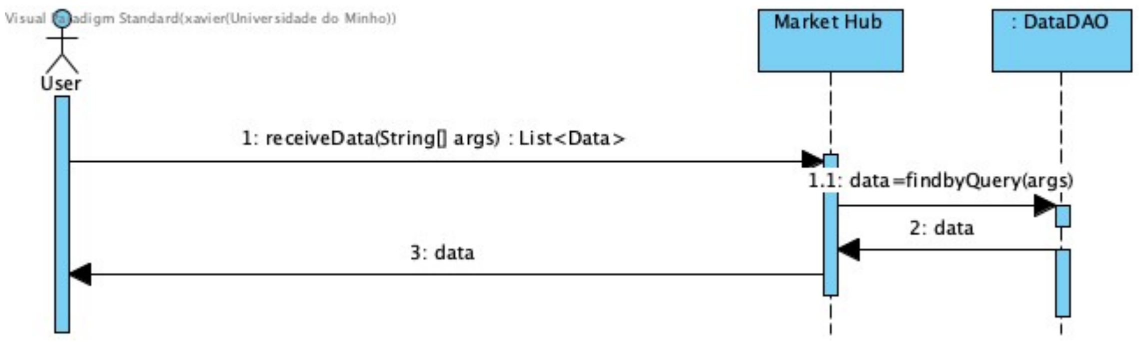

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: DataDAO
1: receiveData(String[] args) : List<Data>
1.1: data=findbyQuery(args)
2: data
3: data

Figure 72:Receive Data Implementation Sequence Diagram

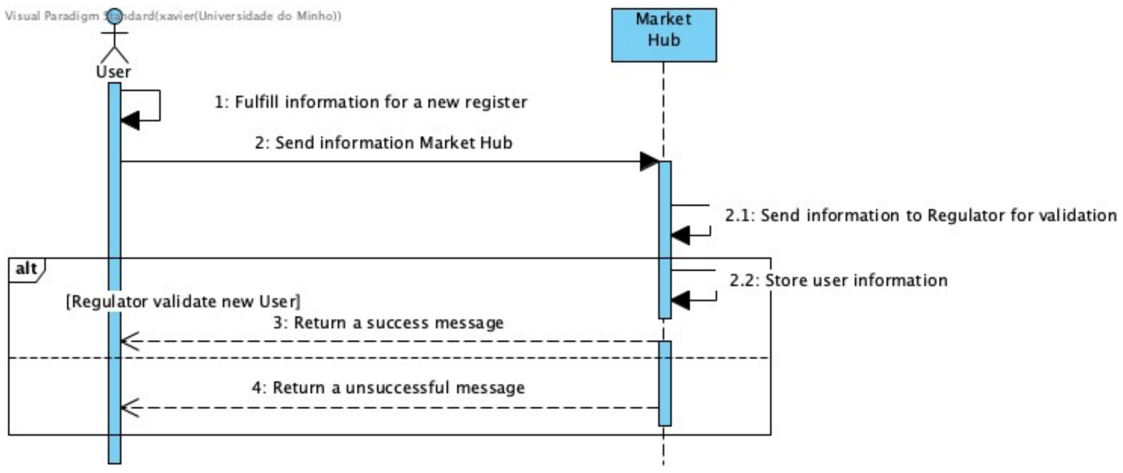

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Fulfill information for a new register
2: Send information Market Hub
2.1: Send information to Regulator for validation
alt
2.2: Store user information
[Regulator validate new User]
3: Return a success message
4: Return a unsuccessful message

