UNIVERSITÀ DEGLI STUDI DI PAVIA
FACOLTÀ DI INGEGNERIA
DIPARTIMENTO DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

DOTTORATO DI RICERCA IN BIOINGEGNERIA E BIOINFORMATICA
XXX CICLO – 2019

PRODUCT LINE ARCHITECTURE FOR HADRONTHERAPY CONTROL SYSTEM: APPLICATIONS DEVELOPMENT AND CERTIFICATION

PhD Thesis by
Carlos Eduardo Fernandez Afonso

Advisor:
Prof. Cristiana Larizza

Industry Supervisor:
Eng. Luigi Casalegno

PhD Program Chair:
Prof. Paolo Magni

Acknowledgments

Firstly, I would also like to express my gratitude to Dr. Carlsten Welsh for organizing the Optimization of Medical Accelerators project, and the European Union’s Horizon 2020 research and innovation program for funding this research project[1].

I would like to express my gratitude to my PhD supervisor, Prof. Dr. Cristina Larizza, for all support and academic advice throughout these years. I would also like to thank my industrial supervisor, Eng. Luigi Casalegno, who has provided abundant guidance during this project, and performed an active role in the design and development on the project that is the basis of this thesis.

In addition to my supervisors, I would like to thank Dr. Monica Necchi, for all encouraging words and actions during the project, as well as the management of the project.

I would like to thank my colleagues at CNAO for the all the time spent together, and the discussions we had on the various subjects related to our work. In addition, I would like to thank the other OMA project fellows for all discussions we had in our shared meetings and trainings.

Finally, I would like to express my sincere gratitude towards my family and friends, for supporting me and being there for me during this project, and that, without them, this would not have been possible.

[1] This project has received funding from the European Union’s Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement 675265

Abstract

Hadrontherapy is the treatment of cancer with charged ion beams. As the charged ion beams used in hadrontherapy are required to be accelerated to very large energies, the particle accelerators used in this treatment are complex and composed of several sub-systems. As a result, control systems are employed for the supervision and control of these accelerators.

Currently, The Italian National Hadrontherapy Facility (CNAO) has the objective of modernizing one of the software environments of its control system. Such a project would allow for the integration of new types of devices into the control system, such as mobile devices, as well as introducing newer technologies into the environment.

In order to achieve this, this work began with the requirement analysis and definition of a product line architecture for applications of the upgraded control system environment. The product line architecture focuses on reliability, maintainability, and ease of compliance with medical software certification directives. This was followed by the design and development of several software services aimed at allowing the communication of the environment's applications and other components of the control system, such as remote file access, relational data access, and OPC-UA. In addition, several libraries and tools have been developed to support the development of future control system applications, following the defined product line architecture.

Lastly, a pilot application was created using the tools developed during this work, and preliminary results of a cross-environment integration project are presented. The approach followed in this work is later evaluated by comparing the developed tools to their legacy counterparts, as well as estimating the impact of future applications following the defined product line architecture.

Contents

Abreviations

ANL - Argonne National Laboratory
API - Application Program Interface
CA - Certification Authority
CE - Conformité Européenne
CERN - Conseil Européen pour la Recherche Nucléaire
CLI - Command-Line Interface
CNAO - Centro Nazionale di Adroterapia Oncologica
CORBA - Common Object Request Broker Architecture
BLOB – Binary Large Object
CRUD – Create Read Update Delete
DAO – Data Access Object
DCS - Distributed Control System
DLL - Dynamic-Link Library
ELI - Extreme Light Infrastructure
EPICS - Experimental Physics and Industrial Control System
ESRF - European Synchrotron Radiation Facility
FDAS - File Data Access Service
FEC - Front End Computer
FPGA - Field-programmable gate array
FTP – File Transfer Protocol
GUI – Graphical User Interface
GSI – GSI Helmholtz Centre for Heavy Ion Research
GTA - Ground Test Accelerator
GTACS - Ground Test Accelerator Control System
IBA – Ion Beam Applications
ICALEPCS - International Conference on Accelerator and Large Experimental Physics Control Systems
IDE - Integrated Development Environment
INFN - Istituto Nazionale di Fisica Nucleare
JSON - JavaScript Object Notation
JWT - JSON Web Token
LAN - Local Area Network
LANL - Los Alamos National Laboratory
LDAP - Lightweight Directory Access Protocol
LHC - Large Hadron Collider
MVVM - Model-View View Model
ORM - Object Relational Mapping
OPC - Open Platform Communications
OPC-UA - Open Platform Communications Unified Architecture
PCI - Peripheral Component Interconnect

POCO	Plain Old C# Object / Plain Old Common language runtime Object.
PXI	PCI eXtensions for Instrumentation
RBE	Relative Biological Effectiveness
RDAS	Relational Data Access Service
REST	Representational state transfer.
RF	Radio Frequency
SCADA	Supervisory Control and Data Acquisition
SKA	Square Kilometre Array
SSC	Superconductive Super Collider
SQL	Structured Query Language
SOLEIL	Source optimisée de lumière d'énergie intermédiaire du LURE
TCP	Transmission Control Protocol
UML	Unified markup language
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Optimization of Medical Accelerators Project

Particle therapy is a cancer treatment method that is becoming increasingly more prominent. In contrast to radiotherapy, in which radiation is used to damage cancer cells, in particle therapy ions are accelerated to deposit energy in the cells. The main advantage of hadrontherapy over radiotherapy is the former's dose deposition profile, which allows for lower dose deposition in healthy tissue surrounding the targeted tumor. However, hadrontherapy requires the use of a particle accelerator of large dimensions and complexity. The large quantity of equipment involved in generating the particle beam and applying it in patient treatment requires the facility to have an elaborate control system.

The Optimization of Medical Accelerators (OMA) project is a European Training Network funded by the European Union's Horizon 2020 research and innovation programme. This project, started in 2016, joins several institutions, such as universities, research centers, and hadrontherapy facilities, to perform projects under a common set of goals. The network created by the project, which contains more than 30 entities, is composed of beneficiaries, partner organizations, and adjunct partners.

The goals of the network are separated into a total of three work packages. This work is part of the last work package, which is entitled "Optimization of facilities".

When writing the proposal, each beneficiary organization proposed a project, with the scope aligned with one of the work packages, and later received funds to hire a research fellow to work on the project for a duration of three years.

The Centro Nazionale di Adroterapia Oncologica (CNAO) was in the position of hosting two of the OMA network's projects. The OMA project entitled 'Light ion therapy software for data exchange' is the one underlying the work performed in the context thesis.

The goals of the project, as initially written, were "creating a common software bus that shall enable any present and future package to easily interconnect in a complex and widely distributed hadron therapy facility environment" as well as "the design and development of libraries to support these protocols and enable to automatically connect and operate devices". In summary, the original vision for the project was based around designing the intercommunication of software applications in the upgraded configuration and support environment, as well as several libraries to support the new software applications in this environment.

1.2 Motivation

As the software components of the control system were first designed in 2003 and have been in operation for several years, applications of the configuration and support environment were incompatible with several newer software features. By performing a top-down technological upgrade of the environment, the CNAO facility aims at incorporating several newly developed libraries and frameworks to the control system, including multi-platform software targeting. Incorporating mobile devices into the control system allows for the addition of several new control system features not possible previously. For example, control room operators will be allowed to briefly step out of the control room, while being able to continue to monitoring of the accelerator system.

While planning the environment's technology upgrade, the opportunity has been seized to develop solutions in order to improve control system in areas such as application maintainability, multi-platform execution, security, and medical certification.

Finally, extensive study has been performed regarding the software architectures for accelerator control systems, and in the development of reusable solutions for technical requirements of these systems. In this work, we combine the development of reusable software elements and the selection of architectural patterns in software applications of the control system in order to facilitate the integration of these software elements. As a result, we expect that this work provides a step forward on the application of software engineering techniques in the medical accelerator control system domain.

1.3 Objectives

The main objective of this work is planning and performing the upgrade of a control system software environment of a hadrontherapy facility. The software in this environment performs a support role in the medical facility, and is used for tasks such as configuration of the accelerator systems, and supporting the clinical workflow. As part of the upgrade, new technologies were chosen for the environment, and a product line architecture was designed to define the general architecture of future applications. In addition, services to be used by future applications were designed and developed.

The objectives of the work were as follows:

- Requirement analysis of the upgraded configuration and support environment. Definition of objectives for the new environment.
- Definition of a product line architecture to guide the design of the applications in the upgraded environment.
- Design and development of libraries and services to perform the operations to be standardized in the configuration and support environment.
- Development of several graphical wizard generators to create configurable skeleton applications, which follow the product line architecture.
- Investigation on how to make use of the product line approach to aid the medical software certification of the upgraded applications.
- Development of a pilot application, documentation and discussion of reusability of the developed components, and impact of the project in the control system.

1.4 Organization of thesis

The thesis is organized into eight chapters. The remaining chapters are described as it follows:

- **Chapter 2, Background:** Chapter 2 initially provides a brief description of the physics concepts used in particle therapy, and the history of the field. Later, the field of accelerator control systems is introduced and previous accelerator control systems designs are presented. After discussing the field in general, we describe the control system at the CNAO facility.
- **Chapter 3, Upgrade of the Configuration and Support Environment:** This chapter begins with the requirement analysis performed to design the upgraded configuration and support environment. Later, the product line approach for the work is described in detail, as well as the designed architecture for the applications of the environment.
- **Chapter 4, Standardization of services in the Configuration and support environment:** The product line architecture envisages the presence of standard services that are to be used by the applications. This chapter describes the design and development of the standard services in the upgraded environment.
- **Chapter 5, Implementing the basis of the product line architecture:** A family of frameworks and wizard code generation tools was designed for supporting the product line approach upgrade of the environment. In Chapter 5, the development and usage of the frameworks and wizard generators is presented.
- **Chapter 6, Towards certification of control system applications:** The design of the product line architecture was heavily

influenced by the mandatory medical software certification processes that applications in the upgraded environment must undergo. Chapter 6, begins with a description of the certification process. Additionally, it is explained how the design tactics adopted in the product line architecture to aid the certification process is detailed. Finally, a discussion of the expected results of these tactics is presented.

- **Chapter 7, Results and evaluation:** The Chapter 7 describes the results of the work performed in the thesis. A pilot application of the upgraded environment is presented, as well as the early results of the integration project. Later, we evaluate the work performed, focusing both on the impact of individual components, as well as on the contribution of the upgraded environment to the control system as a whole.
- **Chapter 8, Conclusion and future work:** Chapter 8 concludes the thesis, summarizing the work performed and the achieved results. Afterwards, there is a brief description of the expected continuation of this work, as well as areas where it can be expanded upon.

Chapter 2

Background

Large particle accelerators, whether medical or research oriented, are complex systems that can only be developed by combining the efforts of personnel from several heterogeneous fields. When designing a complex accelerator, the earliest obtainable architecturally significant requirements are originated by the users of the system, such as researchers performing scientific experiments, or healthcare providers in the treatment of patients. These initial system requirements have to be analysed by personnel knowledgeable in particle physics and the domain of accelerator design.

As the accelerator is composed of several systems, such as accelerator cavities, communication networks between components, multiple types of sensors, as well as computation software and hardware that perform data processing, and equipment control. In order to flesh out the initial accelerator design into a concrete design, professionals from several fields of engineering participate in this process. Additionally, other important stakeholders also need to be consulted, such as accelerator operators, and maintenance staff.

As a control system developer, it is essential to maintain communication channels to the various stakeholders. Additionally, in order to analyse various requirements, and obtain iterative feedback during the development process, basic domain knowledge of hadrontherapy and accelerator design is required for control system developers.

In this chapter, we present a short description of several research and industrial topics adjacent to the field of medical accelerator control systems. This chapter begins with a short introduction on particle accelerator and hadrontherapy, including the basics of hadrontherapy cancer treatment, and the history of the medical procedure. Common accelerator designs, as well as a comparison of the effects of different particles are subsequently described in this chapter. Later, we present the most widely adopted control system architectures, as well as their historical background. Finally, we finalize with a description of commonly used software frameworks for developing accelerator control systems.

2.1 Hadrontherapy

Hadrontherapy is the general term given to the usage of particle beams for patient treatment. This term encompasses the usage of a large variety of particles, such as neutrons, protons, and charged ions (e.g. helium, boron, oxygen, and carbon ions) [1]. During treatment, the particles are accelerated using a particle accelerator, and delivered into the patient's tissue, where they deposit energy. In this section, a review of hadrontherapy is presented, including a brief overview of the history of hadrontherapy. Afterwards, a brief description of the supporting equipment necessary for the operation of hadrontherapy will be presented.

Research into Hadrontherapy began in 1930s, by the brothers Earnest and John Lawrence, shortly after the invention of the cyclotron by the former[1]. Research was performed in 1936 on the effects of particle beams compared to x-rays in normal and tumor tissue. This research demonstrated that neutron particle beams had a greater effectiveness at destroying cells than x-rays [2]. Based on the earlier research, the first neutron particle therapy treatments occurred in 1938 for the treatment of cancer [3]. The earlier applications of neutron particle beam resulted in undesirable side effects in the form of dosage to heathy tissue, and the treatment was discontinued in 1948 [4]. In 1954, the first patient was treated with a proton beam. Treatments with other charged ions soon followed, with the usage of helium in 1957, neon ions in 1975, and carbon ions in 1994 [3].

2.1.1 Particle beams

In radiology, beams have different characteristics that depend on their physical properties. Two important beam characteristics are the dose distribution profile, and the biological effect of the particle beam. Dose distribution profile, sometimes referred to as dose depth curve, is a measure of the energy deposited along the distance travelled in a material. The biological effect of a particle beam is usually denoted by the beam's relative biological effectiveness (RBE). The RBE compares the beams' biological response to the biological response caused by exposure to a reference beam, in a scenario where the energy deposited by both beams is the same. Considering an arbitrary deposited dose, a particle beam with higher RBE has more biological effect than a lower RBE beam.

Charged ions lose part of their kinetic energy as they pass through objects such as tissue and water, as a result of nuclear and atomic interactions [5]. The prevalence of collisions and interactions increases as charged particles lose kinetic energy, resulting in a concentration of the amount of interactions in the end of the beam penetration range. Consequently, the largest quantity of dose is deposited at the end of the end of the particle range [5]. This range, where most a large amount of dose is deposited, is named Bragg Peak. Figure 2.1 displays the dose depth curve comparison of several beams [6]. As displayed in the figure, photon, neutron and election beams deposit most of their energy early in their range. Meanwhile, carbon and proton beams deposit most of their energy in the end of their range. The rapid increase in deposited dose, as well as the sharp fall after reaching the peak is the main reason for using charged ion beams in radiotherapy [5].

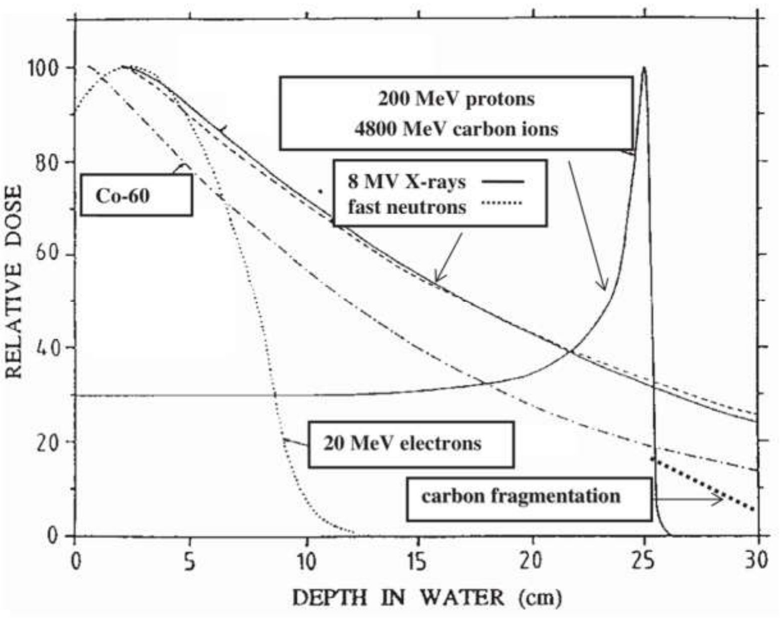


Figure 2.1: Dose depth curve of different beams. Extracted from [6].

While many charged ion beams dose depth curve present Bragg peaks, the shape and relative intensity of the peak varies between charged ions. Carbon beam Bragg peaks are more pronounced than their proton counterparts. Additionally, the ratio of dose deposition at peak/entrance is also higher in carbon beams. However, carbon ions and other heavy ion beams deposit a trailing dose after the Bragg peak. This trailing dose occurs due to fragments produced by nuclear interactions, and is proportionately small enough not to impose severe restrictions to hadrontherapy [5].

2.1.2 Accelerator designs and facilities

Various accelerator designs have been used in hadrontherapy. The most common designs are linear accelerators, cyclotrons, and synchrotrons. Linear electron accelerators are widely used in x-ray radiotherapy, numbering more than 10,000 units worldwide [7]. While proton linear accelerator design has first been proposed in 1989 by Lennox et al. [8], currently there are no hadrontherapy facilities using linear accelerators [9].

There are currently several proposed designs for particle therapy linear accelerators, such as the ADAM company proton linear accelerator design [10], and the TERA foundation carbon cyclotron-to-linear-accelerator design [11]. The arguments for using a linear accelerator in particle therapy are as follows: these accelerators allow for beams with high repetition rates, with over 100 pulses per second, enabling fast energy modulation, as the energy can be adjusted every pulse [3]. Drawbacks of these designs include the accelerator length, and costs.

The most common accelerator design currently used in hadrontherapy facilities is the cyclotron [9], with several companies offering commercial solutions, such as IBA [12], and Varian [13]. In cyclotrons, charged particles are subjected to a constant magnetic field, and accelerated in an acceleration gap controlled via an RF frequency. As the bunched particles are accelerated, the radius of their trajectory increases. Once the radius of the trajectory of the bunch matches the cyclotron extraction window, the bunch passes through the window and is extracted. The resulting extracted bunches from cyclotrons has a periodicity fast enough to be considered a constant beam for the purposes of cancer treatment. Because of their design, cyclotrons produce beams with constant kinetic energy, thus requiring a separate mechanism for energy modulation [3]. The energy modulation is usually performed with an external energy selection system, which places absorbers in the beams trajectory, thus reducing the energy of the beam. This extra energy modulation step affects the beam quality in multiple ways: firstly, the beam shape is modified, and debris is created as result of the collision with the absorber material. Secondly, the energy modulation system process takes a slight amount of time to physically move the absorber wedges, leading to a short delay [3]. Amaldi et al. note that, in cases of high beam energy, attenuation of protons, or of carbon ion beams, the nuclear interactions may cause the energy selection system area to become radioactive [3].

Cyclotrons designs have two widely available variants, the isochronous variant and the synchrocyclotron variant. These variants behave similarly, but unlike the isochronous version where the RF frequency is constant, in synchrocyclotrons the RF frequency decreases during the acceleration.

Another common accelerator design for particle therapy is the synchrotron. In the synchrotron design, the particle beam repeatedly travels a circular path as it is being accelerated. Along the circular path, several magnets perform the bending of the beam and beam compression. Once the particle bunch reaches the desired energy, an extraction mechanism is activated and the beam is delivered. Currently, all carbon ion particle therapy facilities use synchrotron accelerator designs [9]. Table 2.1 presents a list of all currently in operation carbon therapy facilities, as well as their respective location, maximum energy per nucleon, and start of operation date [9].

Loma Linda, the first hospital-based hadrontherapy center started treating patients in 1990, and has treated 13,500 patients with protons from until the end of 2008 using a synchrotron accelerator [3]. Synchrotron designs usually provide low beam periodicity rate, with conventional designs taking between more than a second to reduce the magnetic field, prepare a new beam, and accelerate it to the desired energy. As noted by Amaldi et al. the periodicity of synchrotrons may present issues in the treatment of moving tumors, as it may roughly coincide with patients breathing patterns[3]. Recently, novel designs are being proposed, offering much faster repetition rates for proton synchrotrons [14].

Table 2.1: Table of carbon ion facilities currently in operation. All facilities mentioned use synchrotron accelerator designs. Information from PTCOG website [9].

COUNTRY	WHO, WHERE	MAXIMUM ENERGY PER NUCLEON (MeV)	START OF TREATMENT
Austria	MedAustron, Wiener Neustadt	403/u	2019
China	IMP-CAS, Lanzhou	400/u	2006
China	SPHIC, Shanghai	430/u	2014
China	Heavy Ion Cancer Treatment Center, Wuwei, Gansu	400/u	2019
Germany	HIT, Heidelberg	430/u	2009, 2012
Germany	MIT, Marburg	430/u	2015
Italy	CNAO, Pavia	480/u	2012
Japan	HIMAC, Chiba	800/u	1994, 2017
Japan	HIBMC, Hyogo	320/u	2002
Japan	GHMC, Gunma	400/u	2010
Japan	SAGA-HIMAT, Tosu	400/u	2013
Japan	i-Rock Kanagawa Cancer Center, Yokohama	430/u	2015
Japan	Osaka Heavy Ion Therapy Center, Osaka	430/u	2018

Overall, cyclotron and synchrotron designs are the current norm in field of particle therapy. Cyclotron designs offer benefits such as lower acquisition cost, simpler and less costly operation [5], smaller size, and provide a continuous beam. However, there are currently no fully functioning cyclotrons designs capable of accelerating carbon ions at the energy necessary for hadrontherapy [9]. On the other hand, synchrotron designs are more versatile, allowing for the accelerated energy to be determined every acceleration cycle, and thus not requiring absorbers to modulate energy.

Currently, there are several accelerator designs aimed at addressing issues and improving hadrontherapy, in various development stages. Super cooling technologies have already been proposed and successfully implemented in order to reduce the size and weight of cyclotrons and synchrotrons. IBA is currently designing a super conductive synchrocyclotron aimed at accelerating protons and carbon ions. Finally, several linear accelerator designs have been proposed to provide faster energy modulation without using absorbers, such as by the ADAM company [15], and the TERA foundation [16].

2.2 Control systems

In the field of particle accelerator design, the term control system is used more broadly than in some other industrial engineering domains. According to Müller [17], the role of an accelerator control system is to supervise all devices and subsystems of the accelerator, taking into account the states and transitions of the system. At a software level, an accelerator control system should be able to perform error detection, and follow up by providing proper recovery methods. Additionally, a mapping must be made from all operation requirements of the accelerator into control system operation modes, allowing operators to fulfill the requirements by selecting the appropriate operation modes [17]. Because of these requirements, and the complexity of medical and experimental accelerators, these control systems are composed of hardware, networking components, off-the-shelf, and tailor made software [17].

The ICALEPCS [18] biennial conference, which gathers control system specialists from around the world, interprets the scope of the term "control systems" to include the following:

- "all components or functions, such as processors, interfaces, field-busses, networks, human interfaces, system and application software, algorithms, architectures, databases, etc."
- "all aspects of these components, including engineering, execution methodologies, project management, costs, etc."[18]

2.2.1 DCS and SCADA

A commonly presented question is the difference between a Distributed Control System (DCS) and a Supervisory Control and Data Acquisition system (SCADA). These terms are used to categorize control systems, or parts of control systems. The difficulty of differentiating these terms is often higher in the domain of accelerator control because of the frequent usage of custom software and hardware in control systems in order to achieve the domain's particular requirements.

In the literature, SCADA systems often do not refer to full control systems, but a layer exclusively made of software packages, and that rest above lower layer of a control system. Under this interpretation, a SCADA system main purpose is to acquire data from the lower layer components and provide supervision capabilities [19]. According to Galloway et al. [20], DCSs are similar to SCADA as both communicate with lower layers, and provide a centralized interface for operators. However, the author notes that at a technological level, DCSs are more closely connected to the associated hardware, and use process driven communication rather than event driven [20]. Table 2.2 contains a short summary of distinctions between the two systems.

Table 2.2: Comparison of properties of DCS and SCADA systems [20].

DCS	SCADA
Process driven, continuous stream of measured data	Event driven, often reporting only changes to system
Suited to closely integrated systems	Suited towards less integrated, more independent systems
Does not need to be concerned with data quality due to integration	Often deals with unreliable data, requiring features to deal with unreliability

In the context of this document, we will refer to control systems components and frameworks by the designation given to them by their developers.

2.2.2 The control system "standard model"

When designing new systems, usually the first step is to analyze the implementation of currently available systems, attempting to find solutions for shared requirements. In the accelerator control systems community, as soon as the first custom solutions were finished and documented, similarities were identified leading to categories and models being proposed [17].

In this section, we present general models for designing control systems present in the literature, as well as the requirements and constraints addressed by such models.

The control system "standard model" was proposed by Kuiper in 1991 [21]. This model proposes a separation of concerns into control system tiers. For each tier, hardware components are specified, as well as communication between components, and allocation of software applications. Figure 2.2

contains an illustration of the “standard model” and components present in each one of its tiers.

The “standard model” separates accelerator control systems in three tiers. The first tier consists of workstation machines that provide the user interface to operators. The workstations display the current status of the control system, current and historical data measurements, alarms, as well as any other domain specific information operators require. The second tier, referred to as “communication tier”, transports data between the first and third tier of the control system. The third, and last, tier is responsible for implementing the domain logic of the control system. Devices in this tier[2] are responsible for the acquisition and processing of distributed data, supervision of the control system, as well as sequential and closed-loop control. This tier is the only one that communicates directly with the accelerator hardware [22].

The standard model also recognizes the importance of ensuring the correlation of data, as accelerators are distributed systems with strong real time constraints. Three approaches for correlating sensor data from the accelerator are defined by the model:

- Packaging data along with a correlation identifier generated at the source of the data, such as a time stamp.
- Synchronize the data collection system, in such a way that all data collection is triggered simultaneously.
- Determine a single component that collects all data [22].

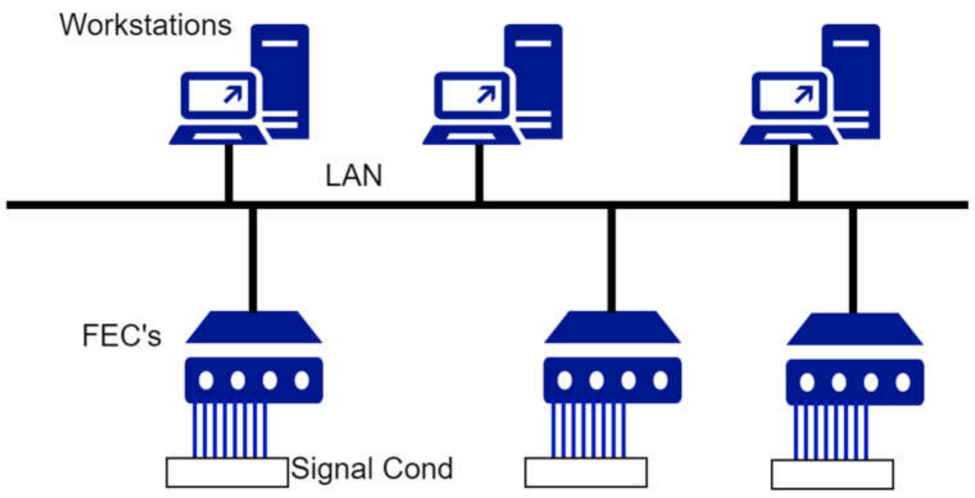


Figure 2.2: The three-tier standard model as proposed by Kuiper[21].

[2] In Kuiper’s article, originally published in 1991, the third layer devices are referred as Front End (Micro)Computers, or FECs. Meanwhile, computational devices have changed considerably and the term “front-end” is more commonly used to refer to devices that interface with the end user. In order to avoid confusion, we use the broader term devices when describing the standard model’s third layer computers.

Several variations of the standard model have been proposed to address challenges encountered in accelerator control systems. These solutions usually address specific requirements, and extend the standard model.

For example, Thuot et al. [22] propose an extension to the standard model to address three important requirements that were analyzed when designing the Superconductive Super Collider (SSC). These requirements were as it follows: firstly, dealing with the large spatial footprint of the accelerator and large number of devices interconnected, as the synchrotron was designed to have a diameter of 87 km. Secondly, the need for fast accessibility to global data. Finally, distributing the control system data to multiple users, without interfering with the control network [22].

To address these requirements, Thuot et al. [22] propose an extension to the standard model, adding multiple network tiers, using different infrastructure to address challenges in each tier. For subsystems with stronger real time constraints, a reflected memory scheme is added to several controllers to allow these controllers to have necessary data for operation in a short time window [22].

2.3 Control system hardware

While this document focuses on control system software, it is imperative for software engineers to understand the physical components the control system operates on. IT components designed to support control systems often have additional requirements, especially in particular domains as the accelerator, or the healthcare domain. In this section we go into detail about common hardware requirements of control systems, and the way they impact control system software.

Important requirements for an accelerator control system are dependability, fault tolerance, predictability, extensibility, and usability [17]. In the hadrontherapy domain, additional importance should be given to fault tolerance, dependability, and safety, as system failures pose unacceptable risk to patients and medical personnel. On facilities developed by commercial hadrontherapy system providers, that provide a catalogue of turn-key hadrontherapy solutions, the importance of extensibility is often lowered. In the medical domain, the usability requirements should be reinforced as poorly designed interfaces may lead to incorrect usage by trained operators, leading to errors which may cause permanent and irreparable harm [23].

2.3.1 Upper tier

In the accelerator control system domain, upper tier of control system presents the least amount of non-standard hardware requirements. Muller [17] states that control equipment at this tier only differs from regular IT equipment by requiring assurances regarding availability. Modern control system software at this tier often is able to run on all major operating systems.

2.3.2 Networking

In the network tier, control system equipment often differs from conventional network equipment. These differences have been highlighted by Galloway et al. [20] in a comparison between conventional and industrial control networks. The most visible difference is often the network hierarchy, with control networks often having much an architecture involving additional layers. Moreover, these different networks layers with different protocols allow communication between instruments and controllers, and then to operator interface equipment [20]. One example of layered network presented by Thuot et al. for the SSC control system [22], was previously discussed in section 2.2.2.

The network traffic patterns in control systems are also inevitably dissimilar from conventional networks. Control system network traffic must account for periodic data in the form of sensor measurement data, alarms, and state messages. Besides, real-time constraints in the order of microseconds must be accounted for when designing the network. Meanwhile, in conventional networks, network traffic usually consists of larger, aperiodic messages, and a “best-effort” policy is often used [20].

In order to fulfil real time requirements, custom network and I/O solutions are often used. One such example is presented by Štefanič et al. [24] for the Timing System of the MedAustron hadrontherapy center. This subsystem is responsible for broadcasting coordination messages to several accelerator controllers. Requirements for this subsystem included the creation of a deterministic real-time network to support a 1 microsecond control loop resolution [24].

2.3.3 Lower tiers

The lower tiers of the control system are often the most varied. In systems following the standard model, the lower tier should contain ‘Front-End Computers’ or microprocessors. This tier is the closest to the accelerator, and so usually possesses the strictest real time constraints. Devices in this tier are also responsible for obtaining sensor data, and if possible perform processing over them before sending the data to the upper tiers. Additionally, in many accelerator facilities the data correlation process occurs in this tier, by appending sensor measurements with a time stamp or cycle number for synchronization.

2.4 Control system software frameworks

As previously mentioned, particle accelerator control systems often follow patterns that can be leveraged to reuse concepts and architecture designs. Unsurprisingly, the same also occurs for control system software. These similarities were first pointed out during the first few international conferences [17]. Soon afterwards, several accelerator teams proposed control system software designs, while seeking collaboration partners for the development of reusable toolkits for the construction of accelerator control systems [17]. From these joint efforts, several successful open source frameworks for developing control systems were developed, such as EPICS [25] and Tango [26]. Additionally, several commercial options have been made available for usage in accelerator facilities. In this section, we describe some of the most widely adopted software frameworks for control systems in the particle accelerator domain.

We will use the term software framework as defined by Buschman et al. [27], who states that: "a software framework is a partially complete software system that is intended to be instantiated. It defines the architecture for a family of systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made" [27]. In our opinion, the software toolkits presented in this section can be classified as software frameworks, as they provide the building blocks for the creation of the control system, as well as its subsystems.

2.4.1 Experimental Physics and Industrial Control System (EPICS)

The EPICS architecture is the oldest control system framework designed for the particle accelerator domain. Dalesio et al. [25] note that at the time it was designed, no industrial solutions had offered the control system capabilities required by the experimental accelerator community [25].

EPICS is a distributed control system framework that runs on top of an infrastructure composed of workstations for operators, I/O controller computers, and file servers, connected via a LAN connection [28]. Later additions to the framework include a distributed database, as well as several subsystems necessary for accelerator control, such as alarm management, archiving, sequencing, and operator display [25].

The EPICS software framework was designed based on the following requirements:

- Providing standard functionality required in an accelerator control system.
- Reducing time required for control system development and alterations.
- Allowing for extensibility of the framework, following the requirements of experimental accelerator physicists.[22]

The EPICS project started in 1989 with the collaboration between the Los Alamos National Laboratory (LANL) and the Argonne National Laboratory (ANL). Los Alamos had just developed a control system, named GTACS, for their proton linear accelerator. Meanwhile, the ANL team had been tasked to build a control system for a new accelerator, the Advanced Photon Source [29]. The teams were aware that diverging requirements between the two very different accelerators designs might create issues throughout the collaboration, and force the development teams to diverge, but decided to go ahead with the co-development project [29].

During the early stages of the collaboration, the ANL and LANL teams worked together to improve the GTACS, improving utility, portability, and enabling additional device support. Finally, the co-designed control system was renamed EPICS [29]. The results of this collaboration was first presented in the ICALEPCS 1991 conference by Mcdowell et al. [30], and the description of the EPICS architecture by Dalesio et al. [25].

From that point onwards, the popularity of the project increased dramatically, and the resulting portable toolkit was then used by several other accelerator projects [29]. Notably, the Superconducting Super Collider Laboratory (SSCL), which was commissioning the largest experimental accelerator in the world, chose to use the EPICS framework [17].

2.4.2 TANGO Framework

TANGO is an open source framework for development of control systems with the DSA or SCADA architectures. TANGO is an object-oriented framework, and was developed on top of the CORBA [31] communication standard, using it to provide the network layer [32]. The CORBA standard defines a notation, used by software applications to communicate over a local or remote network. As a result, in control systems implemented using TANGO, control actions are performed by invoking CORBA methods or accessing CORBA attributes [26].

The TANGO framework is multi-platform and supports the Java, C++, and Python programming languages [33]. A few years after the start of the project, the TANGO framework had already implemented several services for developing control systems, such as, but not limited to [34]:

- A system wide relational database for persistent storage.
- Naming system for device addressing and discovery.
- Several standardized communication methods, with implementations for events, logging, data archiving, polling, executing device commands, and obtaining device attributes [34].

As part of the framework's original design philosophy, the TANGO framework is designed to omit the underlying CORBA implementation details from control system developers by wrapping communication under an API [26]. This philosophy has proven to be beneficial since, as of 2015, the TANGO roadmap includes the complete removal of CORBA middleware communication in favor of the ZeroMQ protocol [33]. By hiding the communication details under an object-oriented API, it is easier to exchange the underlying implementation.

The TANGO framework has been successfully used in the development of control systems in several domains. Currently, several accelerator systems use TANGO, such as ESRF, SOLEIL, ALBA [32]. Additionally, projects in other domains also use the TANGO framework, such as experimental lasers (ELI beamlines), and telescope control systems (SKA radio telescope) [33].

2.4.3 Commercial software toolkits

Several industries also depend on the usage of control systems, such as the automatic manufacturing industry, oil and gas distribution, and power generation. Therefore, several institutions in these industries rely on commercial providers of supervision and control systems. The accelerator domain is not different, and there are several providers of proprietary tools for building control systems [17]. Some large providers of control system solutions in the accelerator domain are National Instruments, developers of the LabVIEW language [35] and PXI instruments [36], Siemens, which commercializes the SIMATIC WinCC framework [37], and Vista Controls [38], which sells the Vsystem control system solution. As described in detail later, CNAO particle therapy facility uses several commercial components as parts of its control system, namely from Siemens and National Instruments.

Facilities using the Vsystem solution include the ISIS Neutron and Muon Source research center. This research center decided that its previous control system had to be replaced. In order to do so, the ISIS development team chose the Vista Controls as a software provider, and eventually migrated to Vsystem [39]. As of the current date, the commercial product is still the basis of their accelerator control system [40].

The SCADA system sold by SIEMENS, WinCC, which was previously named PVSS, is extensively used in the industry. The most notable accelerator control systems using WinCC are several of the CERN projects, which use it in four of its large experiment detectors. The decision to adopt the commercial control system framework was made following a survey project in 1997. The project aimed at investigating available commercial solutions, and evaluating their capability to fulfill the LHC requirements of their users. The project concluded that the adoption of a commercial SCADA system for the control of the LHC detectors would be technically suitable and beneficial [41]. Although initially adopted only for development of the LHC experiment control systems, the usage of the framework soon became more prominent, being adopted in several other projects [42]. As noted by Muller [17], WinCC has since then became the standard for SCADA related tasks at CERN [17].

Finally, an important commercial provider of hardware and software is National Instruments. In the hardware field, the company provides a large catalogue of components for PXIs (acronym for PCI eXtensions for Instrumentation), such as crates, modules, and controllers. In a crate, a controller runs a standard or real time operating system, manages and communicates with the crate modules. Multiple modules may also be installed, thereby being able to fulfil a wide variety of specialized tasks. Specifically, FPGA modules allow the execution of control software with real-time capabilities and execution speed not available in off-the-shelf computers. In modern accelerator control systems, FPGA modules and integrated boards often fulfil roles previously assigned to FECs in the standard model, discussed in section 2.2.2.

Additionally, National Instruments is also the developer of the LabVIEW programing language. The LabVIEW programming language supports a wide variety of drivers for diagnostic equipment, which is attractive for engineers in the accelerator domain [17]. Additional advantages of the LabVIEW language in this domain are its support for compilation for conventional platforms (Windows, Linux), as well as real-time platforms (FPGAs, LabVIEW real time). This allows the language to be used in several layers of a control system. Lastly, National Instruments have commercialized their DCS module for development of SCADA systems [43].

Several particle therapy facilities utilize National Instrument products in parts of their control system [17], including MedAustron [44], and CNAO. More recently, the Flerov Laboratory of Nuclear Reactions are developing the control system for their new isochronous cyclotron based on the DSC module [45].

2.5 Centro Nazionale di Adroterapia Oncologica

The role of a control system software engineer in a hadrontherapy facility is the development and/or maintenance of software applications for the correct operation of the facility. Therefore, good understanding of the underlying requirements of such a facility is essential. In the previous section, concepts relevant to hadrontherapy were introduced. While the last section focused on the general concepts, and presented solutions used in several different research and treatment facilities, this section focuses on the facility where this industrial research work has been performed, the Centro Nazionale di Adroterapia Oncologica.

This section is structured as follows. A brief timeline of the CNAO facility design and construction is presented in section 2.5.1. Then, relevant terminology to the control system is defined in section 2.5.2. Section 2.5.3 describes the anatomy of accelerator cycles. The organization of CNAO's control system, as well as description of each subsystems controlled by it can be found in section 2.5.4 and 2.5.5 respectively. Finally, the architecture of the present control system is described in section 2.5.6.

2.5.1 Brief introduction to CNAO

In order to concisely explain the history of the CNAO, it is easier to separate it into four time periods:

- Precursor proposals phase, 1991-2001, consisting of publication of the initial ideas for a multi-particle hadrontherapy center. Initial funding proposals [46].
- Design and construction phase, 2001-2010, composed of creation of Foundation CNAO, final design of the facility, and construction of the facility.
- Clinical trials phase, 2010-2013, consisting of patient clinical trials, and the certification of the facility.
- Operation and continuous improvement phase, after the clinical treatment of patients has started, includes the continuous improvements of the accelerator and sub-systems. Research and Development of accelerator design, research in bioengineering, medical physics and clinical areas.

Amaldi states that the EULIMA project, launched in 1987 and financed by the European Commission, motivated several national hadrontherapy projects, such as the one in Italy [1].

The precursor phase started with the publication of a report entitled "Per un centro di teleterapia con adroni" [47]. The report advocated for the particle accelerator physics expertise held by institutions such as CERN and INFN to be applied into clinical use for cancer therapy. The report, authored by two prominent group directors, located in CERN and the Niguarda General Hospital, attracted the interest of the INFN president at the time. Consequently, in 1991, funds were allocated by the INFN for studying accelerator designs for clinical activity using ion and proton beams [46].

The TERA foundation was founded in 1992, and was the main contributor to what would become the CNAO Foundation project. Headquartered in Novara, Italy, the TERA foundation would come to staff over 170 personnel in the hadrontherapy domain, such as physicists, engineers, and technicians [46]. In the next 10 years, the TERA foundation authored three hadrontherapy facility design proposals. The first two proposed facilities aimed for the city of Novara, Italy [48], while the later aimed at Milan, Italy [49]. While the initial proposals were eventually unsuccessful in securing funding for building the proposed hadrontherapy facility, a study by CERN (with collaboration of TERA, MedAustron, Oncology 2000, and GSI [1]), was performed in the year 2000 [50][51], which provided the TERA the groundwork for the proposal which would later become the CNAO foundation [46].

Regarding technological decisions, the EULIMA project recommended the use of a synchrotron [52]. At that point, oxygen was proposed as the heavy ion of choice, but later scientific consensus indicated that carbon ions would instead be a better choice [1].

The design and construction phase, started in 2001, saw the concrete designs for the facility finalized [53]. The CNAO foundation received, as a result of a signed agreement between the two organizations, extensive documentation from the TERA foundation as well as intellectual property. Other facilities that contributed to the CNAO specification were CERN, GSI, and INFN [1]. The plot of land where the CNAO facility was built, next to the San Matteo general hospital of Pavia, was granted free of charge by the Province of Pavia [1].

During this phase, in 2003, the design of the control system began. In order to do this, the accelerator specification was analyzed, resulting in a set of requirements for the control system. From these requirements, along with documentation detailing a set of best practices and standards, the control system, and its sub-systems were designed and developed.

Finally, in 2013, the CE certification label was obtained for the facility. In addition, the government health authority approved the treatment of particle therapy within the Italian national healthcare system [53]. From the end of the clinical trials in 2013, up to 2015, 256 patients were treated with proton and carbon ions [53].

2.5.2 CNAO control system terminology

CNAO is a medical facility with the purpose of treatment of cancer using particle beams. The control system of CNAO has the goal of supervising and controlling the facility's medical particle accelerator. In this section, we define hadrontherapy and control system terminology that is used throughout this document. The definitions presented here are in accordance conceptual model presented in internal documentation of the CNAO facility [54]:

Treatment - A treatment is defined as the process by which an amount of energy, or dose, is delivered to a patient via particle beams. During a treatment, energy is transferred to a tumor in slices. A treatment is composed by several cycles.

Slice – A slice is a partition of a tumor. Depending on a tumor size, it may have one or several slices. A single slice can be treated with one or more beams cycles.

Beam – A particle beam is defined by its characteristics, such as type of particle, and kinetic energy.

Beam Cycle – A Cycle corresponds to the amount of time it takes for the accelerator to produce a beam. The term cycle is also used to refer to the process by the accelerator to produce a beam. After every cycle, the accelerator returns to the initial state.

Cycle code - In the CNAO facility, beam cycles are defined by a number given to them. Two beams with the same cycle code are accelerated in the manner and should possess the same characteristics. The cycle code contains information such as the cycle energy, the particle, and the accelerator line, which defines the treatment delivery room.

Cycle number - In the CNAO facility, the cycle number is a sequential number that uniquely identifies every accelerator cycle performed.

Running condition – An accelerator running condition is the set of all accelerator equipment settings for producing a beam. For each possible beam, defined by its cycle code, there exists a running condition that will produce this beam. There is, therefore, a mapping between every cycle code, and a set of equipment settings for producing this beam. An accelerator running condition has the lifetime of a cycle.

Cycle event – An accelerator cycle is composed of cycle events. These events represent a specific stage of the acceleration process. The anatomy of an accelerator cycle is discussed further in section 2.5.3.

Operator – An operator is a personnel of CNAO tasked with supervising the accelerator and its behavior. Operators have the responsibility of accepting treatment requests at specific treatment rooms. Additional responsibilities of operators include monitoring the status of the accelerator, which is presented in control system consoles, and acknowledging all control system alarms by taking the appropriate actions.

2.5.3 Anatomy of a cycle

An acceleration cycle is the process of, from a known starting point, injecting particles into the synchrotron accelerator, accelerating the particle beam until extraction, and finally retuning back to the starting point. This process, which occurs for every particle beam delivered, is the central focus of the control system for various reasons. Firstly, the accelerator's subsystems were designed in a way that the vast majority of devices must be provided with their respective configuration settings every cycle before the cycle begins. Likewise, the majority of sensor data is obtained from the accelerator's lower layers only once in the span of a cycle [54]. Tasks that have to be performed in timespan of a fraction of a cycle are relegated to the lowest layers of the control system. Additionally, the synchronization of these tasks is often guaranteed by cycle events generated by the timing system [55].

An acceleration cycle starts with a specific timing system event. Timing system events contain, among other information, the cycle number and the cycle code. The cycle code, as discussed previously, defines, among other characteristics, the beam particle and energy. Each particle beam containing the same cycle code is by convention referred to as of the same type. Every energy level that can be produced by the accelerator is mapped to a cycle code [54]. However, not all cycle codes define a particle beam type. This is because of the existence of empty cycles, which are executed to maintain the accelerator in a stand-by mode. The mapping process is an offline configuration process, and not all energies levels that the synchrotron accelerator is able to produce are mapped at all times. On the other hand, a cycle number uniquely identifies every single cycle accelerated since the beginning of the accelerator's operation. Cycle numbers can therefore be used as a timestamping measurement.

Once the injection process is finished, the start acceleration event synchronizes the acceleration process, accelerating the particle beam to the desired extraction energy. Afterwards, when the energy is met and the extraction event is processed, the particle beam is slowly extracted, leaving the synchrotron. After the extraction, the hysteresis process begins, where the magnetic field in the synchrotron's magnets are set to the maximum value, and brought back to the minimum afterwards. This process has to be performed at the end of every cycle.

At the CNAO facility, acceleration cycles may have varying lengths depending on several factors. The extraction duration, which is variable, has the largest influence on the cycle length variation. At the present moment, cycles range from just over a second, to at most 3 and a half seconds. Figure 2.3 depicts the lifetime of a sample cycle, displaying in the vertical axis the magnetic field over the cycle duration. This chart also displays several timing system events that are fired during a cycle.

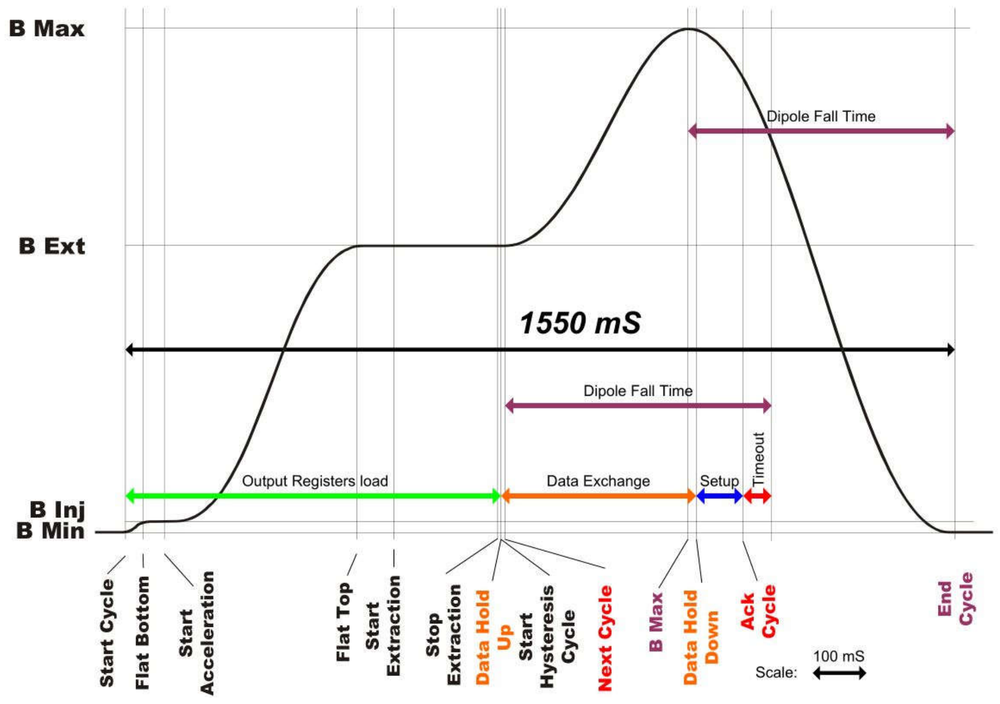


Figure 2.3: Standard cycle chart displaying timing system events. Extracted from [55].

2.5.4 Control system conceptual model

The CNAO accelerator is a very complex system that contains several sub-systems, and communicates with a wide variety of parallel systems.

In order to represent the boundaries of the control system, Figure 2.4 presents an allocation view of the control system [54]. An allocation view presents a mapping between software elements and physical resources [56]. The objective of this view is to separate various conceptual roles and responsibilities in the control system. By displaying the conceptual layers and their allocation to hardware components, we are able to illustrate the complexity of the system, and various different engineering domains required between the various control systems developers.

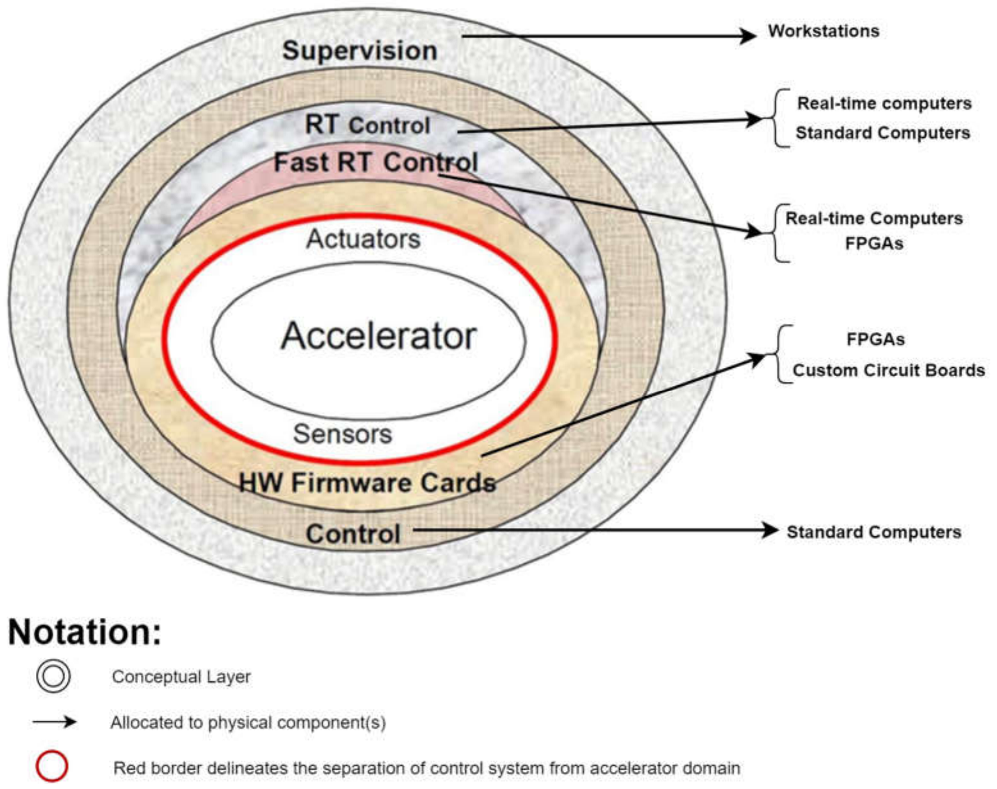


Figure 2.4: Allocation view of CNAO control system. Adapted from conceptual model of the CNAO control system [54].

Each layer displayed in Figure 2.4 is characterized by its set of responsibilities in the CNAO control system. The outermost layer is the supervisory layer. This is the only layer that interfaces with operators. The supervisory layer also provides interface for other non-operator users during offline operations. The physical resources that support this layer are the workstation computers present in the facility's control room.

The control layer performs control of device properties and acquisition of data. This layer deals with the accelerator subsystems as abstractions, regardless of their physical aspects, this layer runs on top of standard, off-the-shelf hardware.