Figure 73: Registration System Sequence Diagram

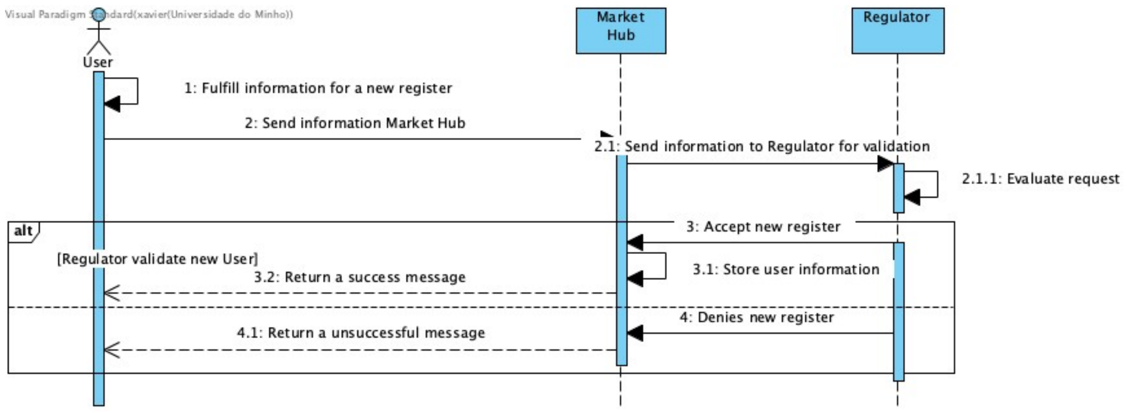

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
Regulator
1: Fulfill information for a new register
2: Send information Market Hub
2.1: Send information to Regulator for validation
2.1.1: Evaluate request
alt
3: Accept new register
[Regulator validate new User]
3.1: Store user information
3.2: Return a success message
4: Denies new register
4.1: Return a unsuccessful message

Figure 74: Registration Subystem Sequence Diagram

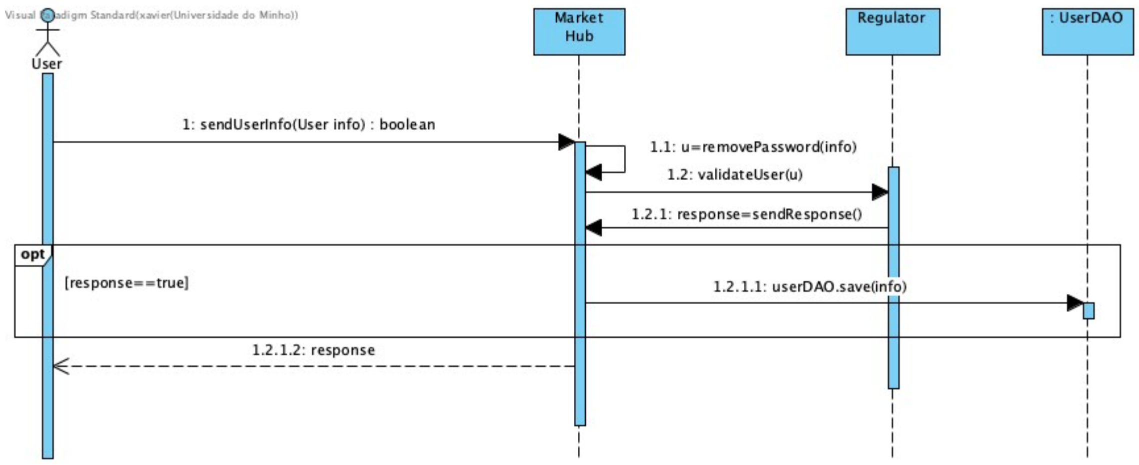

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
Regulator
: UserDAO
1: sendUserInfo(User info) : boolean
1.1: u=removePassword(info)
1.2: validateUser(u)
1.2.1: response=sendResponse()
opt
[response==true]
1.2.1.1: userDAO.save(info)
1.2.1.2: response

Figure 75: Registration Implementation Sequence Diagram

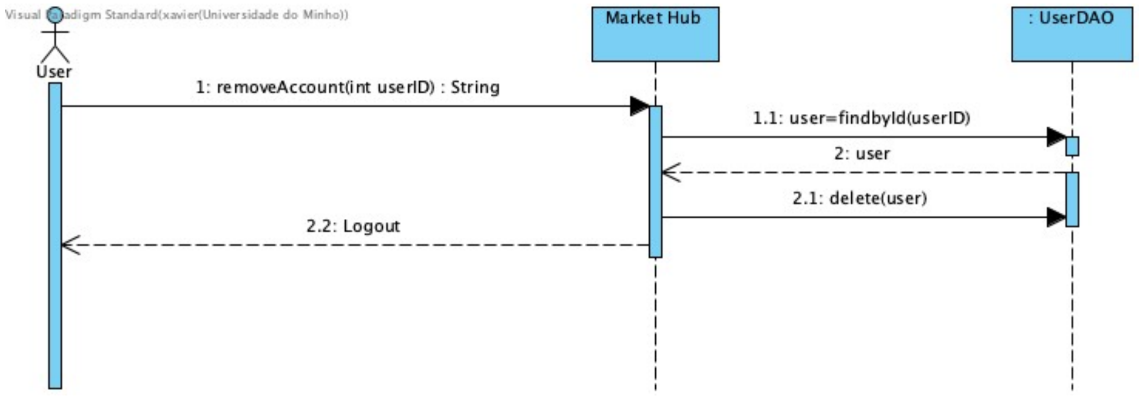

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: UserDAO
1: removeAccount(int userID) : String
1.1: user=findbyId(userID)
2: user
2.1: delete(user)
2.2: Logout