The real time layers perform the majority of control operations and run in a cycle timescale, obtaining data from the accelerator components and setting any values required. Because the regular real time layer was designed to perform operations in the timespan of a cycle, which lasts several seconds, a small part of the real-time does not require specialized real-time hardware. Nevertheless, data acquisition and control operations that require fast real-time capabilities are provided with physical resources according to their timing requirements. Consequently, real-time computers, such as National Instruments real time crates, FPGAs, and even custom circuit boards support this layer's subsystems.

During the original design, the HW Firmware cards layer was mostly composed of specific electronic equipment. These electronic boards would be tasked with processing signals coming from accelerator physical systems into digital values that could be interpreted by the control system layers. However, over the years, with the availability of increasingly more powerful embedded systems, complex operations can be performed at the hardware layer. Therefore, the hardware layer is no longer designed exclusively by sensor and actuator engineers, opening room for local control processes to be performed by specialized circuit boards and FPGAs.

2.5.5 Subsystems

In the CNAO control system's WinCC software, "equipment components" are software entities that represent a recognizable entity type, which can be logical or physical. Examples of equipment components are magnets, power suppliers, or entire crates. Through the WinCC control system, equipment components can be interacted via their properties and operations. Properties are used for reading or writing variables defined by the equipment components. Meanwhile, operations perform an activity in a specific equipment component. Operations and properties may refer to an equipment component instance, or alternatively, an equipment component type, similarly to how static methods behave in object-oriented programming. Furthermore equipment components may be seen in the supervisory layer as a virtual instruments representation.

Figure 2.5 displays the visual instrument representation of a vacuum pump equipment component instance. In the component, properties (such as "Status" in the figure) are used to obtain status information from the instance, additionally, several operations can be executed by operators of the control room, such as 'start pump' and 'stop pump'.

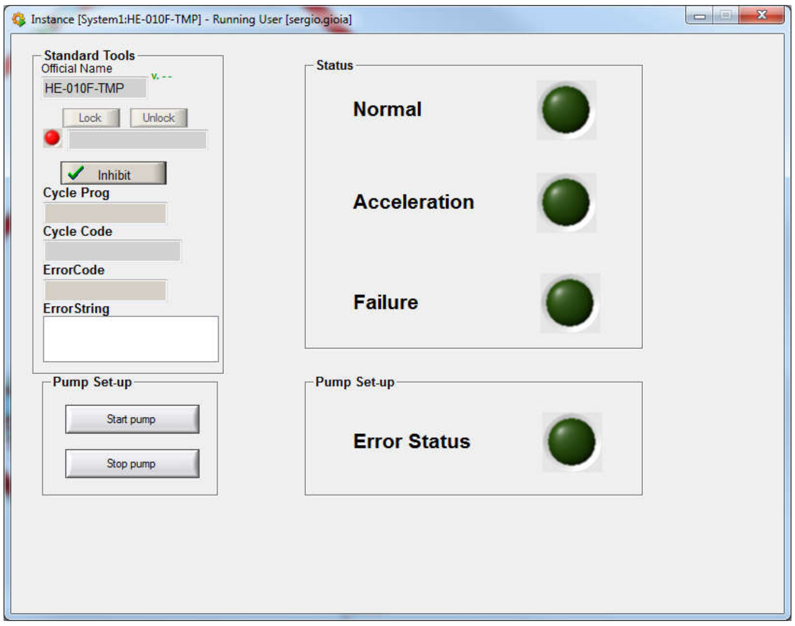


Figure 2.5 – Simple equipment component visual interface representing a vacuum pump instance. Extracted from [57].

The control system supervises several subsystems. These subsystems are not necessarily part of the control system, but are supervised, and sometimes controlled by it. Most subsystems are standalone components, designed by specialized personnel of other engineering departments. These components interface with the control system for supervisory purposes, such as delivering sensor data. In the control system architecture, subsystems are also represented as WinCC equipment components.

CNAO's particle accelerator contains a linear accelerator for the purpose of injecting of particles into the synchrotron. The linear accelerator possesses its own third party control software, developed by its manufacturer. Therefore, an equipment component was developed to act as the interface between the linear accelerator controls and the CNAO control system. From the point of view of the control system, the linear accelerator is a subsystem. Figure 2.6 contains a screen capture displaying the virtual instrument representing the linear accelerator equipment component. As shown in the figure, the linear accelerator equipment component exposes a several properties, which are monitored and archived by the control system.

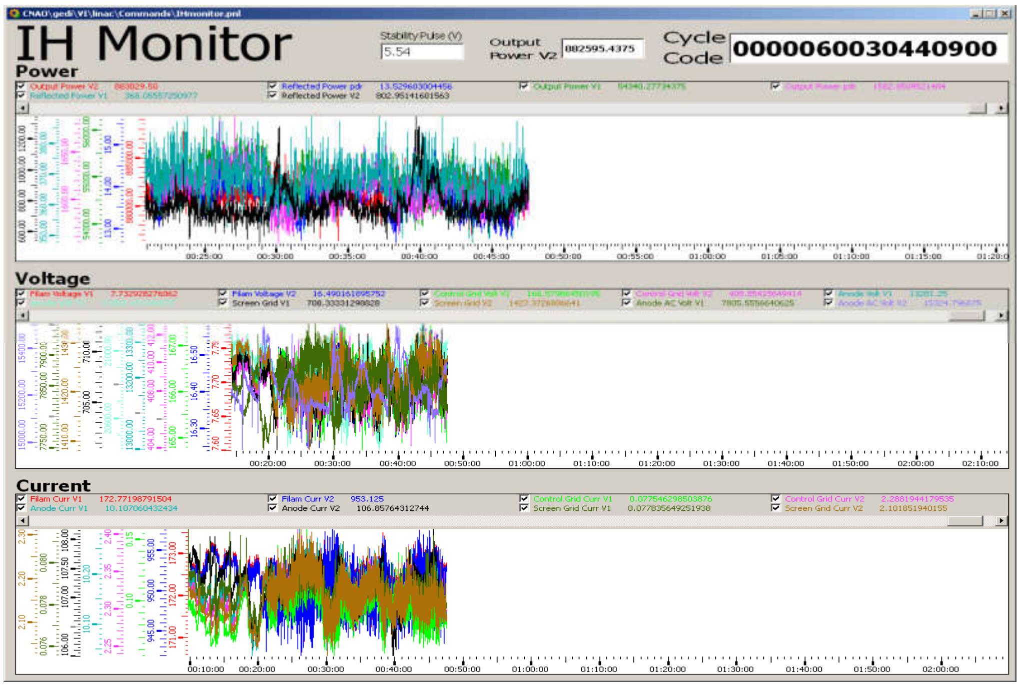


Figure 2.6 - Equipment component visual interface representing the linear accelerator subsystem. Extracted from [57].

The control system at CNAO supervises the following subsystems, according to the control system organization documentation [54]:

- **Radio Frequency Cavities** – Subsystem responsible for controlling RF cavities, which accelerate the beams.
- **Beam Instrumentation** – Subsystem responsible for all beam monitoring functions. Assures beam quality and provides measurements such as beam stability, beam loss, and beam quality.
- **Conventional Magnets** – Subsystem responsible for control and supervision of the accelerator's conventional magnets.
- **Custom Magnets** - Subsystem responsible for control and supervision of the accelerator's special magnets.
- **Vacuum System** - Responsible for maintaining the correct vacuum level inside the vacuum chambers, where the beam travels.
- **Particle sources** – Subsystem responsible for control and supervision of the sources that generate the particles to be injected.
- **Linear Accelerator** – After generation and before injection into synchrotron ring, the beam passes through a linear accelerator in the center of the synchrotron. This linear accelerator was purchased as a standalone component, and it mostly manages itself. Therefore, it interfaces with the control system as a subsystem.
- **Beam Delivery System** – Complex subsystem that controls the steering magnets, positioning the beam for scanning each slice. It is

also responsible for calculating the dose delivered to each part of the tumor.

- **Conventional Plant Manager** – Manages the treatment plant infrastructure, such as water, electricity, temperature, and personnel access control.
- **Patient Positioning System** – The patient positioning system calculates the patient's position in the treatment room, ensuring that the patient is properly positioned in the treatment equipment. This system is necessary for the accuracy of the treatment [54].

Due to technical and integration issues, not all subsystems follow the conceptual model presented earlier. Some subsystems, located in the lower layers, are not integrated with the control system. This means that they are not defined as equipment components, cannot be interacted with directly through operations, or even indirectly with processes and procedures. Examples of non-integrated subsystems are the radio frequency cavities, the sources manager, and the beam delivery system.

Because these non-integrated subsystems still require supervision, they communicate via TCP socket directly to the supervisory layer, and expose their internal state and operations in a non-standard way. Due to not being integrated, they require custom made supervisory applications, custom data pooling, and archiving procedures. This is a current control system integration inadequacy, and adds limitations to accelerator control procedures. Figure 2.7 presents the virtual instrument of a non-integrated subsystem, as displayed to operators in the control room. This virtual instrument supervises and controls the sources manager subsystem.

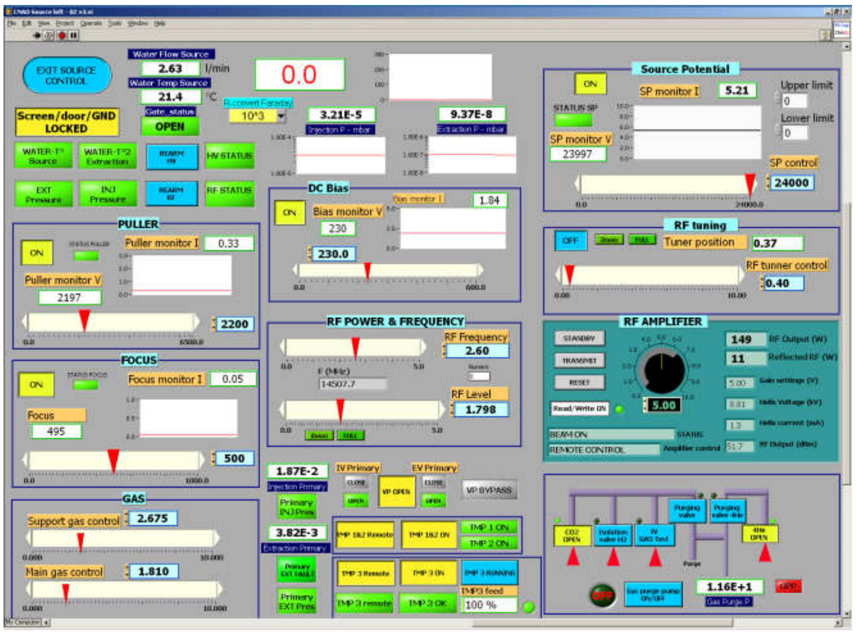


Figure 2.7: Virtual instrument representing the Particle Source subsystem. Extracted from [57].

2.5.6 Control system architecture

The current physical architecture of the control system is presented in Figure 2.8 [54]. This figure shows the physical levels of the control system, specifying the hardware equipment and technology used in each level. Additionally, the diagram presents the data transfer periodicity of the communications between components.

The architecture was described internally in two internal reports by Casalegno et al. [54][58]. Since then, more than 15 years later, the iterative improvement of the control system has resulted in some architectural drift. Architectural drift, is described by Bass et al. [56] as the mismatch between the latest documented architecture and the physical system, designed from such architecture. In this thesis, we bridge the architectural drift presenting the current architecture.

This architecture focuses on the upper levels of the control system because the scope of this project is the technological update of the Configuration and Support environment, which is located in the first and second level. This thesis describes the upper levels in detail to allow readers to understand the Configuration and Support environment and its interactions with the rest of the control system. Consequently, extensively detailing the lower levels would be outside of the project’s scope.

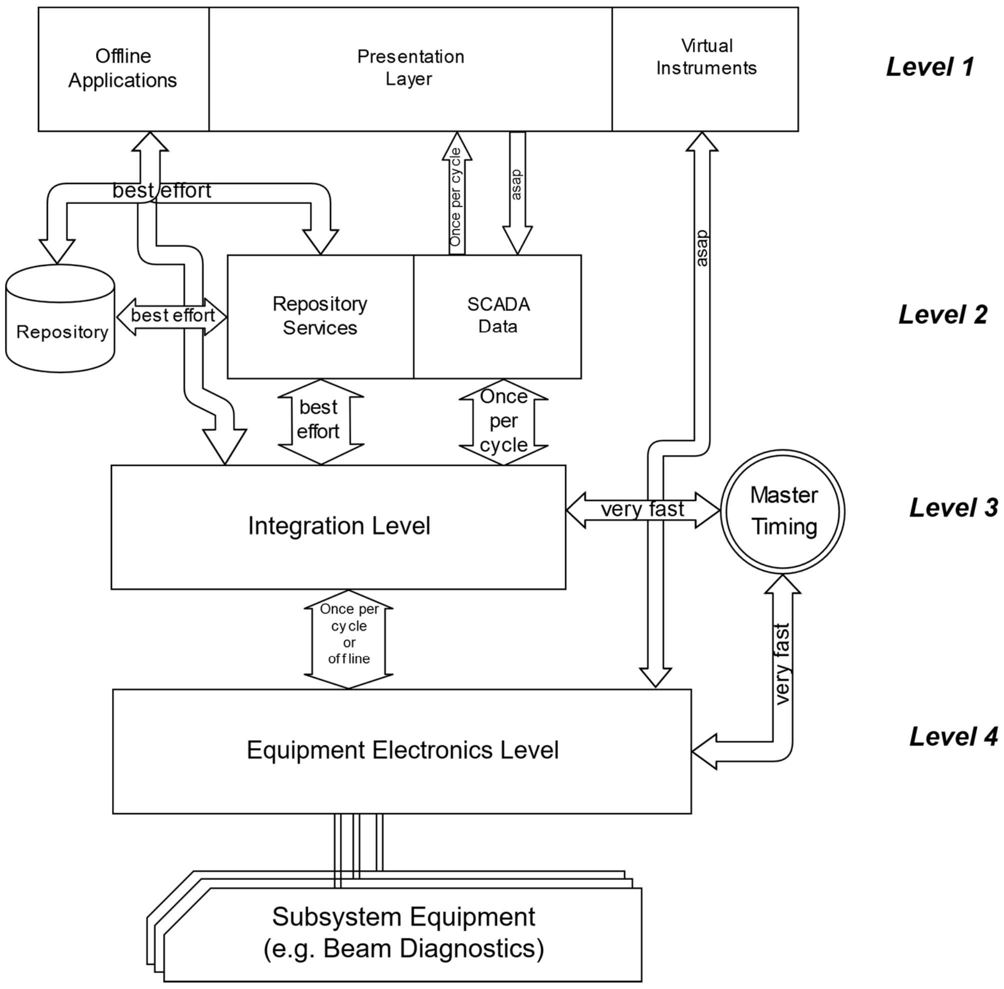


Figure 2.8: Control system architecture, adapted from [54] and altered to take architecture drift into account.

2.5.6.1 Level 1 - Presentation and operation

The first level is entitled '*Presentation and Operation Layer*', and contains several types of human machine interface enabled applications. The largest component present in the level is the collection of SCADA system client applications. These applications display all information relevant to operators gathered in the WinCC SCADA system, and allow operators to call operations and procedures. The information presented in the client applications is retrieved from the second level WinCC SCADA system, and updated at every cycle during accelerator operation. Outside of accelerator operation, these applications allow the execution of accelerator maintenance and quality assurance procedures.

Virtual Instruments, written in LabVIEW, provide human machine interface to operators for systems that are currently not fully integrated into the control system supervision system architecture presented here. These virtual instruments generally communicate directly with the fourth level via TCP sockets or proprietary protocols. No operations are required to be performed to these instruments during operation of the accelerator.

Repository management and support front-end applications are also part of this level, represented in the diagram as offline applications. This set of applications, written in C#, possesses two main purposes. To begin with, they are responsible for configuring and managing information located in the repository, which is part of the second level. The second purpose is to serve as support applications, without real time constraints. These are generally applications that are not well suited for the WinCC SCADA development environment due to their complexity and lack of real time constraints. Support applications in this level include the patient scheduler, which keeps track of patient schedule and treatment progress [59], monitor the treatment delivery workflow [60], or view historical data and print necessary daily reports [61].

2.5.6.2 Level 2 - Concentrator and data management

The second level is entitled ‘Concentrator and Data Management Level’. The control system conceptual model presented in section 2.5.4 is implemented as WinCC SCADA components. Additionally, the SCADA system in this level is responsible for obtaining data and alarms from the third level and delivering them to the upper level SCADA terminals. Operational data from the accelerator is also archived at this level.

This level also contains the repository and the repository services. The repository is a database cluster that contains all information necessary for setting up and running the accelerator. This information includes information on the physical characteristics of the accelerator, configuration settings for software applications, accelerator settings for all planned treatments, and information to link patients to their planned treatments.

Information in the repository is only transferred to the lower levels by the repository services when explicitly directed to do so by maintenance operators, and never in the middle of the operation of the accelerator. Only applications that do not directly interact with the accelerator are able to interface with the repository on a regular basis.

The repository service applications are responsible for writing all set up information to the third level when required. Additionally, repository applications in this level provide services to their counterparts in the first level. The repository services, along with the repository and support front-end applications, form the Configuration and Supervision Environment.

2.5.6.3 Level 3 – Integration Layer

The integration level is the smallest level in the control system. This level is responsible for the real time data transfer from the fourth level, making it available to the upper levels. This process is synchronized with the ‘Data Hold Up/Down’ event displayed in Figure 2.3. This allows the third level to perform synchronization of data gathered each cycle.

Between the third and fourth level, communication can be performed via a multitude of protocols. While the standard communication protocol in the third level is OPC-UA [62] (updated from the older OPC-DA protocol which was used previously), not all fourth level equipment is able to communicate using OPC-UA. Therefore, the third level was designed with wrapping services that communicate in the other proprietary protocols used in the fourth level. All communication performed from the second to the third level is performed using the OPC-UA protocol over Ethernet. Consequently, the equipment server level conceptually behaves as an aggregation and translation device, allowing data originated from several subsystems to be accessible in a standard manner.

OPC-UA is platform-independent standard that defines operations which must be supported by equipment to allow communication and interfacing. By following the standard, devices are able to communicate, expose data, internal state, and methods to other devices, securely and efficiently [62]. The standard is extensive and defines many possible configurations in order to better suit the operational environment, such as data encoding, security and transport protocol. Additionally, while the standard defines a wide amount of potential operations between OPC-UA enabled devices, these devices only have to implement a subset of them, as defined by their profiles. OPC-UA profiles define an array of testable functionalities, for example, if an application supports all functionalities defined in the profile, then the application is defined to support the profile [62]. In CNAO, applications in the third level support data access, and historical access. Other profiles such as method invocation are not supported.

2.5.6.4 Level 4 - Equipment electronics

The equipment electronics level is the most diverse, as components in this level are mostly developed by the various groups designing the accelerator systems. This level is the closest to the accelerator equipment, and possesses the strictest real-time requirements. Components in this level are composed of electronic cards with microprocessors or crates with real time processors and electronic cards.

Data required for operation in this level should already be contained in the electronic cards before treatment begins. This happens because, during treatment operation, the fourth level receives information from the timing system, in the form of cycle code, and events. Outside of treatment operation, information may be loaded to the fourth level from the equipment server level.

2.5.6.5 Timing and Signal Distribution Services

Following the defined architecture of the control system, the timing system is not part of any level. The timing signal distribution services is responsible for generating and distributing the signals necessary for coordinating the activity of the accelerator facility. The main timing system generates and distributes the timing events that define the accelerator cycles, as explained in section 2.5.3.

Subsystems that subscribe to timing system events, and require hard real-time behavior, are equipped with custom-made electronic circuit boards capable of receiving and decoding timing events in real-time. Fiber optic field busses connect the main timing system and the receiving subsystems. A conceptual model of the main timing generator is shown in Figure 2.9.

Additionally, some of the equipment in the accelerator requires specific information such as the magnetic field present in the machine, and the radio frequency, which represents the revolution speed of the particles inside the accelerator. In order to provide these signals, the timing and signal distribution service is responsible for the generation and delivery of these signals to the required subsystem components [55].

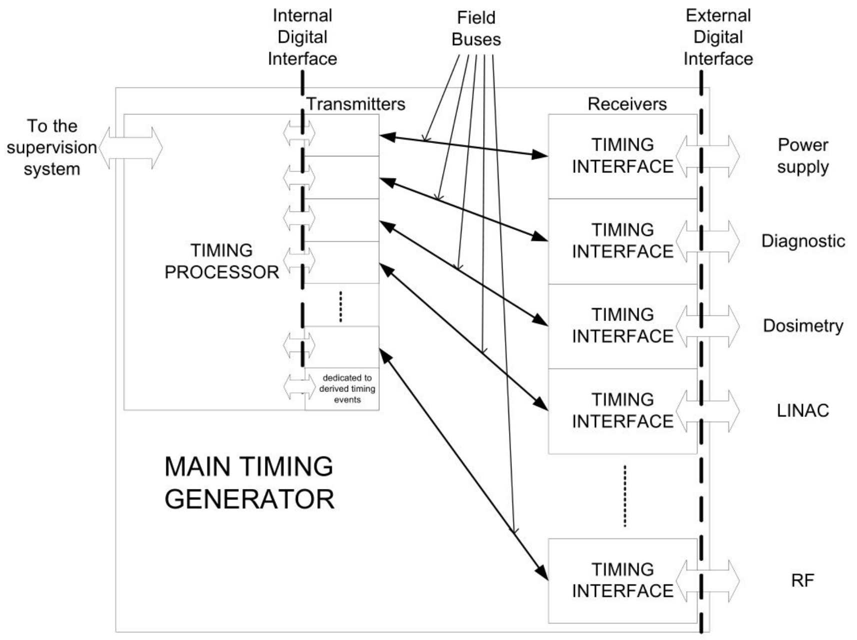


Figure 2.9: Diagram representing the main timing system conceptual model. Extracted from [55].

2.6 Product line architecture

As defined by the Software Engineering Institute, a product line is "a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission, and that are developed from a common set of core assets" [63]. In this definition, core assets refer not only to software components, but also documentation, designs, and other development artifacts. In a product line approach, the architecture and core assets are designed with reusability in mind, by specifying a series of variation points, allowing the assets to be reused more easily in each product line family member.

A highlight of the product line software development strategy is that it also presents strategies for the organization and roles of the personnel involved in the software development [64]. According to McGregor [64], product line development teams can be separated into the roles of core asset developers and product developers. Under these roles, core asset developers develop resources to be used in final products, whereas product developers are responsible for developing a final software application, and use core asset resources when necessary, guided by the scope of the application [64]. It is also highlighted that core assets are not only software components, but also support material such as architectures, and test plans [64].

The product line approach, if successful, allows minimizing the effort spent in development for individual products [56]. This approach has been regularly used in fields other than software, such as the manufacturing field. In the software development field, companies such as Hewlett-Packard, Cummins Inc., Deutsche Bank, and Phillips have used this approach with very good results [56].

Before defining a product line architecture, the architect must identify a set of possible products to be developed, or alternatively, possible functionalities present in such products. This set becomes the scope of the product line, and defines what products should, and should not, be developed using the product line architecture. The architecture should provide enough non-varying aspects to be useful, but sufficient variation points and assets to incorporate all features or products defined in the scope [64].

A product line architecture differs from a regular software architecture by the description of varying aspects in addition to non-varying aspects. The varying aspects are described using variation points, and for each variation point, variation strategies, as well as possible implementations have to be described by the architecture [56].

A similar concept also present in the literature is the notion of a reference architecture. One of many definitions of reference architectures in the literature, provided by Angelov et al. [65], defines a reference architecture as “a generic architecture for a class of information systems that is used as a foundation for the design of concrete architectures from this class” [65].

Nakagawa1 et al. [66] compare the two concepts, concluding that product line architectures possess narrower scopes, as they focus on an application family, and also encompasses the description of the variabilities present in each software of the family. Meanwhile, a reference architecture attempts to stay at a higher abstraction level, and provide solutions and design knowledge to be used for the design of an often broader set of applications [66].

Chapter 3

Upgrade of the Configuration and Support Environment

The configuration and support environment is a set of applications present in the first and second layer of the control system that share requirements, as well as a development environment. This environment is composed of the repository management applications and support applications, present in the first layer, as well as the repository services, present in the second layer. Applications in this environment interact with the repository, the third layer's OPC-UA services, and the patient management system. These applications are used by CNAO personnel for the operation of the facility. Finally, configuration and support applications generally do not possess real-time requirements, and share the following responsibilities:

- Managing the control system repository, as described in section 2.5.6, allowing qualified operators to set the facility's configuration settings.
- Obtaining patient information from the facility's commercial patient management system during the start of operation, and relaying this data to the accelerator subsystems.
- Transferring all configuration data from the repository to the third layer when reconfiguration is requested by qualified operators.
- Dealing with facility management operations that require a complex behavior, such as:
  - Scheduling of patients and monitoring the treatment progress [59].
  - Management of historical data, and generation of documents necessary for compliance with government regulation [61].
  - Management of accelerator cycles in the repository, allowing the upload of accelerator cycles into the required subsystem [67].
  - Monitoring the treatment delivery workflow, displaying the current state to operators present at the control room [68].

At the start of this work, we estimated that there were around 50 deployed software applications belonging to this environment at the CNAO facility.

3.1.1 Objectives and constraints analysis

The objective for the configuration and support environment technological upgrade is the design and development of tools necessary for the development of a new generation of software applications.

In this section we present the objectives and constraints gathered for the upgrade of the configuration and support environment. We differentiate the two concepts, objectives and constraints, in the following way. Constraints are environment properties that are already present in previous version the configuration and support environment. These constraints are considered essential for the correct operation of the environment, and the control system. If constraints for the new environment are not followed, a substantial amount of potential applications of the control system will no longer be able to be supported under the new architecture. Therefore, if any constraint is not followed, the upgrade is considered a failure. Alternatively, objectives define goals, that once achieved provide extensions to the legacy configuration and support environment. By fulfilling objectives, new application designs become viable, improving the facility's control system. If a solution does not fulfil all analyzed objectives, the presented design may be considered limited in scope, but nevertheless a viable solution.