Figure 76: Remove Account Implementation Sequence Diagram

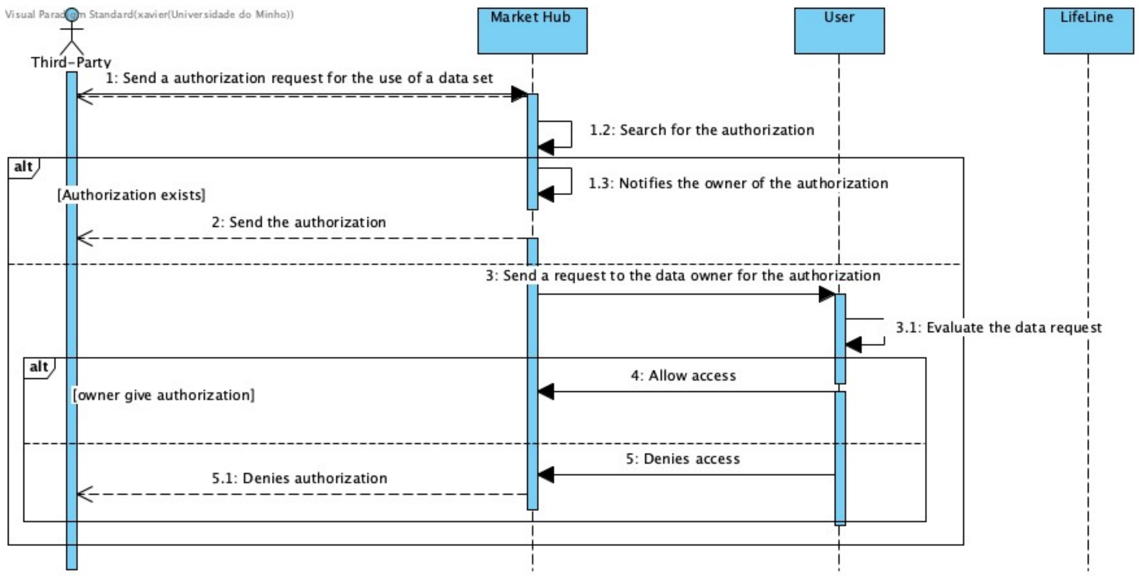

Visual Paradigm Standard(xavier(Universidade do Minho))
Third-Party
Market Hub
User
LifeLine
1: Send a authorization request for the use of a data set
1.2: Search for the authorization
alt
[Authorization exists]
1.3: Notifies the owner of the authorization
2: Send the authorization
3: Send a request to the data owner for the authorization
3.1: Evaluate the data request
alt
[owner give authorization]
4: Allow access
5: Denies access
5.1: Denies authorization

Figure 77: Get Authorization Subsystem Sequence Diagram

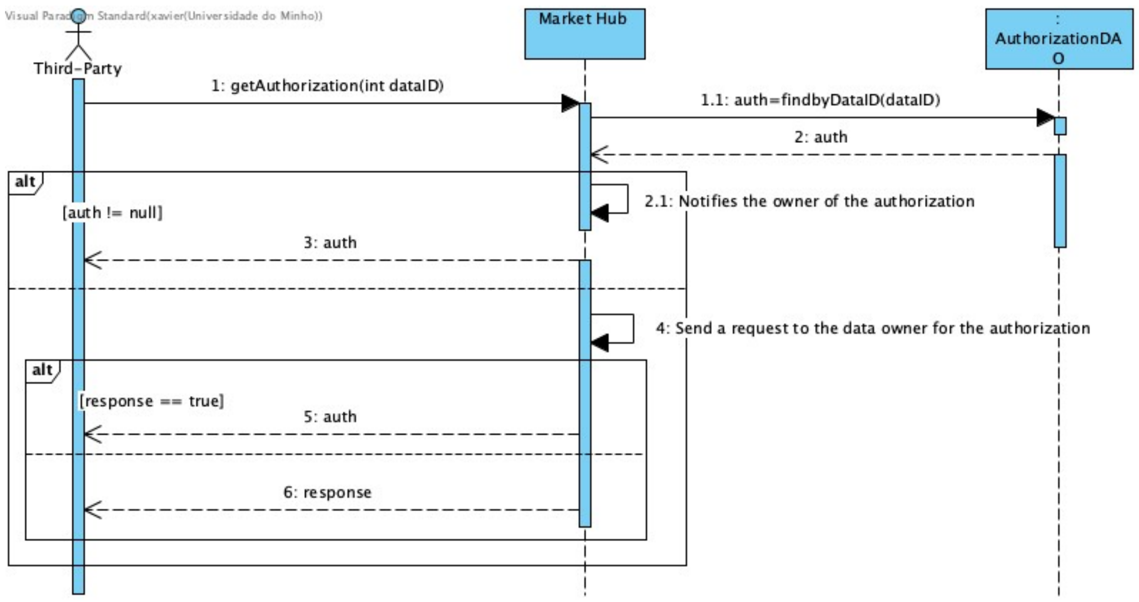

Visual Paradigm Standard(xavier(Universidade do Minho))
Third-Party
Market Hub
: AuthorizationDAO
1: getAuthorization(int dataID)
1.1: auth=findbyDataID(dataID)
2: auth
alt
[auth != null]
2.1: Notifies the owner of the authorization
3: auth
4: Send a request to the data owner for the authorization
alt
[response == true]
5: auth
6: response

Figure 78: Get Authorization Implementation Sequence Diagram

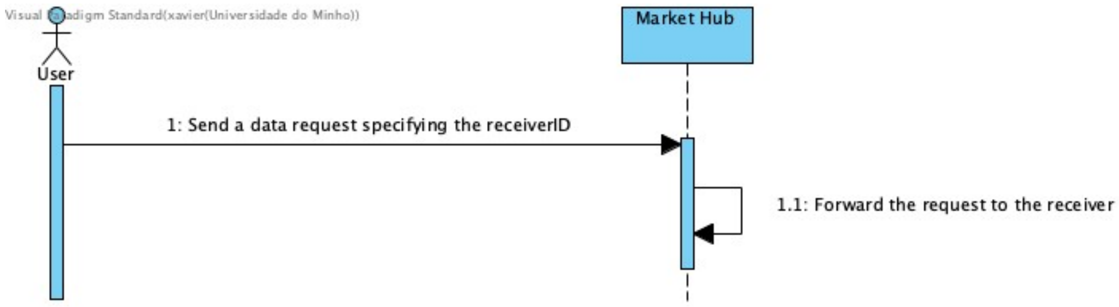

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
1: Send a data request specifying the receiverID
1.1: Forward the request to the receiver

Figure 79: Send Data System Sequence Diagram

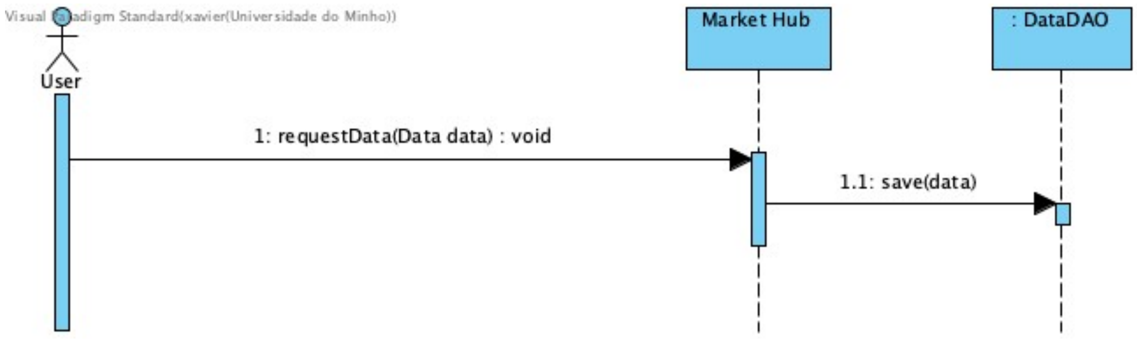

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: DataDAO
1: requestData(Data data) : void
1.1: save(data)

Figure 80: Send Data Implementation Sequence Diagram

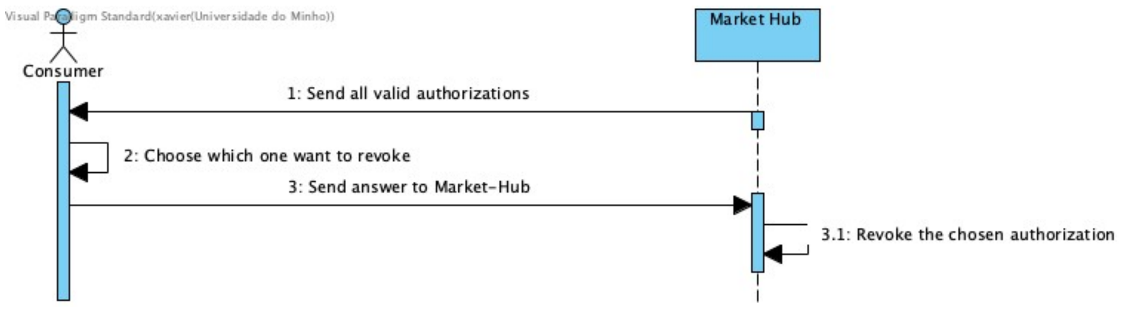

Visual Paradigm Standard(xavier(Universidade do Minho))
Consumer
Market Hub
1: Send all valid authorizations
2: Choose which one want to revoke
3: Send answer to Market-Hub
3.1: Revoke the chosen authorization