The main objective in regard to technological upgrade was permitting the integration of mobile applications into the control system. In 2003, when the control system was designed, and technologies were chosen, mobile phones were capable of executing the operations for monitoring the accelerator. However, at the start of the project, in 2016, mobile devices already offered a suitable platform for accelerator operator to use for monitoring purposes. By achieving this objective, control system operators are allowed more mobility. Because architectures that facilitate multiplatform development have become more prominent, namely service-oriented architecture [69], and multiplatform development frameworks have become more robust, the integration of mobile devices has been estimated to be an achievable objective.

Another objective was assuring that new environment applications depend on services implemented on a standard and open interface. These services should allow other environments of the control systems to interface with them if desired. This objective was originated from the analysis of the three applications types that populate the presentation layer of the control system: SCADA, LabVIEW, and C# applications. While the SCADA system and C# applications were able to access the third layer, only C# applications are able to access the repository. This objective dictates that the environment architecture should be designed considering service interoperability as a desired quality attribute.

Additionally, a proposed objective was improving the validation process, tackling issues identified in the legacy environment. This objective aims at reducing application flaws, and reducing the effort for medical software certification. Unlike the previous objectives, which contained a clear evaluation criterion, this objective does not contain an expected solution and evaluation method. As part of this objective, achieving a greater separation of concerns in software applications, namely between user interface elements and business logic elements, was proposed as a starting point.

Finally, the last objective was to phase out deprecated or stale development tools and technologies in order to extend commercial and community support for components used, as the previous environment's lifecycle has so far exceeded 10 years of operation. Support for many of these technologies is not assured for long. The technologies to be chosen for the new environment should be expected to have an acceptable level of support during the next decade.

Regarding the constraints, the main constraint was the necessary re-implementation of several standard operations performed in the legacy environment. This is because that, due to the large number of applications expected to be designed and maintained, not having a standardized implementation for these operations would result in longer development times. The operations considered to be standard operations are, among others: repository access, logging, configuration file management, and initialization.

Applications of the legacy environment had access to a vast selection of libraries implementing standard operations, shortening their development times. Without them, the developers would have to re-implement these operations for every application. In order to maintain this functionality, new libraries implementing equivalent operations have to be developed for the new environment. Alternatively, if applicable, old libraries can instead be ported from the legacy environment to the upgraded one.

Newly implemented applications of the configuration and support environment are expected to run side by side with legacy applications, and cannot impact their counterparts negatively. Therefore, another constraint is the compliance with the previously defined application's conventions, since legacy applications take their compliance for granted. Compliance with formally defined best practices [70], as well as undocumented internal conventions have to be followed. These internal conventions, such as database table naming conventions, cannot be changed without an acceptable justification, and a documented interoperability solution.

Additionally, a new access control system has to be implemented. The previous authentication and access control system used Microsoft's Active Directory [71], and thus it cannot be accessed by mobile devices. Any solution must contain an authentication and authorization procedures available to applications in all platforms.

Finally, the last constraint was the compatibility of the solution to automatic code generation techniques. In the legacy environment, application development based on automatic code generation has been performed, and concerns were raised if these could not be implemented in the upgraded environment. Consequently, at minimum, a proof-of-concept implementation of automatic code generation is required.

3.1.2 Product line requirements

In order to design the first iteration of the product line architecture, the objectives and constraints presented in section 3.1.1 were analyzed to extract architectural significant requirements. The results are presented in the following:

- **R1:** Application projects developed following the product line must be able to be executed in the Windows 10 platform and in the Android platform. Extending the application to a second platform should not consume more than 40% of the total development time.
    - Non-functional requirement: portability.
    - Validation: presentation of prototypes executing in multiple applications, demonstrating duplicated logic is limited.
- **R2:** Product line applications should depend on services with platform agnostic interfaces.
    - Non-functional requirement: portability.
    - Validation: documentation of services in product line architecture.
- **R3:** Applications domain layer should not depend on presentation layer classes.
    - Non-functional requirement: reusability, Testability.
    - Validation: separation documented in product line architecture.
- **R4:** No deprecated libraries should be used in the product line at the time of the upgrade.
    - Non-functional requirement: maintainability.
    - Validation: product line architecture documentation.
- **R5:** At the time of choosing third party libraries to be used in the environment, the latest stable version should be chosen. Exceptions to this requirement may apply if newer versions do not conform to other requirements.
    - Non-functional requirement: maintainability.
    - Validation: product line architecture documentation.
- **R6:** The following operations should have one and only one correct implementation, developed by the core asset team, and used as an application component: Configuration setting management, local logging, and page navigation.
    - Non-functional requirement: reusability.
    - Validation: product line architecture documentation.

- **R7:** Applications should be able to communicate with the following services:
    - Repository service, which provides access to the control system repository.
    - Remote file system service, which provides access to a common file system for configuration and support applications.
    - Devices in the control system third level, using OPC-UA.
    - Remote logging service.
- **R8:** Product line architecture applications should be able to provide credentials to authenticate the operator using the application. Once credentials are provided by the operator, credentials should be used for all secured services interfaced by these applications.
    - Functional requirement
    - Validation: Prototype following product line architecture and displaying communication capabilities.

3.2 Product line architecture scope

A central goal of the project is the design of a software architecture for the configuration and support environment applications. As this architecture has to be used in the development of multiple software applications that share similar requirements, a product line architecture was chosen.

In this project, the scope of the product line architecture has been defined as a subset of configuration and support environment software applications of the CNAO control system. This subset encompasses all applications with user interfaces, written in the C# language, developed using the Xamarin development framework [72]. Applications that fall outside of the product line scope are, among others, service applications, as they contain no interactive user interface.

3.3 Configuration and support environment product line architecture

3.3.1 Technology choices

After analyzing the objectives and constraints presented in section 3.1.1, and refining them into architecturally significant requirements presented in section 3.1.2, a product line architecture was designed. In this section, we present the methodology used in the design of the architecture, followed by the technological choices of the environment. Lastly, we present a class diagram of the product line architecture.

Regarding the choice of technologies, the product line architecture should specify the programming language and common runtime infrastructure to be used by applications. Several legacy libraries, using the C# programming language, define standard operations, which need to be supported. Therefore, using the same programming language facilitates the task of porting the programming logic to the new architecture. The common language infrastructure used in the legacy environment is the .Net Framework 4.5. However, the CLI is not expected to be the most supported in the future by Microsoft and the open source community. Therefore, we have decided to move to the .Net Core CLI. Porting libraries from .Net Framework 4.5 to .Net Core is not automatic, but much of the original code are able to be repurposed.

The .Net platform also contains the Xamarin development environment, which allows the compilation of C# code into native Android. Using Xamarin, the same C# codebase can be used in both platforms. However, some native features present in only one platform, such as GPS, and accelerometer support, still require development targeted to each platform. Still, developing configuration and support front-end applications using C# using the Xamarin development environment fulfils multi-platform execution requirements (R1).

Regarding separation of multi-platform, and platform specific code, Xamarin applications may be organized in three different ways. In the Shared Asset Project approach, a single software project is compiled to all platforms, while special notations indicate platform specific code [73]. This approach creates maintainability challenges, as single and multi-platform code can be found in alongside each other in the same files. The portable class libraries approach, and the .Net Standard base library approach, define shared library projects that can only contain multi-platform logic. These libraries are then used by multiple single-platform projects [73]. The code sharing model of these two approaches comply with R6, which dictates that standard operations should be designed as reusable libraries. During the start of the project, only the PCL approach was supported by Microsoft, but since then, the PCL approach has been deprecated [73]. Therefore, the .Net Standard base library approach is defined as the code sharing mechanism of choice in the product-line architecture.

Unlike .Net Framework or .Net Core, .Net Standard is not a CLI, but a set of standards that defines the compatibility with existing platforms. Software applications can only target a single platform, but libraries can target .Net Standard, and then be used by all platforms that are compatible with that standard. When designing a library, developers can choose a .Net Standard version to target. Based on the version they choose, the developers will have access to a varying amount of platform operations. Developers that target a low .Net Standard version, such as version 1.0, are able to use their library in applications targeting most CLIs, but have access to fewer operations [74]. Because the technological upgrade of the configuration and support environment aims at introducing new tools developed since the design of the legacy environment, a high version (2.0) of .Net Standard has been chosen as reference for the upgraded environment. This version is supported by all major C# CLIs, as noted in Table 3.1 [74]. This table contains all major C# platforms, as well as their minimum version compatible .Net Standard 2.0.

Table 3.1 - .Net Standard compatibility table, representing the minimum platform version for importing .Net Standard 2 libraries. Obtained from [74].

NET Standard	.NET Core	.NET Framework	Mono	Xamarin Android	UWP
2	2	4.6.1	5.4	8	10.0.16299

The aforementioned platform choices are represented in Figure 3.1. In this figure, executable projects are represented by larger box outlines. For each environment application, one executable project should exist per platform targeted. In these projects, only platform specific code should be provided, such as the *main* method, and implementations of platform specific operations. Following the chosen code sharing model, all platform independent code should be contained in the base application library, which is used by the executable projects. As discussed previously, the core assets should be implemented as .Net Standard 2.0 library projects and should be designed with focus on reusability. Finally, because the configuration and support environment of CNAO relies on several external commercial and open-source libraries, base application libraries, and standard operations libraries may use these external libraries.

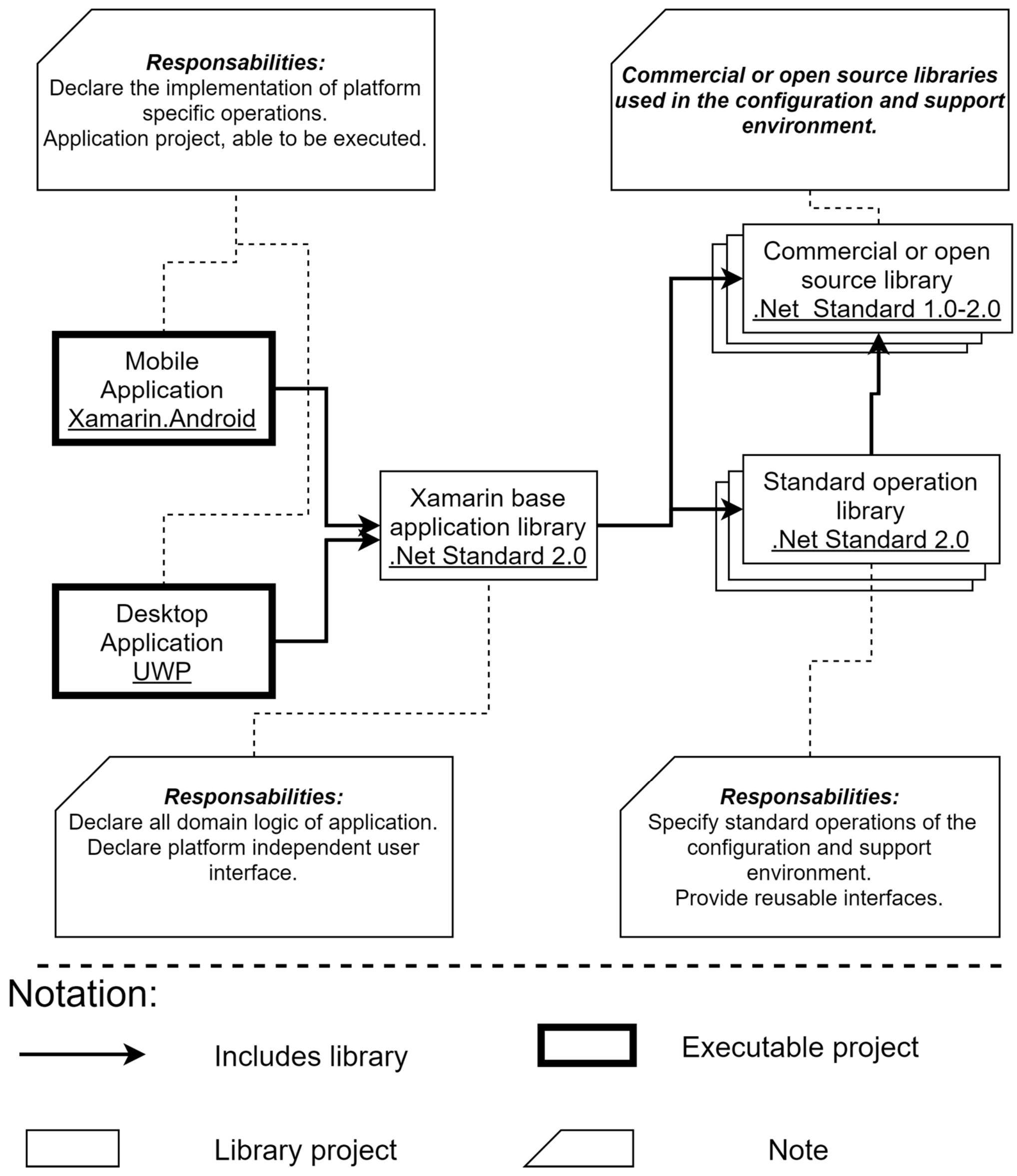

Responsabilities:
Declare the implementation of platform specific operations.
Application project, able to be executed.
Commercial or open source libraries used in the configuration and support environment.
Commercial or open source library
.Net Standard 1.0-2.0
Mobile Application
Xamarin.Android
Xamarin base application library
.Net Standard 2.0
Standard operation library
.Net Standard 2.0
Desktop Application
UWP
Responsabilities:
Declare all domain logic of application.
Declare platform independent user interface.
Responsabilities:
Specify standard operations of the configuration and support environment.
Provide reusable interfaces.
Notation:
Includes library
Executable project
Library project
Note

Figure 3.1: Allocation diagram representing the projects which compose an application.

Variability points specified in the diagram illustrated by in Figure 3.1 are as follows:

- Not all applications will target both platforms, therefore only one executable project may exist.
- The definition of platform specific interfaces is not mandatory, but when necessary these should follow the Dependency Service software pattern [75].

3.3.2 Application architecture

After the definition of technologies and CLIs to be used, a class diagram view of the product line architecture is presented. Figure 3.2 presents a class diagram which illustrates the class design of the product line applications, and the relationship between each application layer. It also describes some of the software patterns to be used, as well as the varying and non-varying application components. By understanding the product line architecture, configuration and support environment developers are aware of a list of software patterns and libraries that should be used in the application, and are shown which software elements should be created, and which ones should be generated by code generation tools.

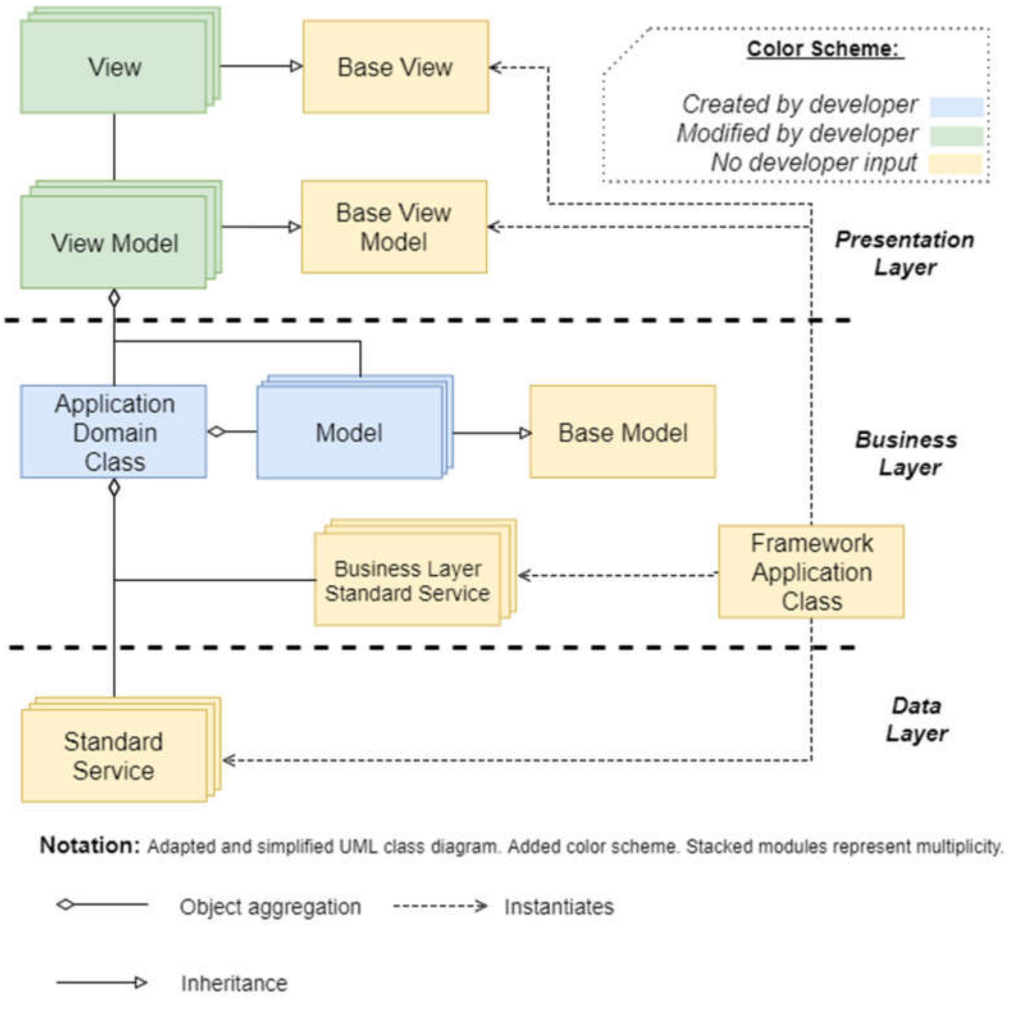


Figure 3.2: Class diagram view of the product line architecture of the configuration and support environment application.

The diagram presented in Figure 3.2 uses an adapted UML notation. In the diagram, 'classes' do not represent concrete classes, but types of classes, defined by their role in the application. Following the proposed product line architecture, applications' presentation layer should follow the MVVM software pattern [76]. The MVVM software pattern separates the application user interface into three types of classes, namely views, models and view-models. View classes declare the application's user interface elements, such as buttons, and forms. Following the MVVM pattern, View classes do not interact directly with view models, and have no dependency to the Model classes.

View-model classes are declared to serve as intermediaries between the Views and the application's domain model. Views subscribe to data, and, at run-time, view-model handlers are hooked into the view's subscriptions. This indirect interaction allows view-models and views to be executed and tested individually, without the presence of their counterpart. This software pattern promotes loose-coupling and allows greater isolation when performing unit testing. Compliance with this software pattern is enforced to aid the compliance of the R3. Model classes encapsulate state and information that should be presented. Under the MVVM pattern, view-model classes can update and call methods from models. In the opposite direction, models should be capable of raising events describing data changes, which are then provided to the view. Model classes are sometimes part of the application's domain layer, but preferentially these should only encapsulate data and provide validation techniques, such as Data-Transfer-Objects[77]. Figure 3.3 illustrates the relationship between components in the MVVM software pattern.

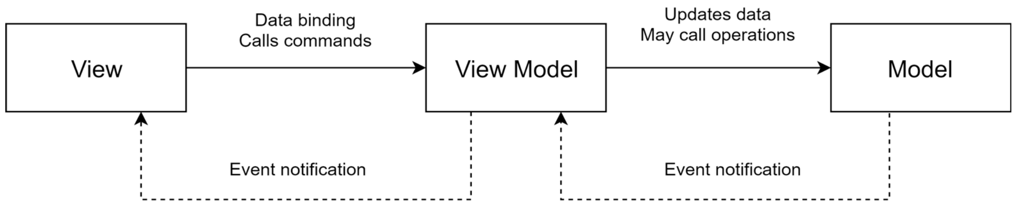


Figure 3.3: Diagram illustrating the MVVM pattern components, as well as their run-time interactions. Adapted from [77].

The Implementation of the MVVM software pattern usually relies on an MVVM framework. The framework is tasked with the wiring from the views to their respective view-models, and provides tools for performing data binding, invoking commands, and sometimes even supports page navigation. For this task, the Prism [76] framework was chosen. Additionally, base view, view-model, and model classes are provided by a framework developed during this work, AppBase2020. Developers should then inherit their views, view-models, and models, from the base classes defined in the AppBase2020 framework.

The framework application class manages the lifecycle of configuration and support environment applications. It has the responsibility of instantiation and initialization of core assets used by the application.

Finally, the product line architecture also includes the service classes, which are available to product line applications. These services, entitled standard services classes, are domain and data layer classes implemented by the standard service libraries.

3.3.3 Product line architecture services

In the previous section, the presented product line architecture class diagram view contains several references to standard service classes. In this section, we describe these services, and their role in the environment. A product line architecture defines several reusable core assets, which are implemented as reusable libraries to be imported by the application projects. By implementing the core assets as libraries, the core asset development team is able to maintain a greater control over their lifecycle. For example, if changes need to be performed to a core asset, then a new version of the respective library may be developed and deployed. From that moment, in order to update a group of product line applications, the library version can be updated, followed by the recompilation of the applications, and later deployment. Applications which cannot be updated regularly, or those not affected by the changes, can then simply not perform the update process. While the update process does require direct action from the control system team, this task is much less error-prone than altering existing applications by editing their source code.

Operations that were expected to be performed by multiple product line applications were selected as candidates for being packaged into reusable libraries. Additional criteria considered when defining the scope of the standard service libraries was whether these operations could be encapsulated into a well-defined abstraction. Two types of standard service libraries were defined. The first library type allows applications to interface with other components of the control system, such as the repository, or the OPC-UA equipment in the third level. These interface libraries were designed to fulfil the R7 (chapter 3). The second type of standard service libraries implements a set of commonly performed operations in the environment, such as configuration loading and validation, or event logging.

The following standard service libraries have been implemented in this work:

- **RDAS client** - Standard service library performing object relational mapping and interfaces with the control system repository service.
- **FDAS client -** Standard service library designed to interface with CNAO’s remote file servers.
- **CnaoLog** - Standard service library designed to perform local and remote logging.

- **Framework2020IdentityManager** - Standard service library designed to interface with the new authentication and authorization service.
- **OPCUASiprod** - Standard service library designed to perform communication with OPC-UA servers.
- **CnaoApplicationServices** - Standard service library that provides implementation of commonly used domain layer operations, such as reading and validating configuration files.

In the next chapter, the development of the client and server side of the services described above is presented.

Chapter 4

Standardization of services in the Configuration and support environment

In the early phases of the work, we defined a standardized way to perform several common operations in this software environment. These operations are, for example, repository access, definition and loading of configuration settings, and user authorization to resources.

The legacy implementation of most of these operations was defined alongside the design of the control system, in 2003. However, since then, the implementation of several of these operations in the legacy environment was found to be less than ideal. Therefore, the design of the upgraded configuration and support environment presented an opportunity to redefine the implementation of several of these operations. In addition, some operations, which did not possess a standardized implementation, such as remote logging, now possess a standardized implementation.

The most important activity of this work is creation of several libraries and services to support the applications of the upgraded configuration and support environment. These services and libraries have been developed to provide the implementation for the standardized operations.

Section 4.1 presents the service defined to provide applications with object-oriented access to repository data, named RDAS. Section 4.2 presents the design and development of FDAS, which allows applications to access remote files. Later, the authentication and authorization process developed for the environment is described in detail in section 4.4. Afterwards, the remote and logging services are described in section 4.5, followed by the configuration file management library, in section 4.6.

The last section of this chapter presents a sub-project that has been started in the later stages of this work. After the development of the initial versions of the environment services, it was decided that some LabVIEW applications, which belong to another environment of the control system, could consume services developed for this work. Therefore, a project was approved to facilitate the integration LabVIEW applications with some of these services. This section presents the tasks performed for the integration project, as well as its preliminary results.

4.1 RDAS - Relational data access service

The repository is the most important control system element that configuration and support environment applications interface with. As mentioned previously, the repository contains configuration data necessary to set up a large amount of the accelerator’s equipment. It is the main responsibility of applications of the configuration and support environments to manage this data, as well as, when requested, deliver it to the third layer. Because of this, permitting the communication of configuration and support environment applications with the repository is of the utmost importance. By designing and implementing a standard repository communication interface, we provide applications with an implementation of the rules to be followed when interfacing with the repository. Additionally, by enforcing the usage of the interface, we hope to increase the likelihood of safe operation, at the cost of limiting application versatility.

Because using the RDAS interfaces is mandatory, it is very important to design the interface broadly enough so that all necessary operations can be easily performed. If the designed interface is too complex or broad, it might be challenging for developers to use it correctly. Alternatively, if the interface is too restrictive, then developers might struggle to adopt it, or, in the worst-case scenario, forgo its usage. Additionally, the interface has to be supported throughout the lifecycle of the environment, independently of any changes in technologies used and architecture.

The product line architecture developed for this project dictates that communication with endpoints should be performed, when possible, following a service-oriented approach. Following this strategy, the repository should be wrapped around a service that manages communication, authorization, and logging. The applications then communicate with this service to interact with the repository. Afterwards, a C# library was designed to standardize the client-side operations for interfacing with the developed service. This library provides ORM capabilities in order to allow developers to develop their applications with an object-oriented approach. Finally, in order to reduce total development time, a generator tool was developed in order to generate a C# classes repository tables, which can then be used alongside the client-side library.

In this section, we present the design decisions made during the development of the RDAS library, the RDAS REST server, and the RDAS generator tool. However, cross-cutting concerns such as logging and authentication of all services will be aggregated into the dedicated logging and authentication sections.

4.1.1 Relational Data Access requirements

The main objective of the RDAS service is the standardization of the communication between control system applications and facility's repository. CNAO's relational data repository is an Oracle database cluster. While most application operations involve querying a single relational database table, several operations require complex operations. Both of these usage scenarios are supported by the RDAS library.

In summary, the RDAS service and libraries are designed to implement the following features:

- Local and remote (REST API) data access following a single, common interface.
- Allowing the execution of arbitrary SQL queries.
- Automatic generation of CRUD queries for classes that represent relational tables.
- Mapping relational query results into object instances.
- Development of dedicated methods for manipulation of large binary data present in the repository.
- Support for CNAO's repository legacy conventions and standards.
- Local resources should be secured by database credentials, while REST resources should be secured using OpenID Connect [78] authentication and authorization.
- Custom tool for automatically generating classes from SQL queries or Oracle tables.

4.1.2 Implemented solution overview

The developed solution was named Relational Data Access Service, or RDAS for short. The solution is composed of three distinct components:

- RDAS client library. The RDAS library is a .Net Standard 2.0 C# library that mediates the interaction between applications and the RDAS server. Additionally, it provides ORM capabilities.
- RDAS server. The RDAS server is a REST API that exposes the repository to applications in the configuration and support environment.
- Class generator tool. The class generator tool is a .Net Core application designed to browse the repository's schemas and generate C# classes, corresponding views and tables. The resulting C# classes, named DataObjects, can be used by the RDAS library to perform CRUD operations, and follow the model role of the MVVM pattern.