Figure 81: Revoke Authorization Subsystem Sequence Diagram

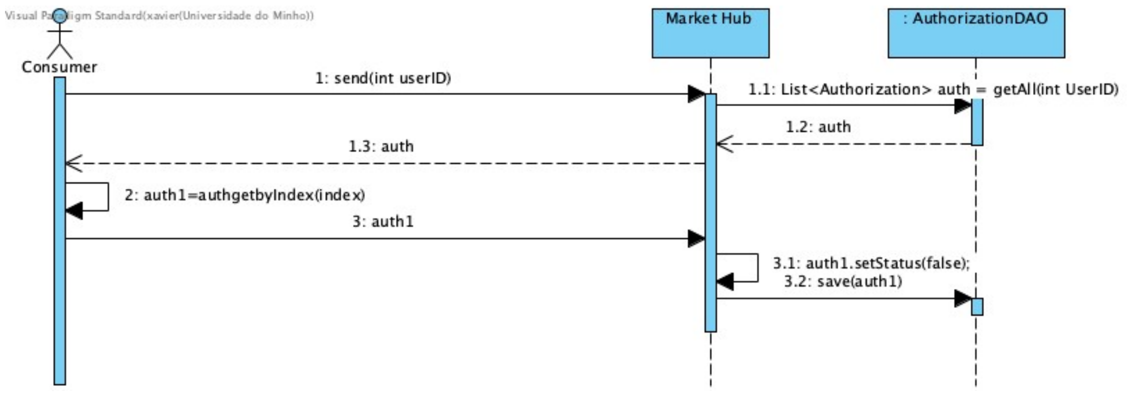

Visual Paradigm Standard(xavier(Universidade do Minho))
Consumer
Market Hub
: AuthorizationDAO
1: send(int userID)
1.1: List<Authorization> auth = getAll(int UserID)
1.2: auth
1.3: auth
2: auth1=authgetbyIndex(index)
3: auth1
3.1: auth1.setStatus(false);
3.2: save(auth1)

Figure 82: Revoke Authorization Implementation Sequence Diagram

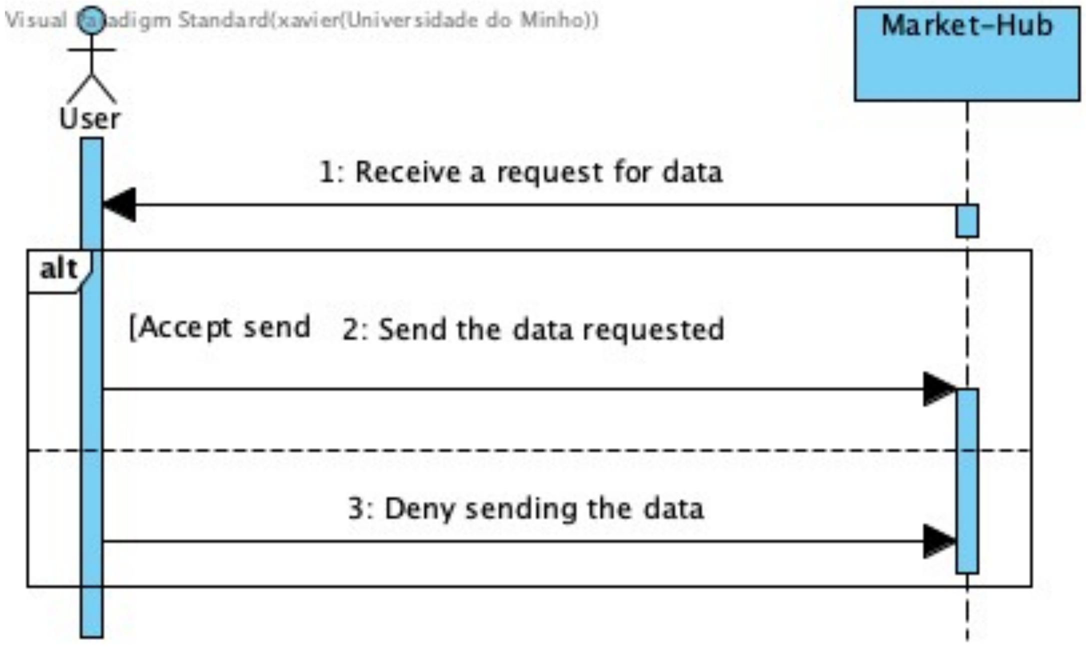

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market-Hub
1: Receive a request for data
alt
[Accept send
2: Send the data requested
3: Deny sending the data

Figure 83: Request Data System Sequence Diagram

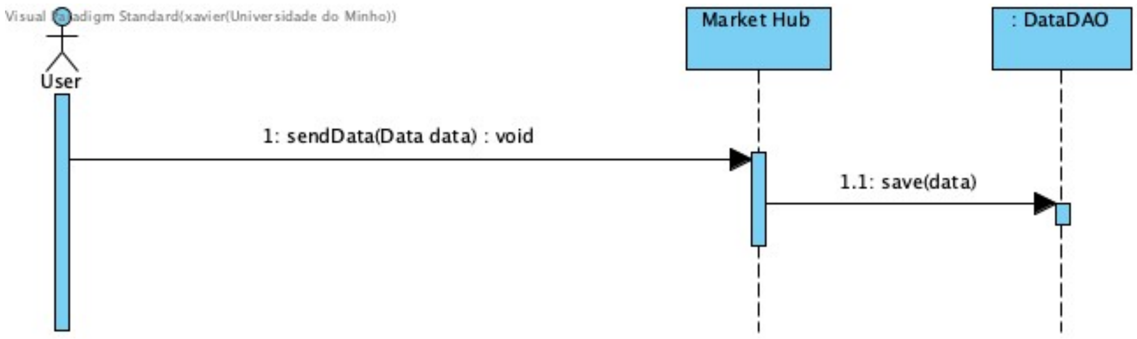

Visual Paradigm Standard(xavier(Universidade do Minho))
User
Market Hub
: DataDAO
1: sendData(Data data) : void
1.1: save(data)

Figure 84: Send Data Implementation Sequence Diagram

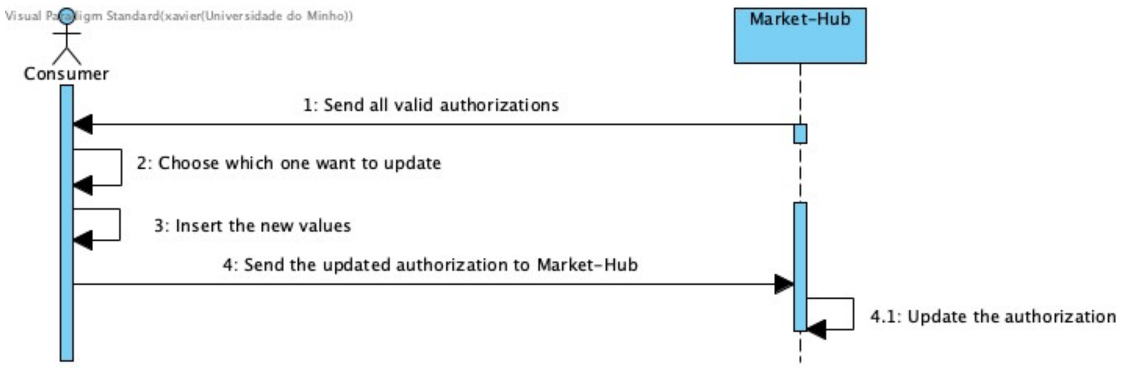

Visual Paradigm Standard(xavier(Universidade do Minho))
Consumer
Market-Hub
1: Send all valid authorizations
2: Choose which one want to update
3: Insert the new values
4: Send the updated authorization to Market-Hub
4.1: Update the authorization