4.1.3 RDAS library

The RDAS library has two main objectives. The first objective is to allow applications to interface with RDAS server instances, requesting the execution of SQL operations in the repository. The second objective is to provide relational-object mapping capabilities, thus allowing applications to avoid using hardcoded SQL.

In order to allow the reader to understand the architecture of the RDAS library, we present a domain model of the library's architecture. The domain diagram below illustrates the most important domain entities and their relations.

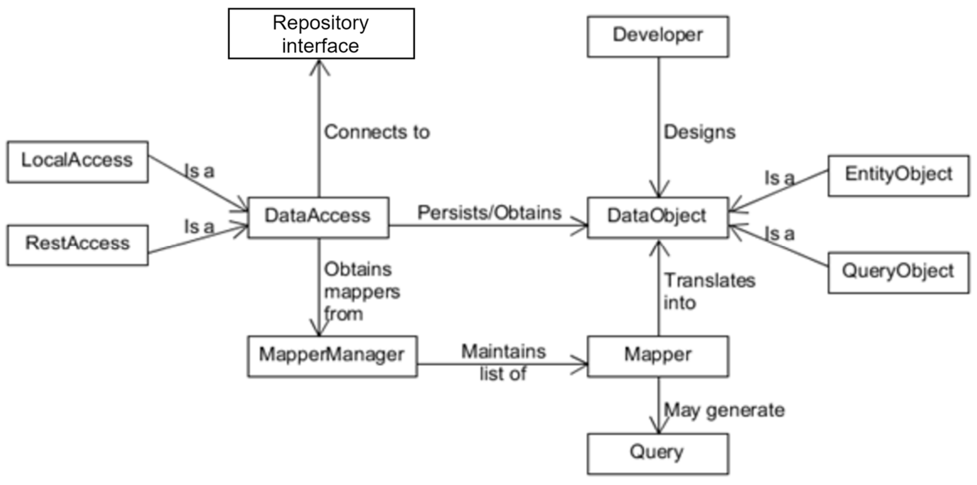


Figure 4.1: RDAS library domain model.

A brief description of each component illustrated is as follows:

- DataBase – A relational database containing data to be accessed. The data are distributed across one or many tables. Database credentials are required to establish direct connection.
- DataAccess – Entity that acts as the main interface and entry point to the library. Applications perform all repository data access through this entity.
- LocalAccess – Local implementation of the DataAccess. Designed for legacy applications located in the same local network as the repository.
- RestAccess – Remote implementation of the DataAccess using HTTP communication. Designed for regular applications of the configuration and support environment. Communicates with RDAS server.
- Developer – CNAO employee that develops control system applications using the RDAS library.

- DataObject – Family of classes used in the RDAS library. DataObjects should represent data contained in the database or expected result of database queries.
- EntityObject – DataObject that represents data contained inside a database table, or view. EntityObjects instances contain the data of one database row. However, it is up to the developer to decide which and how many columns are included.
- QueryObject – DataObject that represents the result of the execution of a SQL query. Each QueryObject instance contains one row of the result set.
- Mapper – Library class that has the responsibility of converting the service's relational result into DataObject instances. Mappers dedicated to EntityObject classes have the additional responsibility to generate SQL queries that perform CRUD operations.
- Query – A SQL query.
- MappingManager – Keeps track of relationships between DataObjects and Mappers. A MappingManager instance decides which mapper to use for each DataObject. Custom mappers should be registered with the MappingManager.

Definition of model types

The relational-object mapping capabilities present in the RDAS library allows developers to design classes for querying the database. Afterwards, the results of these queries are written into instances of the defined class. In the RDAS library, classes defined for storing query data from the database are referred to as DataObjects. These classes are not required to inherit or implement any interface, as all information necessary for defining database operations is present in property attributes and naming conventions. DataObject classes have to be annotated with C# attributes to inform RDAS mappers how to perform the translation from a relational to object-oriented representation. For this purpose, a series of attributes has been declared. These attributes allow developers to provide the context required by mappers.

In CNAO repository, changes to the database schemas are infrequent, and require an approval procedure. Because of this, unlike in many ORM libraries, the object-oriented representation is always modelled after the current database schema, and never the other way around. The RDAS service does not allow applications to change the repository's schemas. Following RDAS library guidelines, DataObject classes may be designed for two purposes: Representing a repository table or view, or representing a query to be sent to the repository. DataObjects representing tables or views are referred to as Entities, while those representing queries are referred to as Queries.

If the class is to represent a table or view contained in the schema, the class itself should be annotated with TableAttribute. Additionally, each of its properties that will receive a value present in one of its columns should be annotated with the ColumnAttribute. The latter allows the declaration of additional information about the column, such as column name, and whether these are primary or foreign keys. In addition, the attributes define information necessary for the RDAS library to follow the control system repository's conventions, such as the special handling of binary large data, and to treat the first column of a view as an indexing element.

Alternatively, Query DataObject classes, which should be annotated with the QueryAttribute, can be used to retrieve arbitrary SQL query results. Result sets of Query classes operations are translated to an object-oriented representation and then inserted into the Query instance's properties by the mapper. Figure 4.2 contains the definition of a sample Entity and a Query class. Several of the configuration settings required for performing the translation between object and relational representations can be instead defined from naming conventions, therefore many annotations' parameters are optional.

```
[QueryAttribute("SELECT NAME, AGE FROM OMA_TEST_SIMPLE
INNER JOIN OMA_TEST_BLOB
ON OMA_TEST_SIMPLE.NAME = OMA_TEST_BLOB.NAME")]
class OmaJoinQuery
{
      public string NAME { get; set; }
      public int AGE { get; set; }
      public OmaJoinQuery(){ }
}

[TableAttribute("OMA_TEST_BLOB")]
class DdasSimpleEntity
{
      [ColumnAttribute()]
      public string NAME { get; set; }
      ColumnAttribute(isBlob: true)]
      public byte[] DESCRIPTION { get; set; }
      [ColumnAttribute(isBlob: true)]
      public Image PROFILE_IMAGE { get; set; }
      [ColumnAttribute(isPrimaryKey:true)]
      public string OMA_TEST_BLOB_PK { get; set; }
      public DdasBlobEntity(){}
}
```

Figure 4.2: Sample entity and query data object classes.

Remote and local interface

While developers are expected to use the remote interface in regular scenarios, the library also defines a local implementation with the same interface. In the legacy environment, there was only the local connection to the repository. In the upgraded environment, a local implementation of the RDAS interface was developed to be used in special circumstances. The local interface is implemented on top of a commercial ADO.NET[79] component supplied by Devart [80]. Additionally, because of the configuration of the repository cluster, an Oracle Client program [81] has to be installed in workstations where applications using the local interface are executed. Therefore, only the remote interface can be used on mobile or multi-platform applications.

The IDataAccess interface declares a wide range of methods to perform CRUD operations using DataObjects, as well as other operations, such as logging events remotely, and obtaining information on the state of the repository. Transaction support is present, but with severe limitations due to REST stateless properties. Transactions have to be performed as one atomic operation, where all queries requested are provided to the remote repository interface in a single request, with only the final result returned. Further versions of the RDAS library may extend the DataAccess interface to implement transactions over multiple requests, but as of this moment, no applications in the configuration and support environment require this functionality.

Figure 4.3 presents a sample code to store an EntityObject instance, named DdasBlobEntity. In this case, only the 'Name' property was assigned a value, therefore all other columns of that table will be filled with the default values.

```
IDatabaseAccess dataAccess = ...;
//Create an entity object instance
DdasBlobEntity newRow = new DdasBlobEntity()
       { NAME = "Sample Name"};
//Insert the instance
string primKey = await dataAccess
       .InsertAsync(new List<DdasBlobEntity>() { newRow });
```

Figure 4.3: IDatabaseAccess interface sample usage.

Implementation and Operations

Figure 4.4 contains a class diagram in the UML notation illustrating the relationships between the various classes in the library. As part of operations involving DataObjects, the IDataAccess class relies on mapping classes to perform translation necessary for dealing with relational data in an object-oriented representation. These operations include the generation of SQL queries, obtaining information about tables or views, and transforming JSON formatted results into DataObject instances. However, the various implementations of the IDataAccess interface rely on the general mapper interface and are not able to select the appropriate mapper for each operation involving DataObjects. Therefore, this task is relegated to the MappingManager class. The MappingManager class follows the singleton pattern and was designed to maintain the appropriate mappers for each designed DataObject class. Additionally, the MappingManager implementation allows developers to define new custom mappers and register them. Additionally, MappingManager's registrations methods allow developers to define which DataObjects are to be mapped by custom mappers, allowing further versatility to final applications.

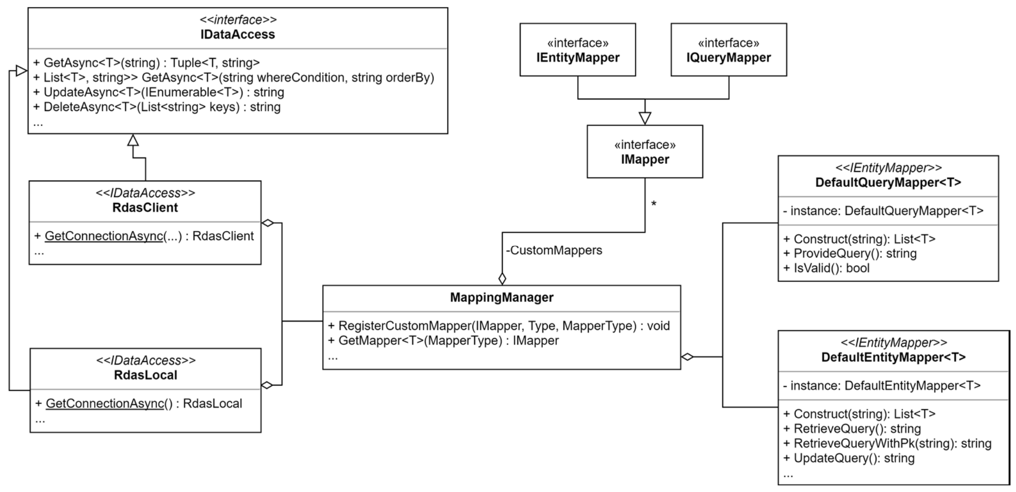

<<interface>>
IDataAccess
+ GetAsync<T>(string) : Tuple<T, string>
+ List<T>, string>> GetAsync<T>(string whereCondition, string orderBy)
+ UpdateAsync<T>(IEnumerable<T>) : string
+ DeleteAsync<T>(List<string> keys) : string
...
«interface»
IEntityMapper
«interface»
IQueryMapper
«interface»
IMapper
<<IDataAccess>>
RdasClient
+ GetConnectionAsync(...) : RdasClient
...
*
-CustomMappers
<<IEntityMapper>>
DefaultQueryMapper<T>
- instance: DefaultQueryMapper<T>
+ Construct(string): List<T>
+ ProvideQuery(): string
+ IsValid(): bool
MappingManager
+ RegisterCustomMapper(IMapper, Type, MapperType) : void
+ GetMapper<T>(MapperType) : IMapper
...
<<IDataAccess>>
RdasLocal
+ GetConnectionAsync() : RdasLocal
...
<<IEntityMapper>>
DefaultEntityMapper<T>
- instance: DefaultEntityMapper<T>
+ Construct(string): List<T>
+ RetrieveQuery(): string
+ RetrieveQueryWithPk(string): string
+ UpdateQuery(): string
...

Figure 4.4: UML class diagram detailing several RdasClient classes, and their relationship.

As can be seen in the class diagram, generic methods and generic classes are heavily present in the RDAS library. This is because the library is designed to allow developers to define their own DataObject classes. When an operation is called, such as the InsertAsync operation presented in Figure 4.3, both IDataAccess implementations request an IEntityMapper instance from the MappingManager, in order to generate the correspondent insert SQL query for each EntityObject instance received. Once the mapper is received and SQL query generated, the query is sent to the database (locally or remotely depending on the configuration) and results are returned. If the database operation returns relational data (such as the GetAsync method), the mapper would be used to generate one DataObject instance for each row. From the point of view of the application, whether the repository is being connected to locally or remotely makes no difference, and thus the IDataAccess interface can be used.

4.1.4 RDAS Server

Previously, applications of the Configuration and Support environment would access the repository directly. In the upgraded environment, in regular situations, only RDAS Server instances communicate directly with the repository. The responsibility of the RDAS server is to provide desktop and mobile applications access to the control system repository.

The design of the RDAS Server was closely linked with the RDAS library, namely the local repository access implementation. Both of these libraries rely on a legacy library named DataComponent for accessing the repository.

The DataComponent library class is designed to encapsulate all standards and operations necessary for interfacing with the repository directly. This library has been developed by porting and adapting a previous library of the configuration and support environment, named CTSIprod.DataComponent, which had been developed in the .Net Framework version 4 CLI. The CTSIprod.DataComponent library had two main issues that prevented it from being reused in the upgraded environment. The first issue was the change in development framework from .Net Framework to .Net Standard. The change in CLI meant that several operations were no longer available and had to be replaced with logically identical code. The second and most important reason was that CTSIProd.DataComponent was a domain layer library that freely performed presentation layer operations, such as opening dialog forms to request information from operators, and present error codes. By having dependencies to presentation layer WPF classes, the libraries components can only be tested using in a full application. Alternatively, the adapted DataComponent library only performs domain layer operations, and thus can be unit and integration tested by independently of any user interface.

All the mapping operations are performed by the client, so the RDAS server provides API that accepts SQL queries and query parameters. Similarly to other services developed, these APIs may require authentication and authorization, provided by the Identity Provider. At the server, several security measures are performed, including verification of user claims and permissions of the client application, sanitation of parameters.

4.1.5 Automatic DataObject Generation

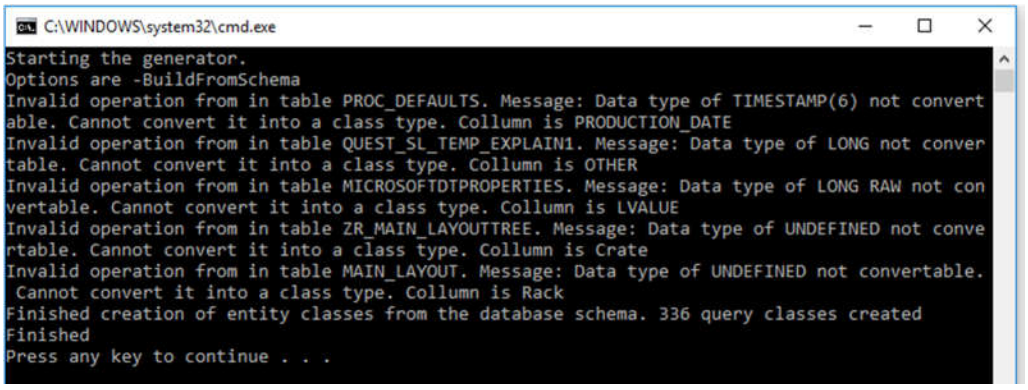
In order to simplify and aid the integration of the RDAS library into future configuration and support applications, a DataObject generation application has been developed. The RdasGenerator application is a command line utility that generates DataObject source code files based on provided configuration files.

The RdasGenerator tool allows developers to define which kinds of DataObject classes they require, and then use the generated C# files in their applications. Currently, the RdasGenerator tool allows for the generation of DataObjects in the following ways:

- Generate entities from repository schema. Using this option, a DataObject class will be generated for each view and table present in the desired schema. Alternatively, a list of tables or views can be provided. If the developer does so, then DataObjects are only generated for those tables or views.
- Generate query DataObject classes from a list of SQL queries. If this option is selected, all queries contained in the configuration file

are processed and for each query a DataObject class file is written allowing RDAS to execute the query and save all resulting values into properties.

DataObject classes generated follow all control system conventions regarding repository access such as correctly marked BLOB properties, view indexing, and foreign key lookup. As the RdasGenerator tool consults the repository to obtain information on the schema, constraints, and query results, the tool can only be successfully executed from a computer where local repository access is enabled. The resulting C# files are created in a customizable output folder, with one C# file written per generated class. During execution, any issues arising from invalid queries, or tables/views not following the expected convention, are presented to developers. Figure 4.5 contains a screen capture of the execution result of RdasGenerator, configured to generate models for each table and view in one of developer's repository schemas. As can be seen, a few tables and views have been found to not follow the convention due to using columns types not supported, and 336 DataObject files have been generated from the remaining tables and views.



```
Starting the generator.
Options are -BuildFromSchema
Invalid operation from in table PROC_DEFAULTS. Message: Data type of TIMESTAMP(6) not convert
able. Cannot convert it into a class type. Collumn is PRODUCTION_DATE
Invalid operation from in table QUEST_SL_TEMP_EXPLAIN1. Message: Data type of LONG not conver
table. Cannot convert it into a class type. Collumn is OTHER
Invalid operation from in table MICROSOFTDTPROPERTIES. Message: Data type of LONG RAW not con
vertable. Cannot convert it into a class type. Collumn is LVALUE
Invalid operation from in table ZR_MAIN_LAYOUTTREE. Message: Data type of UNDEFINED not conve
rtable. Cannot convert it into a class type. Collumn is Crate
Invalid operation from in table MAIN_LAYOUT. Message: Data type of UNDEFINED not convertable.
 Cannot convert it into a class type. Collumn is Rack
Finished creation of entity classes from the database schema. 336 query classes created
Finished
Press any key to continue . . .
```

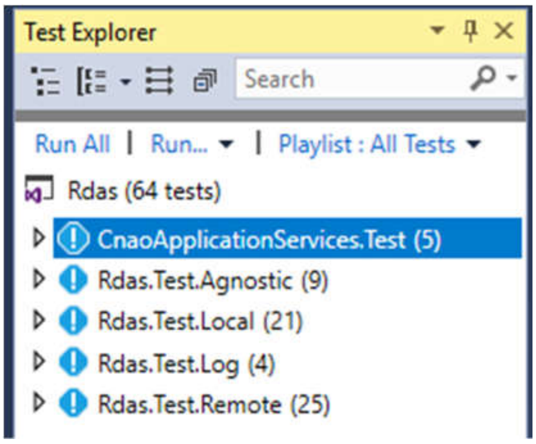
Figure 4.5: Results of the execution of RdasGenerator in the development repository.

4.1.6 Integration and Unit testing

In order to ensure the correct operation of the RDAS service and the correctness of the RDAS library, a series of unit and integration tests were also defined during the development.

The unit tests were created alongside the development of the RDAS library, guiding the development and ensuring that the requirements of classes were correctly fulfilled.

More importantly, a large set of integration tests were also created to ascertain if all control system components intercommunicate correctly during operation. The integration tests evaluate the result of operations performed between full applications, querying several tables in the development repository. The integration tests can be configured to also include the Identity Provider in the requests, thus testing if the security permissions are configured and evaluated correctly.



Rdas (64 tests)
CnaoApplicationServices.Test (5)
Rdas.Test.Agnostic (9)
Rdas.Test.Local (21)
Rdas.Test.Log (4)
Rdas.Test.Remote (25)

Figure 4.6: RDAS solution's unit and integration tests, categorized by test configuration.

4.2 Remote file access

Configuration and support environment applications are occasionally required to access files in other control system devices. In the legacy environment, applications were installed on machines with access to remote network drives, or FTP access. As defined by the product line architecture R1, which aims at the addition of mobile applications to be part of the upgraded environment, files stored in local network drives cannot be accessed by these applications.

The main objective of the design of the remote file data access service (FDAS) was to allow applications in all platforms to access remote files. Additionally, file access should follow the permission system defined by the new authentication and authorization system.

In order to achieve these objectives, the FDAS service was designed and implemented as part of this work. In this section, we discuss the terminology used in the FDAS service, as well as its behavior. Afterwards, we describe the model for defining shared folders and permissions. Lastly, the implementation of FDAS library and server is presented.

4.2.1 Terminology

Figure 4.7 illustrates the domain model of the FDAS library and describes how the FDAS service exposes files to clients. A brief description of the elements shown in the figure is as it follows:

- **FDAS server**: FDAS server instance.
- **File system**: A file management system organizes files inside a devices' storage, such as the device's hard drive partitions.
- **Virtual file system**: Abstraction created by the FDAS server, it exposes folders and files stored in the device's file system to clients.
- **FDAS client**: CNAO application that connects to a FDAS server instance using the FDAS client library.
- **CF2020 OpenID Connect Server**: Configuration and support environment identity provider that follows the OpenID Connect standard. Section 4.4 goes in detail into the environment's permission system.
- **Identity Manager Library:** Library used by client applications to interface with the identity provider.

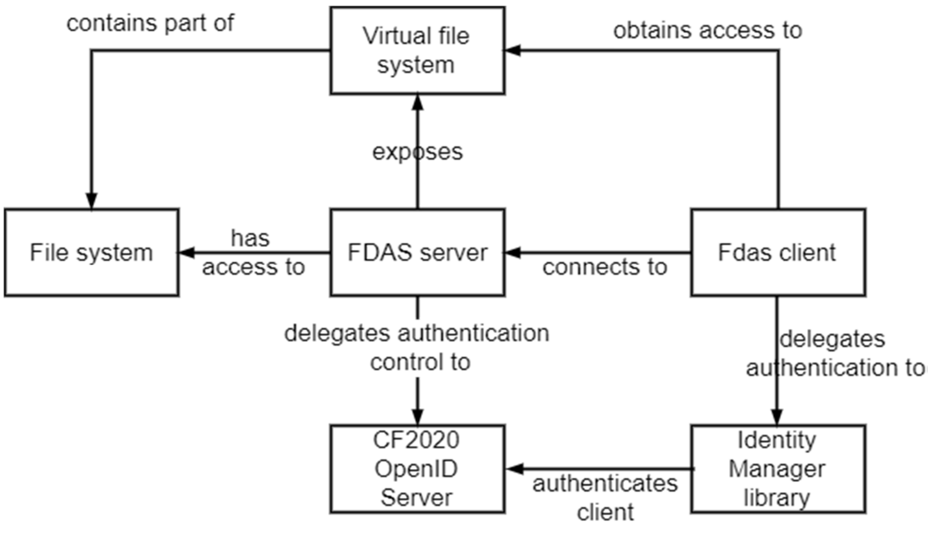


Figure 4.7: FDAS domain diagram, in UML notation.

4.2.2 Exposing files remotely

As mentioned in the previous section, the FDAS service has the purpose of exposing files contained in the server's folders to environment's applications via a RESTful API. In order to allow developers to define which folders are to be exposed, a virtual file system abstraction was defined. The abstraction defines how to inform the FDAS server application which folders are to be exposed, and how to present them to clients.

When configuring an FDAS server instance, developers are required to specify all server folders to be exposed, and provide prefixes to be assigned to each one of them. Figure 4.8 illustrates the mapping abstraction used when defining prefixes.

FDAS servers expose folders recursively, so all sub-folders and files inside a selected folder are also exposed. All exposed folders are given a prefix, and appear to clients as part of the same root folder, with the prefix added. FDAS clients are not informed of the real path of the files they remotely access. Instead they use the abstraction’s path, which is then translated into real file locations by the server.

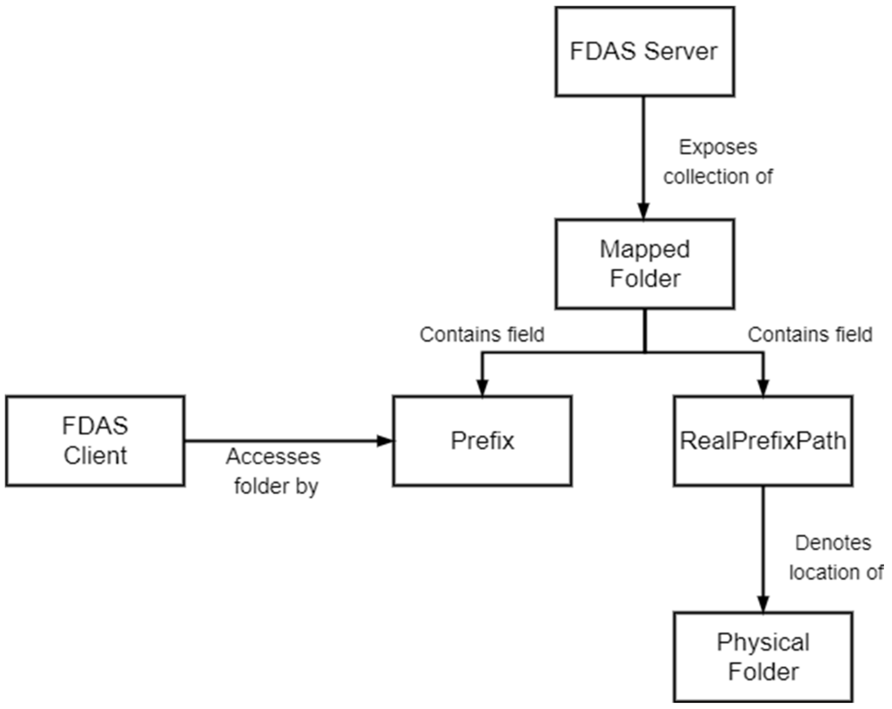


Figure 4.8: Diagram illustrating the mapping abstraction implemented by the FDAS library.

Figure 4.9 illustrates an FDAS server configuration file, declaring two folders to be exposed. Each JSON object contains a *Prefix* and *RealPrefixPath* field. The *RealPrefixPath* field represents the folder’s location in the server’s file system, such as *C:\\TestRepo*. By defining this field, the developer is defining which folders are to be exposed to FDAS clients. As mapped folders are exposed recursively, all files and folders inside the mapped folder are also exposed, and so on. The *Prefix* field denotes the name FDAS clients shall use to access the mapped folder.

```
[
  {
    "Prefix": "TestRepo",
    "RealPrefixPath": "C:\\\\TestRepo"
  },
  {
    "Prefix": "Shared",
    "RealPrefixPath": "\\\\svpvclfs40\\CNAO\\FrameWork2020"
  }
]
```

Figure 4.9: FDAS server configuration sample file, listing two folders to be shared with clients.