Figure 85: Update Authorization System Sequence Diagram

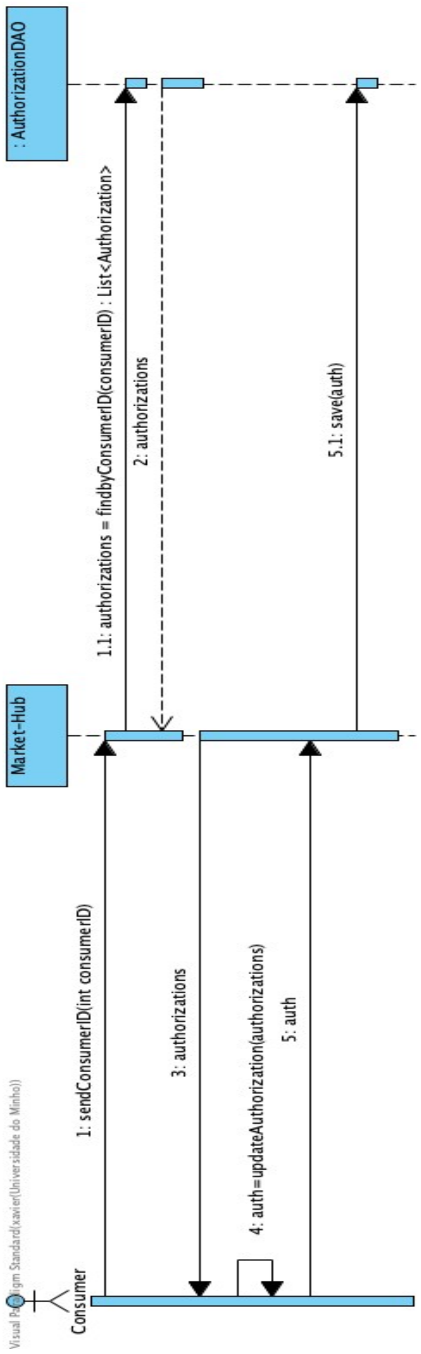

: AuthorizationDAO
Market-Hub
Consumer
Visual Paradigm Standard(xavier(Universidade do Minho))
1: sendConsumerID(int consumerID)
1.1: authorizations = findbyConsumerID(consumerID) : List<Authorization>
2: authorizations
3: authorizations
4: auth=updateAuthorization(authorizations)
5: auth
5.1: save(auth)

Figure 86: Update Authorization Implementation Sequence Diagram

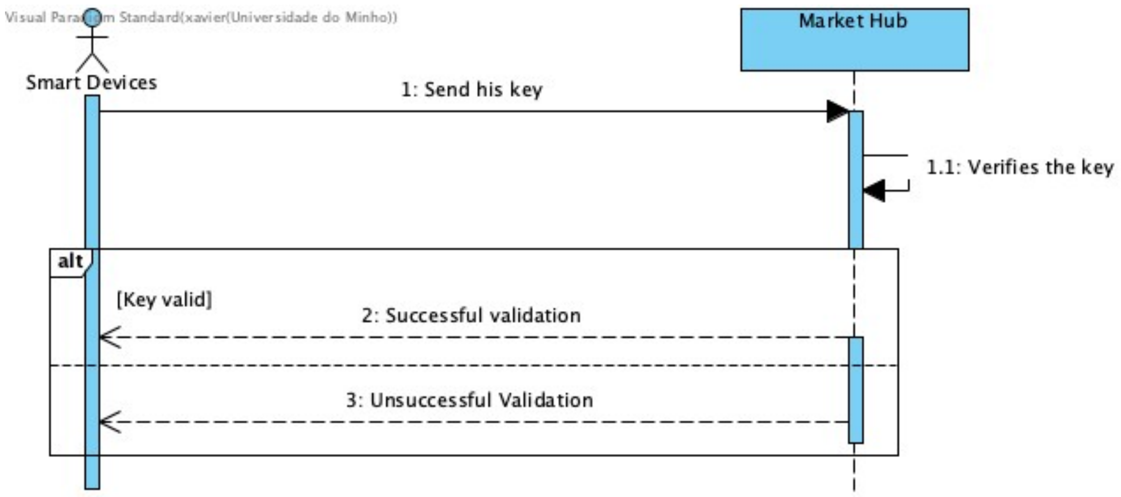
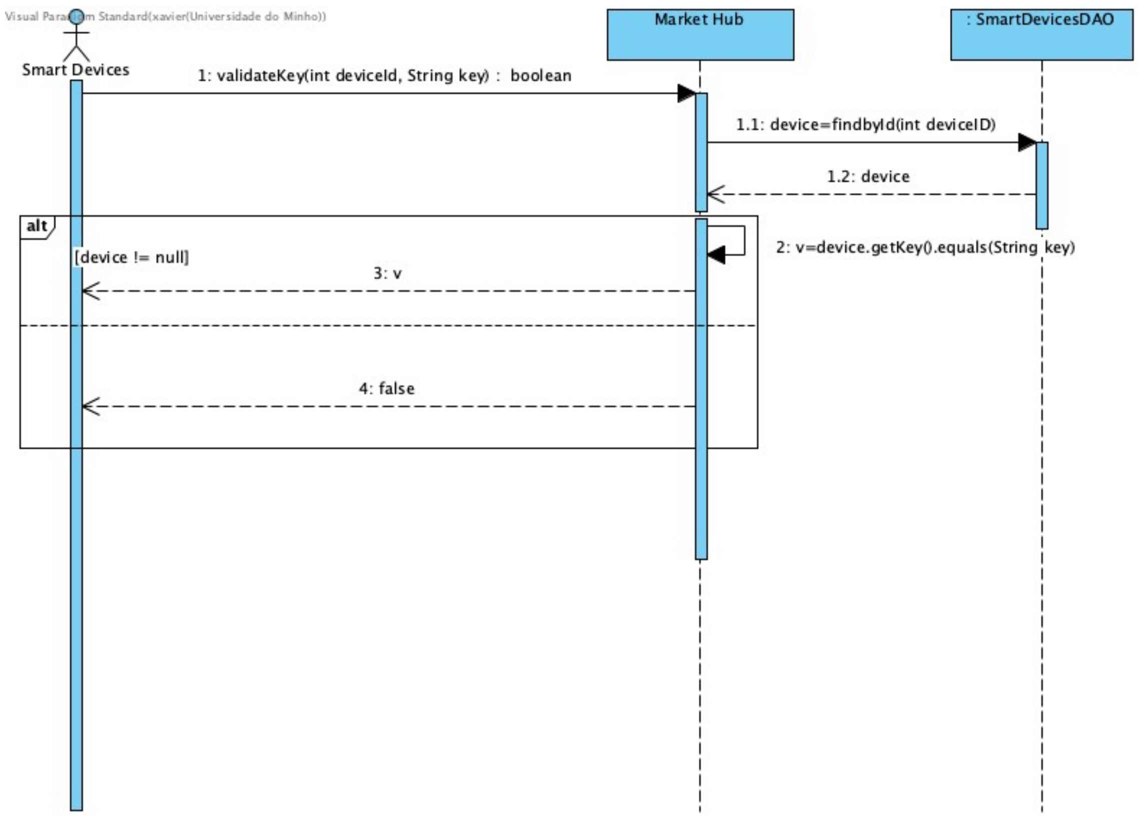