Following the sample configuration presented in Figure 4.9, from the point of view of an FDAS client, the FDAS service would create and expose a “root” folder, or ‘\’. Clients would then use the discovery API and find two folders, named “TestRepo” and “Shared”, in the root folder of the virtual file system. If a client uploaded a file named “image.jpg” in the “Shared” folder, and used the discovery API once again, they would see the file under the address “\Shared\image.jpg”. In the server’s side, due to the presented configuration, the image would have the location of “\svpvclfs40\CNAO\FrameWork2020\image.jpg”.

4.2.3 FDAS server configuration

The other configuration file required for instantiating an FDAS server is the app.config file, displayed in Figure 4.10. This configuration file defines the following configurations settings.

- The “AuthenticationAuthorityLocation” setting defines the address of the identity provider, as the permission control is delegated to the identity provider server.
- The AuthMaskPolicy settings define the permission values necessary for users to download and upload files. These permissions are represented by binary masks. Only users whose mask possess each required permission position with the binary value of “1” are allowed to download or upload files. The permission mask system has been part of the control system since it was designed, and each CNAO employee has their own permission mask. By adopting the authorization mask notation, FDAS uses an already established resource access notation.

Another relevant configuration setting present in this file is whether the FDAS service can behave as a repository for remote log files. Any FDAS service instance can be configured to work as a remote file access server, a logging endpoint, or both. The logging service is explained in detail in Section 4.5.

```
<appSettings>
    <!--Permission mask an user is required to obtain read access-->
    <AuthMaskPolicyRead  value="000000100000"/>
    <!--Permission mask an user is required to obtain write access-->
    <AuthMaskPolicyWrite value="000011100000"/>
    <!--Location of the CF OpenID Authentication Server-->
    <AuthenticationAuthorityLocation value="https://localhost:44353"/>
    <!—Whether this machine is enabled to receive logs-->
    <LoggingEnabled value="True"/>
    <!-- If logs are enabled, Log folder-->
    <LogFolder value="C://Logs"/>
    <Port value="51413"/>
</appSettings>
```

Figure 4.10: FDAS server app.config configuration file. Also included are xml comments, describing each configuration parameter.

4.3 OPC-UA communication

Software applications of the configuration and support environment often communicate with OPC-UA servers located in the control system's third level. It is through the third level's OPC-UA servers that applications of this environment interfaces with the fourth level of the control system. Consequently, porting the previous environment's OPC-UA client library was a necessary for this work.

OPC-UA communication is performed using a request-response approach. In the configuration and support environment, applications can only perform the client role when communicating with applications present in the third layer. Therefore, only OPC-UA client capabilities had to be developed for the upgraded environment.

In the early stages of this work, the viability of the legacy OPC-UA communication library was evaluated. The results of this evaluation were positive, consequently, no major architectural changes were required in the upgraded environment. However, as the upgraded environment's libraries should target .Net Standard rather than .Net Framework, the legacy OPC-UA client library had to be ported.

When porting C# legacy software, unsupported operations contained in the legacy library which was unsupported by the chosen CLI had to be replaced with equivalent ones. As a result, the entire library had to be reviewed. Capitalizing on this opportunity, we have also decided to perform the following changes to the architecture legacy OPC-UA library:

- All dependencies on user interface libraries were removed. These modifications allowed the upgraded library to be used by applications using other GUI libraries, or no GUI at all.
- The scope of the library was limited to the domain layer. Previously, some methods of the OPC-UA client library could optionally perform presentation layer operations. Following the

product line architecture for the upgraded environment, standard service libraries have to be restricted to a single layer.

- The newest version of the commercial OPC-UA communication library was used. The legacy version relied on a commercial library for the low level implementation of the OPC-UA protocol. Since its development, the company that provides this library, DataFEED, had been releasing several new versions of the underlying library, adding new features.
- The legacy library's interface was preserved. Due to the familiarity of developers with the legacy library's interface, the legacy library's API was maintained whenever possible. In cases where the legacy library's API depended on classes no longer used or available, facade classes were developed to mimic the operations performed by the missing classes.

4.4 Authentication and authorization

4.4.1 Security objectives

As a part of the effort to standardize the access to other components of the control system, several security requirements were established. Previously, all applications of this environment targeted the Windows operating system, and were connected to the local area network. However, the environment upgrade envisages the additions of applications running on top of mobile devices, which should still access the same resources. Due to this requirement, the previous authentication solution was no longer a viable option, and a new authentication and authorization protocol was required.

The previous security solution consisted of client side authentication, via the LDAP protocol [82]. Afterwards, the client application decided whether the authenticated user had authorization resources by accessing the repository, and checking the user's permissions. This process had been implemented as a C# library, and was incorporated into all secured applications of the legacy environment.

During the design of the upgraded environment, the previous solution was no longer acceptable for several reasons. The LDAP protocol is not available for the mobile devices. Additionally, only client side security opens the environment to vulnerabilities involving tampered or malicious clients. In the past, and as expected in the near future, all control system applications were developed by CNAO employees, and run on devices belonging to the facility. Nevertheless, authentication and authorization using an identity provider should allow the environment to, in the future, be used securely even by applications outside the local network.

The authentication and authorization solution chosen for this environment is based on the OpenID Connect [83] standard. By delegating the tasks of authentication, authorization, and user information access to an identity provider, clients are able to access resources securely and in an authorized manner.

In this section, a brief description of the concepts of authentication and authorization is presented. Afterwards, a brief description of the OpenID Connect standard and its terminology. Later, the security solution developed for this environment will be explained in detail. Finally, the Framework2020IdentityManager library is presented, which aims at allowing applications to interface with the environment's identity provider, and manage the client's authorizations and tokens.

4.4.2 Authentication and authorization

Often, the concepts of authentication and authorization are used interchangeably. However, the difference between them is important when securing software resources. In this section we present several important terms and their definition in software security. These definitions have been taken from the of the OAuth 2.0 [84] and OpenID Connect [78] documentation:

- "**Resource owner**: An entity capable of granting access to a protected resource. When the resource owner is a person, it is referred to as an end-user.
- **Client**: An application making protected resource requests on behalf of the resource owner and with its authorization. The term "client" does not imply any particular implementation characteristics (e.g., whether the application executes on a server, a desktop, or other devices).
- **Authorization server**: The server issuing access tokens to the client after successfully authenticating the resource owner and obtaining authorization.
- **Authorization code**: Authorization codes operate as plaintext bearer credentials, used to verify that the resource owner who granted authorization at the authorization server is the same resource owner returning to the client to complete the process.
- **Access token**: Access tokens are credentials used to access protected resources." [84]

- "**Authentication**: Process used to achieve sufficient confidence in the binding between the Entity and the presented Identity.
- **Claim**: Piece of information asserted about an Entity.
- **Entity**: Something that has a separate and distinct existence and that can be identified in a context. An End-User is one example of an Entity.

- **ID Token**: JSON Web Token (JWT) [JWT] that contains Claims about the Authentication event. It MAY contain other Claims.
- **OpenID Provider (OP)**: OAuth 2.0 Authorization Server that is capable of Authenticating the End-User and providing Claims to a Relying Party about the Authentication event and the End-User.
- **Relying Party (RP)**: OAuth 2.0 Client application requiring End-User Authentication and Claims from an OpenID Provider.
- **End-User**: Human participant. "[78]

From the previously presented definitions, a few distinctions can be pointed out. The first is the difference between authentication and authorization. While the OAuth protocol specification document [84] never explicitly defines the term authorization, it can be inferred that it refers to obtaining permission to access or use a resource. This definition does not necessarily presuppose authentication, in the way defined by the OpenID Connect specification document, which also is presented above. The authentication process is performed to verify that a specific entity possesses a virtual identity. Authenticating a client, which in this scenario refers to a software application, is a simpler task than authenticating an end-user. When attempting to authenticate an end-user, one has to take into account whether the end-user is actually present, and informed about the authentication process. Meanwhile, these considerations do not apply to software applications.

In the perspective of the OAuth standard, operations performed by a software client are done so on behalf of an end user, the software client requires explicit user authorization to access user data contained resources, and to perform actions in the resource on behalf of the user.

In software engineering, the terms client and user are often used interchangeably. When discussing matters of security in this work, we will use the term client defined above. A client is a software application that seeks access to protected resources, on behalf of a user. The user, by contrast, is a physical entity, who may or may not be present at the time, using the device where the client is executing. As described by Justin Richer [85], some software systems assume the access of a protected resource as equivalent a to proof of authentication. The underlying logic for this is that if the client obtained permission from the end user to access the protected resource, it must have obtained the permission to perform actions using its identity. However, systems that use authorization protocols, such as OAuth 2.0, equivocating authentication with authorization may lead to scenarios where authorization to access protected resources does not directly imply the user's well-informed presence, which is required for authentication.

With the difference of these two concepts in mind, while authorization is not sufficient for determining end user authentication, these two procedures are compatible, and can be performed alongside each other. Due to their compatibility, the OpenID Connect standard was built on top of OAuth 2.0, adding steps in order to ensure that authentication would also be performed.

Other important security terminology differences are those between authorization servers and identity providers. The first term comes from OAuth 2.0 to designate a server that issues access codes. Meanwhile, an identity provider is an extended authentication server, which can also provide proof of authentication via ID Tokens. Other terms not to be confused are ID tokens, authorization codes, and access tokens. ID Tokens are JWTs (JSON Web Tokens) signed by an identity provider and act as proof of authentication, additionally, they may contain claims about the authenticated user [78]. Meanwhile, authorization codes are strings that are given to clients as a result of the authorization grant, and that can be exchanged for an access token [86]. Finally, access tokens are encoded JWTs that allow clients to access resources, such as requesting a secure API, on behalf of end users [84]. In the OpenID Connect standard, all three are used.

For the upgraded configuration and support environment, authentication of users and authorization to access resources have been identified as security requirements, and thus we have decided to adopt the OpenID connect standard.

4.4.3 Access control solution implementation

OpenID Connect is a standard which defines operations, roles, authentication flows, and responses. In order to implement an OpenID Connect identity provider, developers are recommended to use frameworks which provide certified implementations of the standard. At the time we started the development, the most complete and widely supported software framework for developing an identity provider in the C# language was Identity Server 4.

The Identity Server 4 framework is an open source framework implementation of the OpenID Connect and OAuth 2.0 standards. It provides implementations for 4 out of 5 conformance profiles, and has been certified by the OpenID foundation [83]. Most importantly, while the framework implements the operations necessary for conformance with the standard, it also provides clear extension points for developers to insert their implementation of operations that are not standardized by OpenID Connect [87]. Examples of unstandardized operations are storage of user data in the identity provider, the algorithm of user authentication to the identity provider, and the definition and storage of claims.

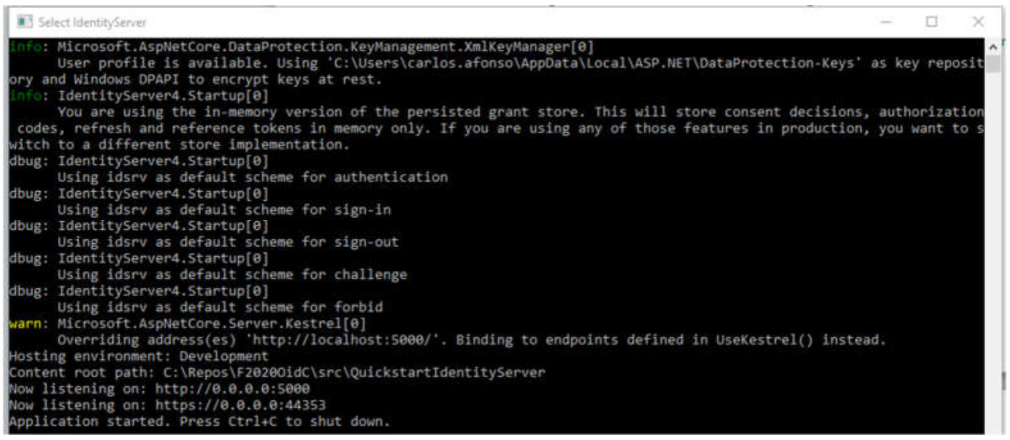
While default implementations to several extension points are available, developers can also easily integrate their own implementation of operations into the framework. On this work, several extension operations have been implemented in order to adhere to requirements regarding compatibility with the legacy environment, and to provide further services for applications of the upgraded environment. In this section, we describe the development process of the CNAO’s OpenID Connect identity provider.

Identity provider development

The development of the configuration and support environment identity provider consisted of two steps. The first step was selecting the most suitable base sample provided by the Identity Server online repository, and assuring that the OpenID Connect standard base features were working as required. At this point, requirements for the identity provider's extensions were obtained, so that extension points could be designed and developed. Finally, the developed extension points were integrated into the framework, resulting in an identity provider that complies with all requirements of the upgraded configuration and support environment.

The main requirement for the identity provider was the integration with the previous authentication and access control system. Another requirement was that all information necessary to configure and run the identity provider had to be stored in the repository. Therefore, the identity provider had to be able to obtain all configuration data directly from the repository. Finally, a key objective was to investigate whether OpenID Connect authentication could be used by applications to obtain security credentials for OPC-UA communication. Currently, OPC-UA communication in the configuration and support environment is performed via secure channel using application certificates [62]. Therefore, a secure way of transmitting application certificates was needed.

The base identity provider used was obtained from the Identify Server's GitHub samples [88] under the Apache 2.0 license. The startup sequence of the sample was changed to include the extensions, and the user interface was slightly adapted, with plans to further changes in latter stages. Figure 4.11 presents a screen capture of the resulting identity provider server executing in development mode.



```
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\carlos.afonso\AppData\Local\ASP.NET\DataProtection-Keys' as key reposit
ory and Windows DPAPI to encrypt keys at rest.
info: IdentityServer4.Startup[0]
      You are using the in-memory version of the persisted grant store. This will store consent decisions, authorization
 codes, refresh and reference tokens in memory only. If you are using any of those features in production, you want to s
witch to a different store implementation.
dbug: IdentityServer4.Startup[0]
      Using idsrv as default scheme for authentication
dbug: IdentityServer4.Startup[0]
      Using idsrv as default scheme for sign-in
dbug: IdentityServer4.Startup[0]
      Using idsrv as default scheme for sign-out
dbug: IdentityServer4.Startup[0]
      Using idsrv as default scheme for challenge
dbug: IdentityServer4.Startup[0]
      Using idsrv as default scheme for forbid
warn: Microsoft.AspNetCore.Server.Kestrel[0]
      Overriding address(es) 'http://localhost:5000/'. Binding to endpoints defined in UseKestrel() instead.
Hosting environment: Development
Content root path: C:\Repos\F2020OidC\src\QuickstartIdentityServer
Now listening on: http://0.0.0.0:5000
Now listening on: https://0.0.0.0:44353
Application started. Press Ctrl+C to shut down.
```

Figure 4.11: Configuration and support environment's identity provider executing in development mode.

Extensions

After the base identity provider was running with sample data, extension points had to be developed and integrated, in this section we document the extension points developed.

- Configuration data for running the identity provider changed from being hard-coded, into being present in the repository database. This data includes API and identity resources, client data, and persisted grants. In total, 18 new tables were added to the repository's database schema.
- Setting up of data access objects (DAO) using the Dapper ORM library to allow insertion and retrieval of previously created data from the repository.
- Development of a resource store to allow the identity provider to access API and identity resource information, overriding the default one.
- Development of a resource store to allow the identity provider to access client information, overriding the default store.
- Development of a custom user repository.
- Development of a custom profile service. The profile service is designed to obtain and provide claims about the user [89]. Claims are user data which are then sent in the JWTs, and take the form of key-value pairs. It is the responsibility of the profile service to provide all claims necessary. Usually, information such as the user's first name or user email is requested by relying parties. In this project, it has been decided that all ID tokens and access tokens shall contain the user's permission mask. The permission mask, as explained to earlier, describes several permissions each CNAO employee has.

Regarding the user repository, the Identity Server 4 framework does not define how user data should be structured. Therefore, when using a custom user repository, developers also have to implement and override all services that use user data. For the configuration and support environment, we have leveraged the existing employee data present in the repository to become user data. Employee data in the repository contains basic employee information, as well as employee's permissions.

The OpenID Connect standard defines that users must be authenticated, but does not explicitly define how authentication should be. For example authentication could be performed through user credentials, or verification of biometric properties using specialized equipment. Therefore, when using custom user classes in the Identity Server 4 framework, developers must also define a user authentication algorithm. For the configuration and support environment, user authentication is performed through their credentials. However, as user passwords are not stored in the repository, but by the IT department, this module will delegate password authentication to the IT department LDAP’s system once in production mode.

All extensions listed above are then integrated into the Identity Server 4 framework at the startup of the identity provider. This process, which is shown in Figure 4.12, is performed by several extension methods which configure each extension point used by our identity provider.

```
// configure identity server for configuration and support environment
services.AddIdentityServer()
       …
       .AddClientDapperPersistence()
       .AddResourceDapperPersistence()
       .AddGrantDapperPersistence()
       .AddCnaoUserPersistence();
```

Figure 4.12: Code snippet of identity provider startup, including the extension methods that configure the extension points.

4.4.4 Protected resources

As defined in the OpenID Connect standard, clients interact with an identity provider in order to authenticate the end-user and obtain access tokens, which can then be used to access protected resources [83]. In this section, we describe how protected resources in the configuration and support environment participate in the security workflow. In the upgraded configuration and support environment, services must be configured to delegate authorization control to the identity provider. This can be done during the service startup with only a few lines of code using the IdentityServer4.AccessTokenValidation library and is presented in Figure 4.13.

```
services.AddAuthentication("Bearer")
      .AddJwtBearer("Bearer", options =>
      {
             options.Authority = "https://localhost:44353";
             options.RequireHttpsMetadata = true;
             options.Audience = "rdas";
      });
```

Figure 4.13: Code snippet displaying the process for protected resource to delegate access control to the identity provider.

Afterwards, each protected API should be configured with the permissions required for their usage. When a client attempts to access a secured API that uses bearer token authorization, the client should include in the request the access token received from the identity provider. The protected resource then validates the access token following the OpenID Connect standard [83].

As defined in the OAuth 2.0 specification, scopes “allow the client to specify the scope of the access request using the ‘scope’ request parameter”[90]. During authentication, the user is informed of the scopes requested by the client. In order for clients to access resources, which can be either API or identity resources, the clients need to request the scope designated to the resource. For example, in the RDAS service, protected APIs are either categorized as requiring the “rdas.read_only” scope or the “rdas.full_access” scope. The RDAS service performs this validation by defining custom authorization policies. Then, each API is annotated with the authorization policies that must be validated before the API can be executed.

An additional validation step developed in this work was the validation of the user’s authorization mask by protected resources. Each CNAO employee is assigned a permission mask in the repository that defines which protected operations they have access to. This permission mask is a list of binary numbers, which contains a ‘zero’ for each operation they do not have access and a ‘one’ for each operation they have access to. As mentioned previously, the profile service developed for this project injects a claim containing the user’s permission mask in all tokens generated.

This validation step is not performed for all protected resources, as not all protected services require one of the permissions represented in the permission mask. Additionally, by definition only employees are assigned permission masks, and yet, clients may also request access tokens. Some protected resources in the configuration and support environment do not require user authentication to be accessed, such as the logging service. In this case, these resources still require a valid access token, but these tokens can also be acquired by client applications, using the client credentials grant.

As an example of permission configuration, consider hypothetical configuration and support environment application which is used by administrators to manage repository data. In this example, this application should be registered with the identity provider, containing its own client credentials, and access to the scopes “openid”, “rdas_log”, “rdas.full_access”. This application can use the client credentials grant to obtain an access token which contains the scopes mentioned previously. Meanwhile, the RDAS service requires user authentication with an adequate permission mask for managing repository data, but only the presence of the “rdas_log” scope in order to access the logging API. This repository management application can thus log data remotely as soon as it is initialized using an access token obtained with its own credentials. However, in order to display and manage repository data, the application requires a user to authenticate.

In the previous example, under no conditions this application can access the remote file service, FDAS, as the APIs present in the service require the “fdas” scope. Even if the application’s user has permission to access the FDAS service, as application is not configured to be able to request the “fdas” scope, the application will not be able to access an FDAS server’s API.

Because each secured resource provider can be configured with which scopes and permission masks are required to access each one of its APIs, this system allows a great deal of versatility in securing resources.

4.4.5 Framework2020Oidc library

The OpenID Connect standard supports several different workflows for validating authentication and authorization. The OpenID Connect specification document defines three code flows for authentication of users. These are the authorization, implicit, or hybrids flows. Each of these flows authenticates users by communicating with them directly, usually through a web browser. In the current implementation of the CNAO identity provider, the workstations available in the control room are not fitted with a web browser. This means that the three previously mentioned authentication flows cannot be used in the near future.

Therefore, it was decided to use the Resource Owner Password Credentials grant defined OAuth 2.0. In the Resource Owner Password Credentials grant, the user provides the client their credentials, and the client in turn uses these to obtain tokens from the event provider. This grant is supported by the Identity Server 4 framework [91], but does have a drawback: since the authentication is on behalf of the user by the client, the user should have a high degree of trust in the client application [90]. However, since all client applications are developed by CNAO control system developers, users can be assured that the credentials provided by the application are only used on their behalf.

Nevertheless, it is possible that future projects investigate the use of dedicated or embedded browsers in the configuration and support environment. As a result of these investigations, proposals can be made for changing the identity provider's authentication algorithm in order to adopt one of the OpenID Connect browser authentication flows.

With the objective of standardizing the interaction of configuration support environment applications with the newly developed identity provider, the Framework2020OIDClient library was designed. This library was designed to perform two operations for control system applications. First, it performs all communication with the identity provider, and exposes only an object-oriented interface. These involves authentication, obtaining and refreshing tokens and claims. The second role of this library is to manage identities used by clients. This is performed by keeping track of the tokens received, making the claims requested available, and ensuring that bearer API tokens are included when the applications requests the use of protected resources. Consequently, an application may have more than one "*identity*" at the same time, such as the client and user identity, with separate permissions and scopes for each of them.

The Framework2020OIDClient library uses the IdentityModel library [92], which is a relying party library certified by the OpenID Foundation [93], and is distributed under the Apache 2.0 license.

Figure 4.14 contains an UML class diagram of the Framework2020OIDClient library, focusing on the interface that it provides to configuration and support environment applications. This interface is the point of entry of the library, and is expected to be used by not only domain layer classes of final applications, but also by other configuration and support environment libraries that communicate with protected resources, such as the RDAS and FDAS libraries.

Classes implementing the IResourceOwnerIdentityManager interface must be initialized, and during the initialization, the connection with the identity provider is verified. Afterwards, authentication methods can be called, such as the AuthenticateAsync and AuthenticateAsyncClient. These methods are used to authenticate a client or a user by providing their credentials, and requesting the desired authentication scopes. As mentioned previously, the scopes define which operations the user intends to seek permission to access, or which identity information the user seeks to share to the client.

Once authentication has been performed, resources can be accessed via a RESTful API, by requesting a C# HttpClient instance. The resulting HttpClient instance contains a bearer token of the desired identity. Several other operations are also available after authentication, such as requesting claims, or requesting the identity's repository identifier. While not present in the class diagram due to size constraints, several methods may be called using a parameter of type ChosenIdentity. This optional parameter allows the application to specify which identity to be used in each operation. Valid options are the user or client identity, or optionally, the identity with the highest permission.

In general, methods defined by this interface also return the current library status, which contains the state of the manager, and whether errors have occurred in previous operations. In addition, the library also controls a lifetime of the authentication, and refreshes the acquired tokens if possible and necessary.

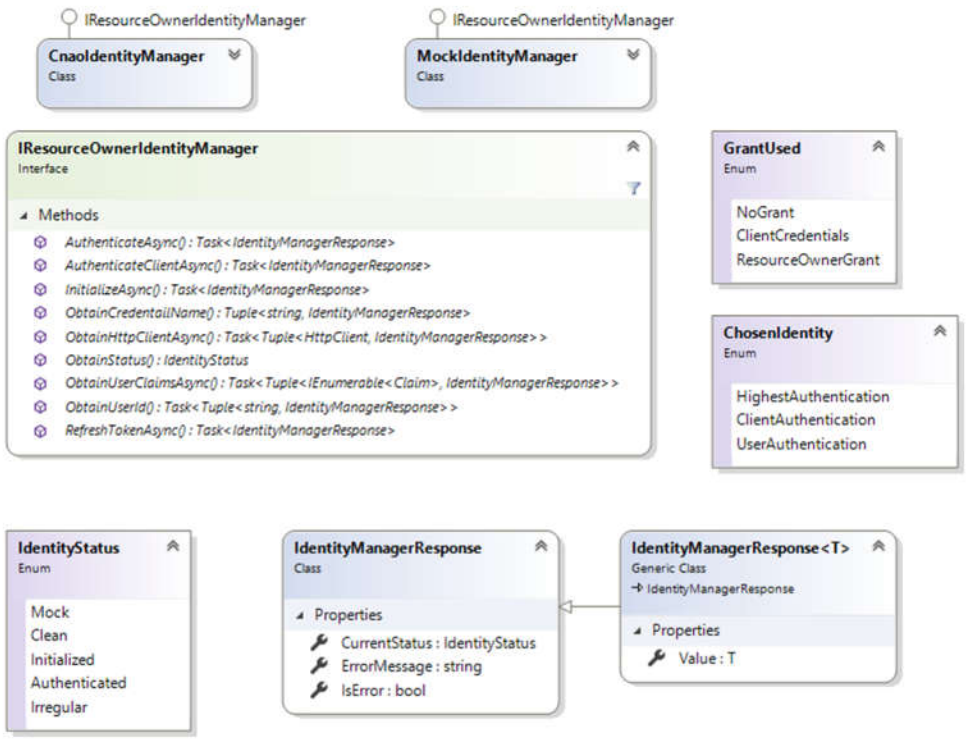

IResourceOwnerIdentityManager
CnaoIdentityManager
Class
IResourceOwnerIdentityManager
MockIdentityManager
Class
IResourceOwnerIdentityManager
Interface
Methods
AuthenticateAsync() : Task<IdentityManagerResponse>
AuthenticateClientAsync() : Task<IdentityManagerResponse>
InitializeAsync() : Task<IdentityManagerResponse>
ObtainCredentailName() : Tuple<string, IdentityManagerResponse>
ObtainHttpClientAsync() : Task<Tuple<HttpClient, IdentityManagerResponse>>
ObtainStatus() : IdentityStatus
ObtainUserClaimsAsync() : Task<Tuple<IEnumerable<Claim>, IdentityManagerResponse>>
ObtainUserId() : Task<Tuple<string, IdentityManagerResponse>>
RefreshTokenAsync() : Task<IdentityManagerResponse>
GrantUsed
Enum
NoGrant
ClientCredentials
ResourceOwnerGrant
ChosenIdentity
Enum
HighestAuthentication
ClientAuthentication
UserAuthentication
IdentityStatus
Enum
Mock
Clean
Initialized
Authenticated
Irregular
IdentityManagerResponse
Class
Properties
CurrentStatus : IdentityStatus
ErrorMessage : string
IsError : bool
IdentityManagerResponse<T>
Generic Class
IdentityManagerResponse
Properties
Value : T

Figure 4.14: Framework2020OIDClient library class diagram.

Finally, client applications in the configuration and support environment are expected to show or hide certain user interface options from users, depending on their permissions. For example, a non-administrator user should not be able to see advanced operations that they have no reason to perform. This can be done by checking the permission mask claim present in ID tokens.

4.4.6 OPC UA integration

In the context of this work, a possible integration between the OPC-UA security workflow with the OpenID access control was also investigated. Currently, applications in the configuration and support environment may act as OPC-UA clients in order to communicate with other control system components. The OPC-UA security model used currently in the legacy environment's applications does not authenticate users. However, in order to do so in the future, using the same OPC-UA workflow, the identity provider service was extended with security certificate management capabilities.

The OPC-UA protocol allows application instances and users to authenticate themselves using X.509 security certificates, which have to be trusted by the other party, or signed by a trusted certification authority [62]. In order to allow applications and users to obtain trusted certificates for communication, the CNAO identity provider has been extended with certificate signing capabilities, thus providing it with certificate authority capabilities.

In order to do so, the identity provider has been given a root certificate, which needs to be installed in the trusted certificate list of the OPC–UA servers. Once configuration and support environment applications perform one of the authentication grants, while requesting an OPC–UA certificate scope, they can then use the access token previously requested to obtain an X.509 certificate. The obtained certificate will have been signed using the identity provider's root certificate, thus being trusted by the OPC–UA servers. The certificate will be generated alongside the public and private key, and will be stored in the repository for further sessions, if an extended expiration certificate is selected.

The OpenID Connect UserInfo endpoint does not allow requests containing arbitrary parameters, so the private keys are generated by the identity provider. As part of this feature, the user and client tables present in the repository were modified to be able to store certificate and information regarding it, so that if a valid certificate is found, there is no need of generating a new one. Figure 4.15 contains an UML sequence diagram depicting the communication between the various entities to allow client applications to obtain security certificates for OPC–UA communication.

A possible variation of the presented solution would be allowing the client applications to generate the key pair, as well as a certificate request. The certificate request would then be sent to and accepted by the certification authority. This has the advantage that only the client application would ever have access to the private key. However, this would not allow the same certificate to be stored and reused later in another device. A dedicated certificate authority application could also be used. This certificate authority would be independent from the identity provider and act as a protected resource. This would allow additional versatility to the types of requests that could be performed, while at the same time guaranteeing authentication and authorization, as the certification authority would delegate access control to the identity provider.

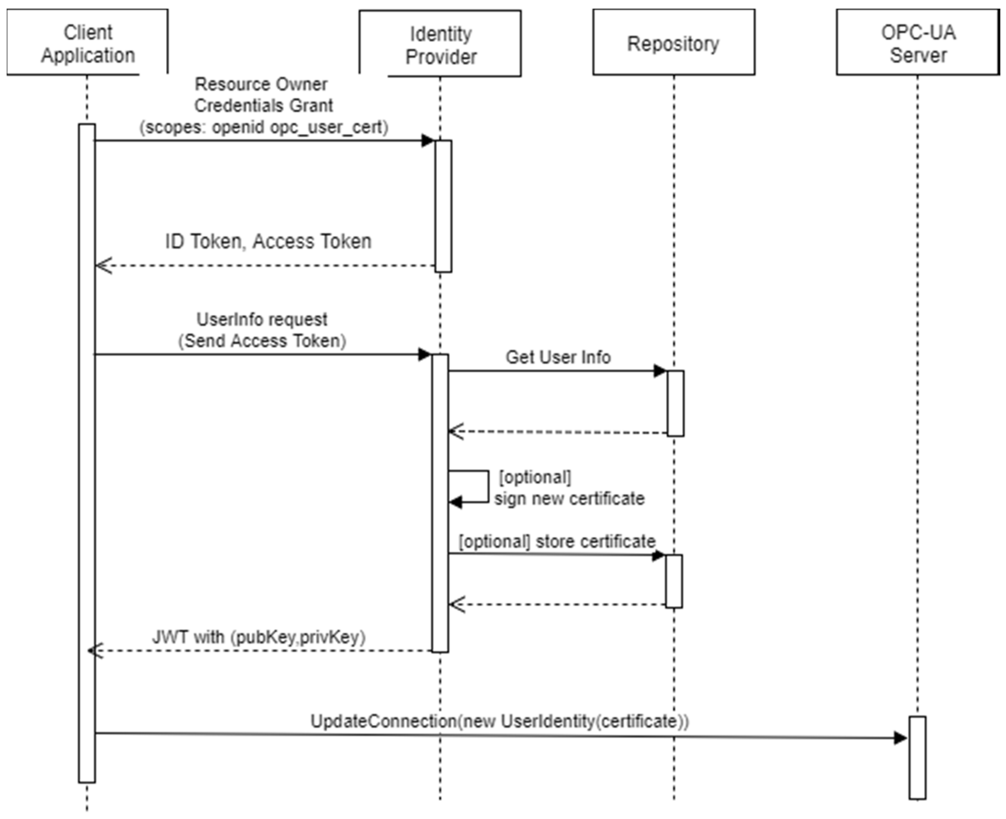

Client Application
Identity Provider
Repository
OPC-UA Server
Resource Owner Credentials Grant (scopes: openid opc_user_cert)
ID Token, Access Token
UserInfo request (Send Access Token)
Get User Info
[optional] sign new certificate
[optional] store certificate
JWT with (pubKey,privKey)
UpdateConnection(new UserIdentity(certificate))

Figure 4.15: UML sequence diagram depicting the workflow necessary for a client application to obtain a user certificate from the identity provider, and later use the certificate to validate the user identity.

As previously mentioned, applications of the configuration and support environment do not authenticate themselves when communicating with OPC-UA servers currently. At the present moment, OPC-UA servers possess a security certificate in order to perform encrypted communication, but do not require applications or users to perform any sort of authentication whatsoever. In order to enable or enforce application instance or user authentication, the servers would have to be reconfigured or redesigned. These servers are also not part of the configuration and support environment. Therefore, making changes to them would require a broader project scope, adding several stakeholders, as well as involving several other development groups.

The aim of this implemented identity provider extension was to provide a proof of concept that could be expanded upon in the future. The main issue encountered was that the generated certificates would be transferred to the client during each application execution. For this scenario, very short term certificates, tailored to each expected session length could be generated at every execution.

Security certificates are usually not meant for short time periods such as an individual user's session, but instead something that would be transferred to the user once, secured by them, and used for multiple sessions. An option to reuse these certificates would be to store them securely in the repository database, and transfer them to clients whenever requested until their expiration. The solution relies on the user trusting the client application to remove the certificate once the execution is over, as well as relying on the long-term usage of certificates. Long term certificate usage should also warrant the existence of certificate revocation lists, kept by the identity provider and consulted by servers. However, as mentioned previously, the OPC-UA servers are not part of the configuration and support environment, and thus the project's scope would have to be increased further.

The *OPC Unified Architecture* book, by Mahnke et al. [62], discusses several ways of organizing certificate management and delegation to OPC-UA enabled devices. The proposal briefly investigated in this work, consisting of providing CA capabilities to an identity provider will require further study regarding its viability and necessity in CNAO's control system in the future.

4.5 Local and remote logging

An important part of application development and maintenance is ensuring their correct operation. During initial development stages, developers have access to debugging tools and source code analysis. However, as the project lifecycle progresses and the final applications are deployed, such methods of analyzing the behavior of applications become unfeasible. Therefore, methods of analyzing runtime behavior become increasingly important, and one such method is event logging.

The upgraded environment logging libraries are designed to provide a standardized way of configuring and deploying logging capabilities to applications. In addition to standardizing the logging format and configuration, the developed CnaoLog libraries aim to provide implementations for logging events into all required control system endpoints, such as the repository, and remote file systems.

In this section, we discuss the main architecturally significant decisions made during the development of CnaoLog libraries, and describe its implementation.

4.5.1 Logging libraries design

The main objective of the CnaoLogging libraries was to provide a standard way for applications of the configuration and support environment to perform event logging, as well events logging into a centralized location. Because software applications of the CNAO control system are deployed in several different types of devices, which may be in different local networks, currently there exists no central repository for all environment applications. For some applications, logging is performed locally only, thus requiring access to the application's device in to access the logs. A centralized logging approach would hopefully allow an easier processing of logs, leading to better troubleshooting and hopefully thus shorter application downtimes.

The upgraded configuration and support environment is expected to contain several types of applications, with several different usage patterns and devices targeted. Currently, many of these applications are launched only occasionally, and used for very short sessions. Meanwhile, other applications, such as the equipment monitoring applications, are expected to run continuously for very long amounts of time. The long term objective of the logging system is to create remote locations where logs from all applications are expected to be kept.

As the applications of the new environment are expected to run in more than one type of device, and may not have access to every service, the CnaoLog libraries were designed as a series of libraries. The main logging library, CnaoLog, is mandatory and should be a dependency of all applications of the new environment. This library contains the logging abstractions, and a default local logging implementation. The optional libraries, which contain names that derive from CnaoLog, such as CnaoLog.Rdas and CnaoLog.Fdas, are libraries that depend on the main CnaoLog, and implement the abstractions defined by the main library in different ways. By using an optional library, and initializing the components of that library, application logs will be stored in one additional location. These libraries are optional because the extension libraries depend on other libraries and services that the application may not use. For example, not all applications are expected to have repository access, therefore, there is no reason to use the CnaoLog.Rdas library, which implements local logging to the repository.

The main CnaoLog library is quite simple, as it does not actually define how logging is performed. This library defines an interface for configuring a logging system, and several methods to log messages and exceptions, but does not actually implement the logging, instead acting as a facade. The logging itself is implemented by a mature logging framework which is widely used in the C# development community, named Serilog [94].

The Serilog library implements logging with a single API, that can then be stored in various formats, and sent to several different endpoints [95]. The Serilog library was chosen due to a combination of widespread adoption in the .Net development community, and possessing a modern and simple API. The Serilog library is licensed under the Apache 2.0 license.

The main reason for using an underlying implementation of logging was that currently developed logging libraries implement all necessary requirements for the applications of the new environment. Any future updates of the library may also be integrated into future versions of the CnaoLog libraries. However, a decision was made to implement a façade over the Serilog API, restricting the possible configurations and logging options. This design choice was performed for two reasons. First, the Control System group coding standards define that the interface of libraries should be developed by members of the Control System group. This allows the developers to take ownership of the interface, and thus control its evolution. Second, by implementing a façade, many optional API features that are not recommended to be used in the environment are hidden from application developers, thus making the library easier to learn and use.

Serilog defines the concept of Sinks to allow application classes to perform event logging without knowledge of where and how these events are going to be logged. The sinks are configured during the application initialization. Later, whenever an event is sent for logging, each sink receives the event and processes it as it sees fit. Because not all applications may want to log events to every endpoint, not all applications will configure and initialize every sink available to them.

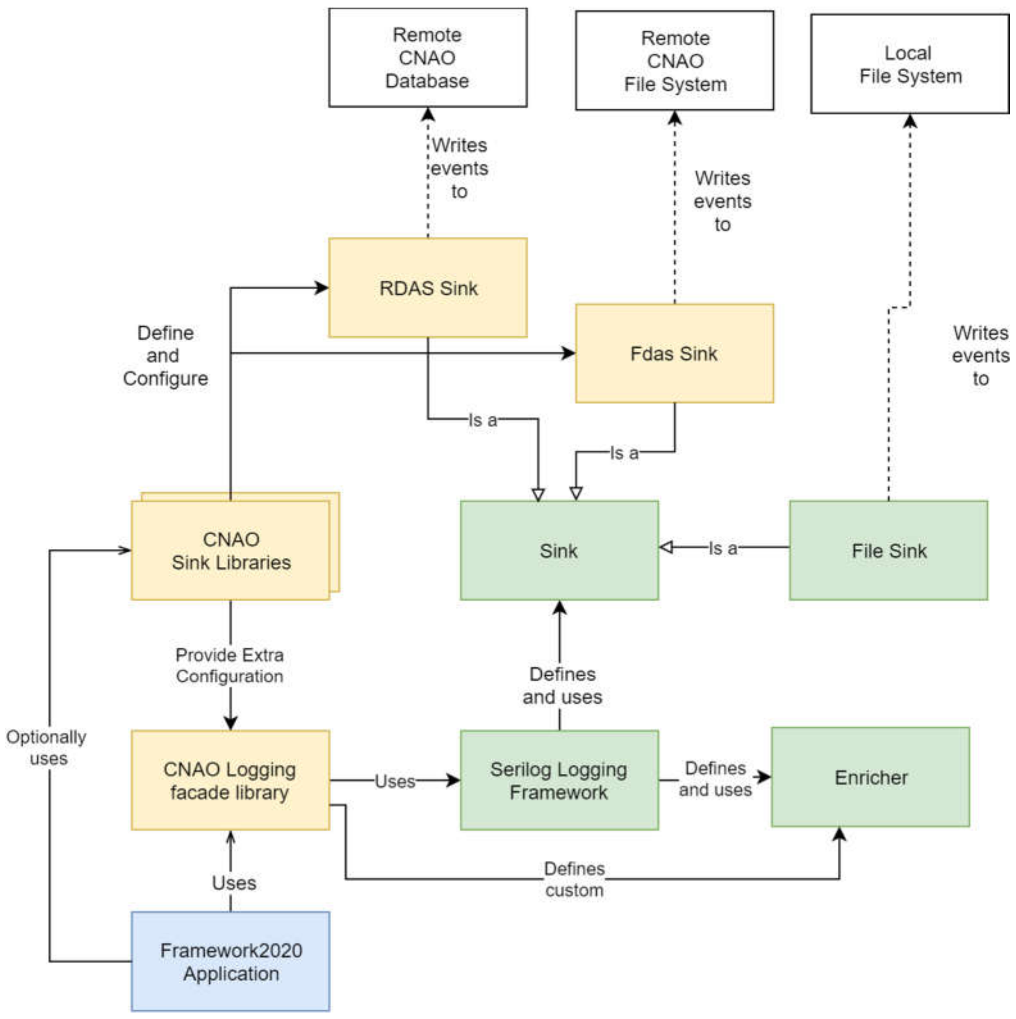


Figure 4.16: Framework2020 Logging domain diagram.

Figure 4.16 presents a domain model diagram illustrating the usage of CnaoLog libraries in applications of the upgraded environment. A description of each element referenced in the diagram is presented as follows:

- **Framework2020 Application**: Slow environment control system application made for the upgraded environment.
- **CNAO Logging façade library**: Library defining the environment's logging abstraction. Defines logging interface and configuration.
- **Serilog logging framework**: Open source, extensible .Net logging library.
- **Sink**: Serilog extension designed to process application logs, sending them into their configured destination.
- **Enricher:** Serilog library extension designed to add additional information to events logged. Enriched properties can be added to the logging framework during the application's start-up, or dynamically.

4.5.2 Implementation of CnaoLog libraries

This section describes the implementation of the CnaoLog libraries, highlighting how the architectural decisions shaped the implementation. Later, an example code sample showcasing event logging will be presented and explained in detail.

Base CnaoLog library

The base CnaoLog library has two main objectives, the first being the declaration of the interface for initializing the logging system, and performing logging. The second goal of this library is to define a static class that can be used to provide access to the log object to every other class in the application.

The interface that the CnaoLog library defines for initializing and performing logging operations was based on the Serilog API, and limits the configuration options available. By doing so, the interface works as a façade. Additionally, the CnaoLog library provides a base implementation for the interface, which relies on the Serilog library. Applications depending on the CnaoLog library are restricted to interacting only with the CnaoLog interface. This allows future implementations of the CnaoLog library to abandon the Serilog implementation in favor of another logging solution, without affecting the code of any application developed.

Optional Sinks

The optional sink libraries were designed with the objective of extending the base logging to standard services, which require the usage of their respective client libraries, without introducing references to unused libraries in any application. Otherwise, if the remote service sinks were defined as part of the base library, the base library would depend on the client libraries for these services. If that were to happen, all applications using the CnaoLog base library would depend on all standard service client libraries, regardless of whether the applications uses these services. The requirements analysis performed for upgrade of the configuration and support environment noted that not all configuration and support environment applications use every standard service. Therefore, several applications would contain unneeded dependencies, increasing loading time and file size.

It was decided that configuration and support applications should not be expected to depend on client libraries standard services that they do not use. Therefore, it was decided to split the logging into several libraries, composed of a mandatory base library, and several optional sink libraries. All applications should depend and use the base CnaoLog library, which allows local file system logging. Additionally, if developers require to perform remote logging into the standard services, they need to include a dependency to the respective optional sink libraries. Consequently, by segregating the sinks into optional libraries, applications are able to depend only on the client libraries of services they plan to use.

Figure 4.17 contains a diagram that illustrates the behavior of the RDAS standard service sink, which provides access to the repository database. In the diagram, it can be seen that even for logging operations, the RDAS server still requires permission from the identity provider in order to allow access to the logging APIs. Since, according to the product line architecture developed for this project, logging is a required functionality for all applications, including those with no user authentication, logging APIs in standard services only require client authentication. Consequently, all applications of the configuration and support environment are defined to possess logging permission. Consequently, all applications in the upgraded configuration and support environment will be able to log events if required, but unauthorized applications in the same network will still not have access to these protected resources.

During the standardization of logging capabilities, dedicated logging operations were developed in two standard services, the remote file access (FDAS), and the repository access service (RDAS). These logging operations allow clients to save one or several event logs in the server, and require only client authentication, requesting logging permissions.

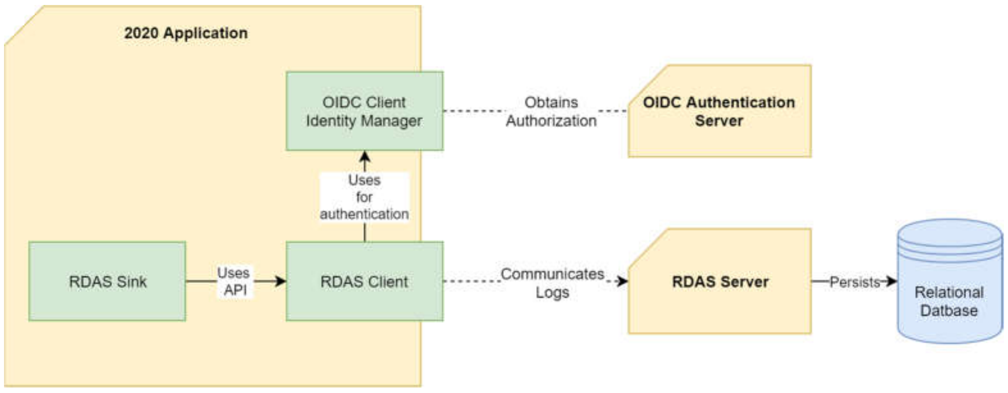


Figure 4.17: Communication between Sink and log destination in case of database persistence.

Sample execution

In this section, we present a code sample that performs initialization and logging using the CnaoLog libraries. Figure 4.18 contains the code for a small sample application, which configures logging through the interface defined in the CnaoLog library, up until the end of its execution.

This sample application configures several clients for accessing standard services, such as the RDAS server, FDAS server, and identity provider. The initialization methods clients related to these services use information contained in the application's configuration file. This configuration method is further detailed in section 4.6. The standard service clients have to be configured first, because they are required as parameters in the initialization of the logging service. In this sample, logging is being configured for three distinct endpoints: local and remote files, as well as the repository database.

The configuration of the logging service uses the builder software pattern. By doing so, every method of the builder class specifies another configuration setting, and optional libraries are allowed to define extension methods to the builder class. In this way, the same builder class is used to initialize the logging service independently of whether optional libraries are being used.

The build method finalizes the building process and returns the log instance. This object can be used to log events with methods such as "Information" or "Error". These methods log the events into all the configured sinks. Finally, because sometimes it is not convenient to inject or supply the log object to all classes which need to log events, the CLog static class was designed to hold a reference to the logging object. All logging is performed via the CLog static class. Before the end of the application execution, the log is disposed using the DisposeLog method. Disposing the log object before finalizing is recommended in order to ensure that all logs are persisted before the applications terminates.

```
class Program
{
    static void Main(string[] args)
    {
        IConfigLoader configLoader = new
ConfigLoaderBuilder().UseCnaoSettings().Build();
        IResourceOwnerIdentityManager authManager = new
CnaoIdentityManager();
        authManager.InitializeFromConfigAsync(configLoader).Wait();
        authManager.AuthenticateClientFromConfigAsync(configLoader).Wait();

        RdasClient rdasClient = RdasClient.GetConnectionAsync(configLoader,
authManager).Result;
        FdasClient fdasRestClient =
FdasClient.GetInstanceFromConfig(configLoader, authManager);

        //Necessary for all
        CLog.Logger = new CnaoSeriogBuilder()
             //Set minimum log level
             .SetMinimumLevelFromConfig(configLoader)
              //Add RDAS sink, optinal
              .EnableSerilogRdasLog(rdasClient, CLogLevel.Information)
              //Add FDAS sink, optional
              .EnableSerilogFdasLog(fdasRestClient, CLogLevel.Information)
              .EnableFileLog("./", 10000000)
              .Build();

        //Todo applifetime
        //...

        CLog.Information("Goodbye, world!");
        CLog.DisposeLog();
    }
}
```

Figure 4.18: CnaoLog client initialization and usage.

4.6 Configuration file management

We have estimated that there are currently around 60 applications on the legacy configuration and support environment. Due to the high number of applications, and the fact that some are only used sporadically, it is difficult to keep track of each application's configurations. Originally, the legacy environment was designed so that that all applications would have a standard XML configuration file with all application settings. This decision led to easier application maintenance and configuration in the past. As a result, it was decided that applications of the new environment would also follow this requirement.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <Logging>
    <Enabled value="false"/>
    <Fdas>
      <Enabled value="true"/>
      <BatchMaximumSize value="99"/>
      <BatchPeriodSeconds value="14"/>
      <MinimumLogLevel value="Warning"/>
    </Fdas>
  </Logging>
</configuration>
```

Figure 4.19: Sample app.config configuration file format. Defining the application logging to the FDAS service.

As mentioned previously, each application of the upgraded environment should contain a single XML configuration file for storing applications settings. This file should be named “app.config”, or “ApplicationName.config”. Configuration settings in this file are defined by an XML node, and its value attribute. A sample configuration file format can be seen in Figure 4.19.

In the legacy environment, it was the responsibility of the developer to define the application code to perform the following operations:

1. For each expected configuration setting, look for its name. If present obtain its value.
2. If the value is expected to be any format other than a string, convert it to its desired format. If the conversion fails, handle the exception properly.
3. Implement code for acquiring certain configuration settings only if other configuration settings are present. For example, if a setting enabling logging is true, then there is need to find the value of the setting containing the desired log file location.

During the development of the project, we have noticed that the application logic necessary to load configuration files of all libraries and applications developed up to that point did not diverge from the algorithm presented previously. Moreover, we found that developing the configuration algorithm was time consuming and error prone. Additionally, it was noted that the implementation of the step number 3 tended to result in a dependency graph, where some configuration settings were only required in specific scenarios. In order to aid the maintenance of these applications, these dependencies should be well documented, highlighting which settings with suitable default values could be left empty. It was found that more often than not, these details were not entirely documented, and thus part of the nuance in the configuration would be lost.

Therefore, a configuration library was designed with three objectives in mind:

- Standardizing the loading of configuration settings following the requirements defined in the product line architecture.
- Providing a faster way for developers to load and read settings from the configuration file.
- Providing a solution that allows a single point where all logic behind the configuration setting validation logic could be contained.

The developed solution for these requirements was integrated into the CnaoApplicationServices library. This library was the designated location for generic methods used in several configuration and support environment applications, and previously included a simple configuration setting reading method.

4.6.1 Definition of configuration models

In order to allow developers to specify a set of typed configuration settings to be read, the concept of configuration model classes was defined. Configuration models are defined as classes with the following set of properties:

- Configuration model classes implement the empty IConfigModel, or IReusableModel interface.
- Each annotated class property defines a configuration setting value to be read from the configuration file.
- The type of the property defines the desired type of the setting to be read.
- Path annotations define the XML path which should contain each configuration setting.
- A loaded configuration model instance is valid only if all required properties are successfully loaded and parsed.
- Annotations defined by the library can be used to define when each configuration setting is required.
- Configuration model composition is allowed. A model which is composed of several sub-models is invalid if any of its sub-models is invalid.

Figure 4.20 contains a sample configuration model class. As shown in the figure, every configuration setting has to define the XML address to locate the configuration setting. As mentioned previously, the configuration settings are declared as properties. Attributes should be defined in these properties to specify the location of each setting, and rules to validate the loaded settings.

Additionally, composition of configuration models can be performed by object composition, as shown in Figure 4.20. In this example, GeneralLogConfigurationModel contains another configuration model. As part of its own validation, if the FdasConf is required, the FdasLogConfiguration will be loaded and validated as well.