Figure 87: Validate Key System Sequence Diagram



Visual Paradigm Standard(xavier(Universidade do Minho))
Smart Devices
Market Hub
: SmartDevicesDAO
1: validateKey(int deviceId, String key) : boolean
1.1: device=findbyId(int deviceID)
1.2: device
alt
2: v=device.getKey().equals(String key)
[device != null]
3: v
4: false

Figure 88: Validate Key Implementation Sequence Diagram

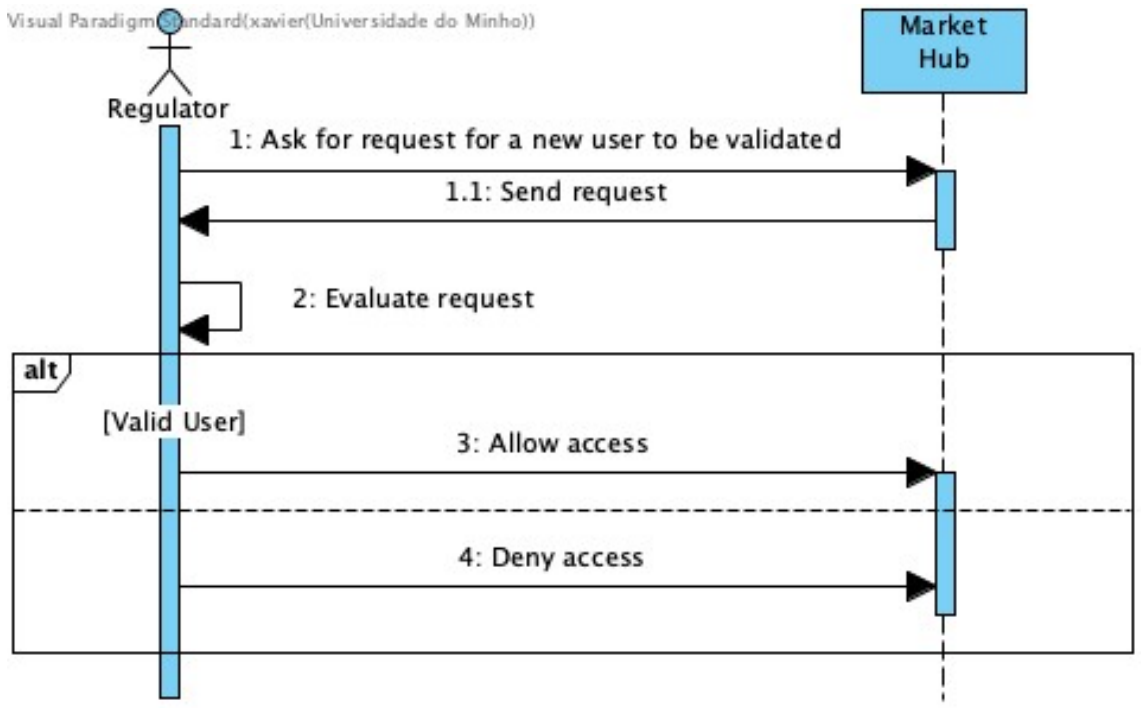

Visual Paradigm Standard(xavier(Universidade do Minho))
Regulator
Market Hub
1: Ask for request for a new user to be validated
1.1: Send request
2: Evaluate request
alt
[Valid User]
3: Allow access
4: Deny access

Figure 89: Validate User System Sequence Diagram