```
internal class GeneralLogConfiguration : IConfigModel
{
        public GeneralLogConfiguration()
        {
        }

        [Required]
        [Location("Logging:Enabled")]
        public bool? Enabled { get; set; }

        [RequiredIf(nameof(Enabled), true)]
        [ValidateObject]
        public FdasLogConfiguration FdasConf { get; set; }
}

internal class FdasLogConfiguration : IConfigModel
{
      …
}
```

Figure 4.20: Sample configuration model illustrating the composition of models.

In classes implementing the IConfigModel interface, the path annotations have to specify an absolute address to find each XML configuration setting. Alternatively, classes implementing the IReusableModel can define relative addresses, which will be appended to the absolute address defined by any configuration model classes that uses them.

4.6.2 Configuration loading

After designing a configuration model, developers can then load the configuration model during application execution. In order to perform this task, the ConfigLoader class has been developed. This class is able to load configuration models by accessing the configuration file, and then validate the resulting configuration model instances.

The ConfigLoader class should be instantiated using the builder software design pattern. The builder class, named ConfigLoaderBuilder, is responsible for providing methods to allow developers to configure and instantiate the ConfigLoader. While several configuration options have been defined, future developers of the configuration and support environment are encouraged to use the UseCnaoSettings method, which provides a default configuration that follows the recommendations defined in the product line architecture.

Figure 4.21 contains a short snippet that illustrates the process of instantiating the ConfigLoader, and then using it to load configuration settings into a configuration model instance. In this sample, the ConfigLoader is configured with the instantiated and CNAO settings. Afterwards the LoadConfiguration method of ConfigLoader instance is called, with the desired configuration model class to load as an argument. This configuration model is the one presented in Figure 4.20. The result of this method is an instance of the ConfigLoadingResult class. This class contains the obtained configuration model as well as the results of its validation. After the configuration model is loaded, developers should consult the result object in order to check if the validation was successful, and if necessary relay any error messages to the user. If the model was loaded and validated successfully it can then be provided to other classes that require the configuration settings.

```
//Instantiate the configuration loader, using builder pattern
ConfigLoader configLoader = new ConfigLoaderBuilder()
             .UseCnaoSettings().Build();

//Obtain configuration model
ConfigLoadingResult<GeneralLogConfiguration> configResult =
            configLoader.LoadConfiguration<GeneralLogConfiguration>();

//verify if the validation was successful
if (!configResult.IsValid)
{
    //if not, then log occurrence, or present issues to user
}

//Configuration model is contained inside the result
GeneralLogConfiguration logConfiguration = configResult.Config;
```

Figure 4.21: Sample snippet for instantiating a ConfigLoader instance, using the default configurations, and then obtaining all necessary settings from the configuration file into an instance of the GeneralLogConfiguration class.

4.7 Integration with LabVIEW environment

In the last year of the project, preliminary applications using the environment’s services started being developed. During this period, willingness to utilize some services developed for the configuration and s support environment arose with developers responsible for other environments. Not long after, initial suggestions were made to the technological project managers. As a result, a project was approved by the facility’s management to allow LabVIEW applications of the first level of the control system to consume the web services of the configuration and support environment. This project foresees the integration of several LabVIEW applications with the repository access service.

In this section, we describe the collaboration project for integrating applications from other environments to the relational data access service.

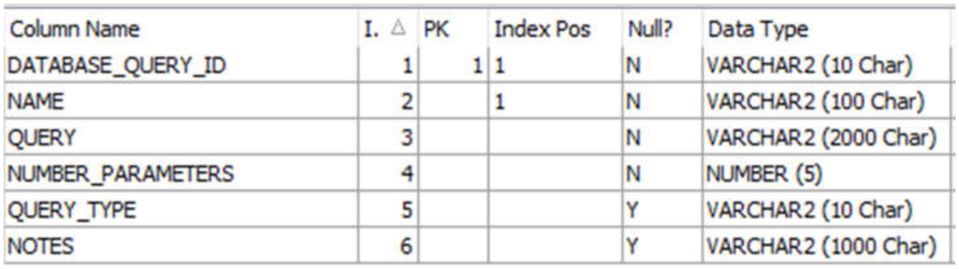
4.7.1 Relational data access service integration

The CNAO control system is composed of several software applications. In the past, some LabVIEW applications in the first level required access to the repository data for a small number of tasks. In order to do so, a server with repository access was developed. This server would communicate via a TCP and allow these applications to access some repository operations.

Due to the precarious nature of the previously mentioned server, as part of this project, it was decided to allow certain LabVIEW applications to communicate with the relational data access service to obtain repository data. Because the repository access service communicates via RESTful API, a RESTful API client LabVIEW component was designed by the LabVIEW development team to communicate this service, named by Query.lvlib.

Because the LabVIEW development team does not handle relational data often, their members are generally unfamiliar with the SQL language. In the past, LabVIEW developers would request a repository developer to define the SQL query, and then the query would be coded into the LabVIEW application, and sent to the custom-made TCP repository server. In order to streamline the process, while still not requiring LabVIEW developers to be proficient with SQL, a database for storing predefined queries to be used by these LabVIEW applications was inserted into the repository. The inserted table and its columns are presented in Figure 4.22.

It was decided that in order for LabVIEW developers to allow their applications to request a new relational query, they would need to contact the configuration and support environment developers and request the development and addition of a new SQL query to the DATABASE_QUERY table. They will have to inform a developer of the configuration and support environment team which repository information their application requires. Afterwards, they will have to provide a document with a future name for the query, a description of its uses, and which parts of the query should be defined by parameters. The configuration and support developer will then design the required query, and insert it in the repository.



Column Name	I. △	PK	Index Pos	Null?	Data Type
DATABASE_QUERY_ID	1	1	1	N	VARCHAR2 (10 Char)
NAME	2		1	N	VARCHAR2 (100 Char)
QUERY	3			N	VARCHAR2 (2000 Char)
NUMBER_PARAMETERS	4			N	NUMBER (5)
QUERY_TYPE	5			Y	VARCHAR2 (10 Char)
NOTES	6			Y	VARCHAR2 (1000 Char)

Figure 4.22: Repository table containing database queries for LabVIEW developers.

At run time, the LabVIEW application first requests the appropriate query from the RDAS server by its name. The application should then replace all escape sequences with the parameter values, and then use the library to request the execution of the query, passing along the appropriate access token. Figure 4.23 shows part of the Query.lvlib library, used to communicate with the RDAS server, using the HTTP protocol.

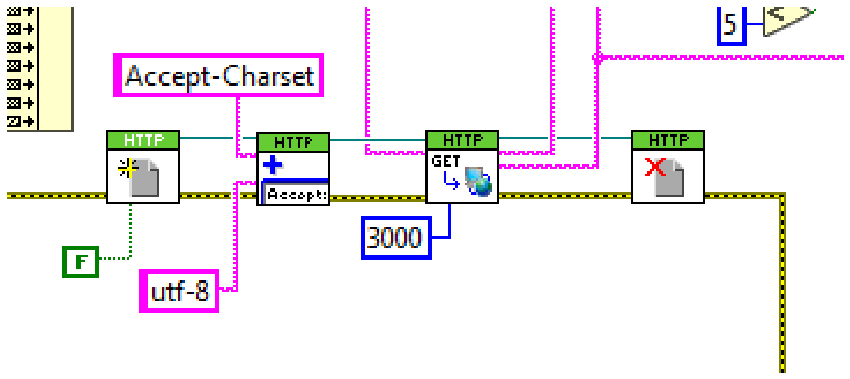


Figure 4.23: Illustration of LabVIEW code of the Query.lvlib [96].

As of the time of writing, the updated LabVIEW applications are in development phase. They do not yet communicate with the configuration and support environment's identity provider, and thus cannot obtain access tokens. Because of this, the RDAS server applications are communicating with uses development configurations, accessing the development repository, and not requiring access tokens for every operation [96]. Once the LabVIEW development team updates the Query.lvliv library to perform the role of an OpenID Connect client, these LabVIEW applications will be able to perform request for client access tokens.

Client credentials for the LabVIEW applications have already been added to the development repository tables that store the environment's identity provider configurations. Once the LabVIEW applications are finalized, they will be able to request access tokens with their client credentials, allowing them to access the repository data necessary to operate.

Chapter 5

Implementing the basis of the product line architecture

5.1 Introduction

Chapter 3 first introduced the requirement analysis for the upgraded configuration and support environment. Later, the designed product line architecture was presented, which aims at guiding the development of the next generation of control system applications in this environment.

In order to develop the applications for the upgraded configuration and support environment, software developers will have to follow the decisions and tactics outlined in the product-line architecture. As a result of this work, these applications do not have to be developed from the ground up, since the reusable software components presented in the Chapter 4 can be used. These components implement common application tasks, but are in general agnostic to the architecture of any application using them. In order to aid the implementation of the architectural design outlined in the product-line architecture a series of framework libraries were developed.

These frameworks contain code that provides a base implementation of the architectural patterns chosen for this environment. In addition, wizard generators were developed to produce customized skeleton applications that implement the product architecture. Each wizard is responsible for building one family of applications and uses one of the frameworks developed.

In section 5.2, we detail the decision process that led to the design of the frameworks and wizards. Later, in section 5.3, we present the design of the frameworks, and the classes shared between them. Afterwards, in section 5.4, we present the development and usage of the wizard generators.

This section had a different distribution of work than the rest of the thesis project. The design of the product-line architecture, standard services, and certification analysis was spearheaded by the PhD candidate. For these parts of the project, the PhD candidate would seek advice from his supervisor over technical matters, architectural concerns, and legacy compatibility. Periodically, the PhD candidate would submit the progress made, and incorporate feedback from his supervisor at CNAO, Eng. Luigi Casalegno.

For the implementation of the wizard and framework, the fellow was tasked in the beginning of the project with developing sample applications using several technologies and libraries. These applications were developed with the objective of evaluating the development environments and commercial libraries.

Afterwards, the fellow's supervisor, Eng. Luigi Casalegno, developed the initial versions of the framework libraries and wizard, including preliminary documentation, using the deliverables of the previous phase. It was during this phase that the great majority of 'out-of-the-box' configurable visual components were developed.

The PhD candidate was then tasked with finalizing certain parts of the wizard and framework. This phase consisted of integrating the developed standard services into the framework elements, developing the initialization process of these services.

Finally, both the PhD candidate and his supervisor worked in the last phase of the development of the frameworks and wizard, which consisted of iterating over the design, developing additional features deemed necessary.

5.2 From architecture to implementation

The product line architecture described several reusable core components designed to perform operations required by applications in this environment. The design and development of these components was presented in detail in chapter 4. Some of these components take form of services, following a client-server communication approach, while others were designed as libraries.

Several of the design choices specified in the product line architecture cannot be translated into reusable components. For example, the design tactic of seeking a clear separation of application layers is a convention that has to be maintained by developers. Reusable components packaged libraries can follow this design by only requesting services from other layers through their interface. However, this design will only persist if final developer also maintains this separation in his code. Accordingly, design choices are translated from the architecture to the application by promoting or enforcing the use of a set of software patterns.

Another way of translating the architecture's decisions into the environment applications is through the usage of software frameworks. Software frameworks are similar to libraries in the sense that both are software code, intended to be re-used in the applications. However, software frameworks are difficult to define. A common definition of a software framework, as cited by Johnson [97], is that a framework is a "reusable design of all or part of a system that is represented by a set of abstract classes, and the way their instances interact" [97]. An interesting aspect of software frameworks is that they allow the preservation of a set of defined interfaces over several applications by providing extension points to applications, where these will implement their variations [98]. A framework often implements several design decisions and demands that applications adapt to them [97].

In this regard, frameworks constrain the architecture of final applications, as they are often only providing extensions to a previously developed design. In return they serve several purposes, such as providing customizable implementations to software patterns, and allowing for software reuse. In our project, we looked into the constraints that frameworks put in the application's architecture as a benefit rather than a limitation. This is because the configuration and support environment has a set of operations and usage scenarios defined as its scope, and in this project there is no interest in developing applications outside of this scope. By designing frameworks which implement a base version of several architectural patterns required by the product line architecture, such as dependency injection and MVVM, developers are constrained to use these patterns in the applications.

The scope of a product line architecture is defined by Bass et al. [56] as a statement of which systems should be built using a given product line. When defining the scope of the configuration and support environment, two distinct application families are obtained. The first family is composed of the configuration and support applications that manage the data in the control system repository. These applications are tasked with creating or editing configuration data and validating the information already present. The second family was composed of specialized control applications to monitor and interact with the control system.

In order to implement design choices in the product line architecture, and further reusable code, a framework was developed for each application family. These frameworks implement several design decisions that form the core of the applications, such as initialization of service clients, usage of dependency injection, organization into software tiers, and user interface navigation.

5.3 Framework2020 design and development

Figure 5.1, initially presented in Chapter 3, displays the software elements surrounded by the red line, which are defined by the application's framework. The framework's role in the application is the implementation of several base software elements to be extended by the application, as well as performing the initialization of the standard services.

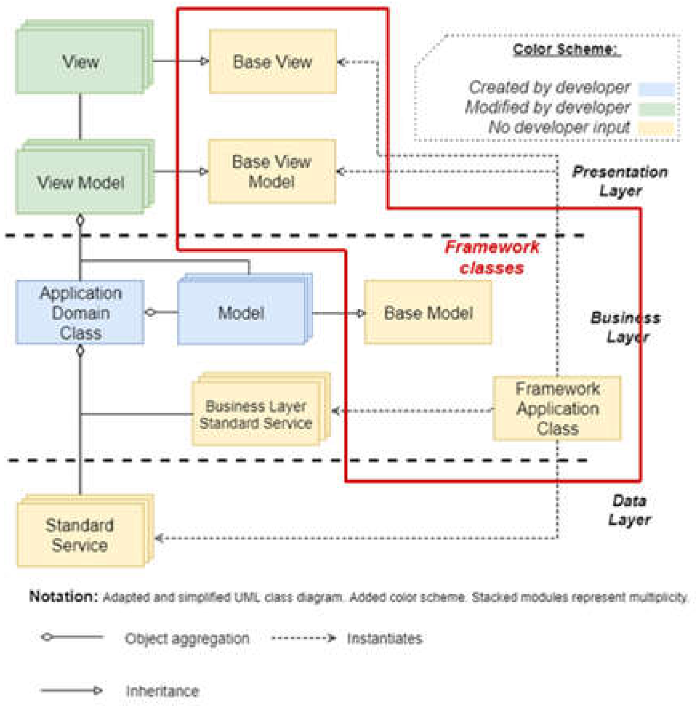


Figure 5.1: Class diagram of a front-end configuration and support application, following the proposed product line architecture.

Currently, there are two framework libraries, with the AppBase2020 framework being the framework supporting data management applications. This framework depends on two libraries, the Dialogs2020 library, and the Framework2020Core library. As noted previously, each of the application families is supported by one framework. However, these frameworks have much more similarities than differences. Currently, the control applications framework, named AppBase2020Control, extends the data management framework.

Figure 5.2 illustrates the framework and supporting libraries dependency graph.

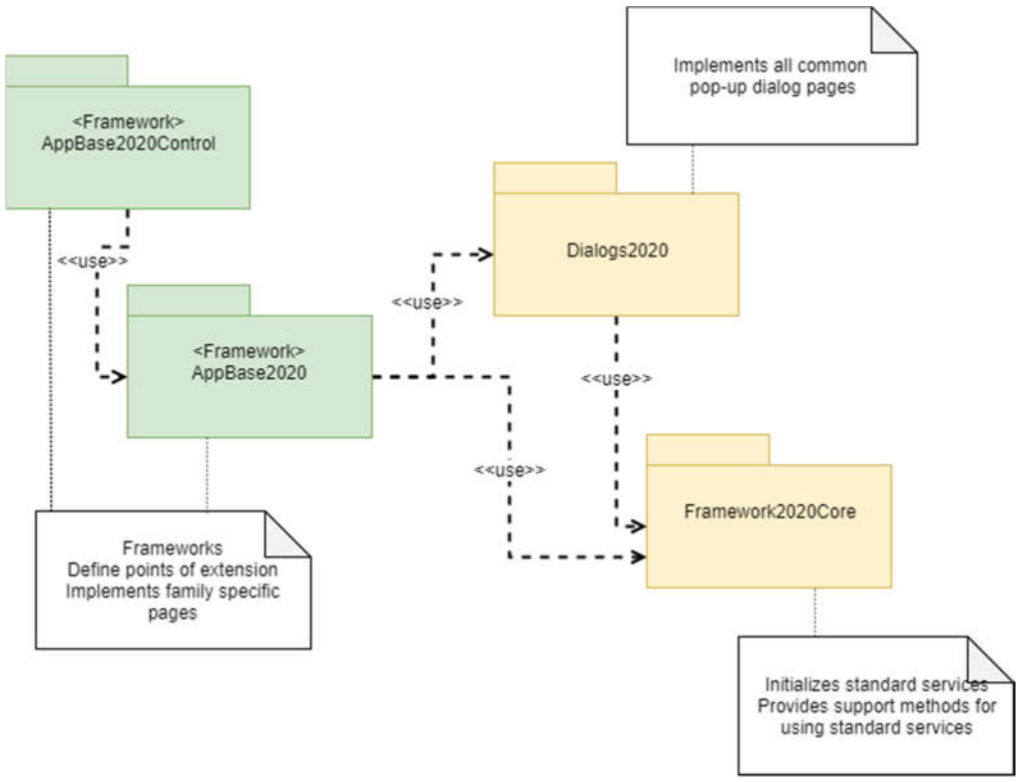

Implements all common pop-up dialog pages
<Framework> AppBase2020Control
<<use>>
Dialogs2020
<<use>>
<Framework> AppBase2020
<<use>>
<<use>>
Framework2020Core
Frameworks Define points of extension Implements family specific pages
Initializes standard services Provides support methods for using standard services

Figure 5.2: Frameworks and their dependencies. As can be seen, the AppBase2020Control framework is an extension of the AppBase2020.

The two frameworks share most of their code, and both also follow the product-line architecture, however they diverge in the initialization process, and provide different sets of configurable visual components. The data management application framework contains a large amount of configurable grid data visualization and editing forms. Meanwhile, the control application framework extends the data management with graph monitoring panels and OPC-UA interaction forms.

The base library of the framework is the Framework2020Core, which all others depend on, directly or indirectly. This library performs services for the framework library, such as initializing services, and defining constant values as well as reusable support functions. Additionally, the Framework2020Core defines several interfaces that have to be implemented by the frameworks.

The Dialogs2020 library defines several presentation layer components that are used in pre-defined framework pages. These pop-up pagers are often used in several framework pages and perform tasks such as provide a message to the user, or allow users to pick specify a date, a file to upload, or where to save a new file. These dialog pages do not have complex underlying logic, instead they were designed to direct the user to provide several types of input values.

The AppBase2020 framework is designed to implement several of the software elements necessary for implementing the product line architecture, such as the base classes for several MVVM classes, as well as the predefined configurable pages. Additionally, AppBase2020 is responsible for implementing the FrameworkApplication class, which is the entry point of the framework, and initializes the services and framework classes. Finally, AppBase2020 also defines the usage of dependency injection, and provides extension points for the developer to add their dependencies and resolutions.

While the AppBase2020 framework is to be used by repository management applications, the AppBase2020Control framework is used by control applications. The control applications wizard uses the extensions defined in this framework library when generating skeleton control applications. Control applications have access to several configurable control pages, which are used to monitor equipment values present in the third level of the control system and display them graphically.

5.4 Application wizards

The Framework2020 wizards are Visual Studio extensions that allow the creation of customized skeleton applications that follow the architecture defined for applications of the configuration and support environment.

The wizards generate applications using the AppBase2020 libraries. Additionally, the wizards allow developers to customize skeleton applications in the following manners:

- Define the navigation method of the application.
- Enable or disable standard functionality forms. Forms such as those for user authentication, and application configuration can be enabled or disabled using the wizard.
- Configure the presence and usage of standard services.
- Insert and arrange pre-defined configurable pages.

5.4.1 Generating skeleton applications with a wizard

In this section, we present the development process of a configuration and support environment application using the Framework2020 wizards. In this section, a brief overview of the installation requirements for the wizards is mentioned. Then, a more detailed walkthrough of the definition and configuration of a data management application is presented. Later, we show the AddTopic component of the wizard, which allows developers to add additional pages to an application previously created with the wizard. Finally, since we only show the development process of a data management application, we will discuss the similarities and differences to the usage of the configuration application wizard.

In order to develop a new environment application using the wizards, developers need to previously have installed the Visual Studio extension that contains the wizard application templates. Figure 5.3 illustrates the project creation screen of the Visual Studio 2017 IDE, showing the Data

Management template. When the user selects the creation of a C# application project with using one of the wizard templates, the corresponding wizard forms are shown.

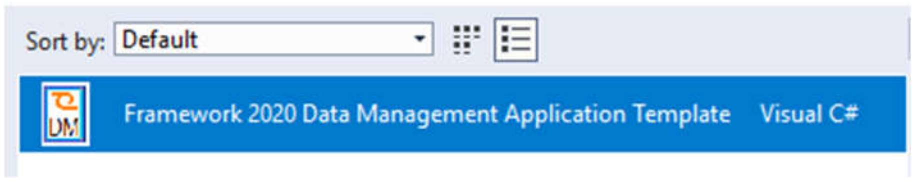
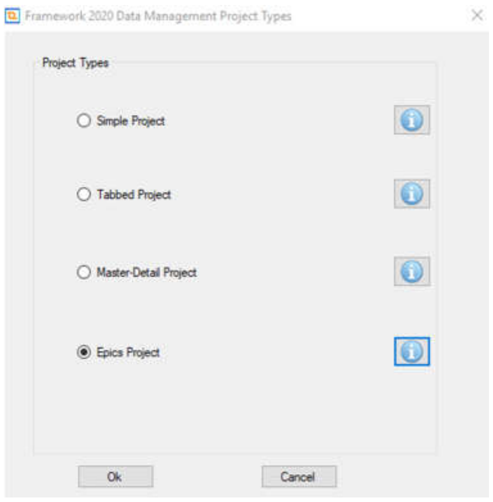


Figure 5.3: The project creation screen of Visual Studio 2017, displaying the Data Management application template. Once this template is used, the Data Management wizard is invoked.

After selecting the project template, the developer must also select a project name and destination folder. Once this is done, the first wizard form opens, allowing the developer to choose the navigation method of the final applications. The currently defined options for navigation are the following:

- Simple project. A one-page project without navigation capabilities.
- Tabbed project. A project with multiple pages, which are laid in a flat hierarchy. Each page can navigate from any other. The name of each tabbed page is displayed at the top side of the screen. Clicking on one of the tabs opens the page.
- Epics project. The Epics project contains the most complex navigation system. The navigation of Epics project contains several page groups on the left side of the screen. Each page group can contain several pages. Once users click the page at the left side bar, the page opens as a tabbed page. Multiple tabbed can be open at once on the top of the screen, but only one will be displayed at each time.

Figure 5.4 presents the wizards navigation selection form. This is the first of many wizard forms, and it allows developers to select the navigation of the project. While the MasterDetail project type is currently presented as an option in the figure, it has been decided that the MasterDetail navigation type will be removed in the later version of the wizard generator.



Framework 2020 Data Management Project Types

Project Types

Simple Project

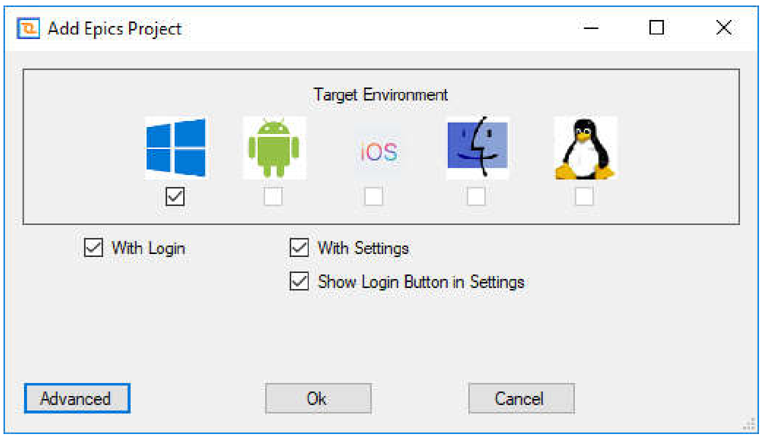
Tabbed Project

Master-Detail Project

[✓ Epics Project]

Figure 5.4: Navigation selection form of the data management application wizard, presenting the possible navigation options for applications. Currently the third option, Master-Detail, is scheduled to be removed in the next version.

Afterwards, developers will be presented with the general configuration form. The general configuration form presents to the developer several optional features that may be included in the application, such as settings and log-in pages, as well as choosing the targeted platforms. This configuration form also provides access to the wizard's advanced configuration form. Figure 5.5 contains a screen capture displaying the wizard's general configuration form. If the developer does not wish to change any advanced configuration, pressing the *Ok* button ends the generation and finalizes the skeleton application.



Add Epics Project

Target Environment

☑ With Login

☑ With Settings

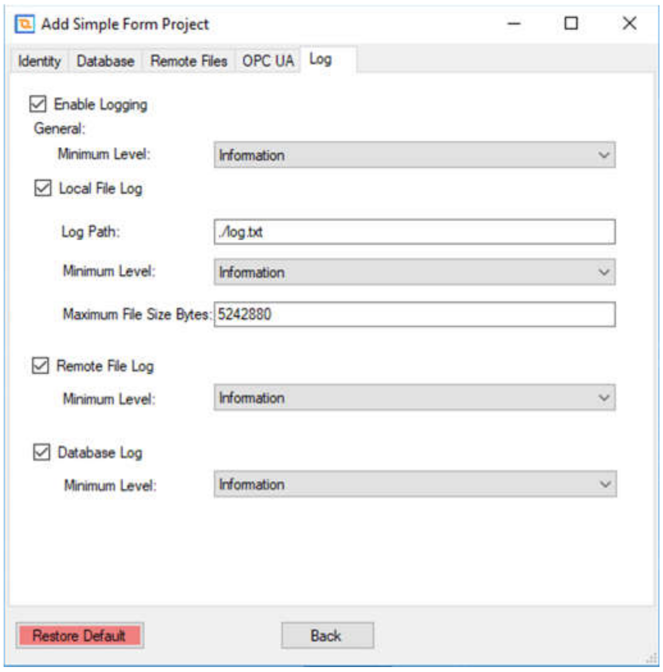
☑ Show Login Button in Settings

Figure 5.5: Data management wizard's general configuration form.

The advanced configuration form is opened if the developer choses the Advanced button in the general configuration form. This form is designed allow the developers to configure the usage of standard services and other support libraries. Figure 5.6 contains a screen capture of the advanced configuration form. This form allows developers to select which standard services or support libraries will be used, as well as provide their configurations. In the screen capture, a configuration tab is present for each of the client libraries presented in Chapter 4.

While in theory, a configuration and support environment application may connect to several FDAS services, generally these applications only connect to a single one. Therefore, the remote file configuration tab only allows developers to specify the location of one service. Currently, the OPC-UA configuration tab is still not finalized, and thus, OPC-UA connections to third level services must be defined by developers after the skeleton application is generated. It is expected that the next version of the generator will support configuring a single OPC-UA connection.

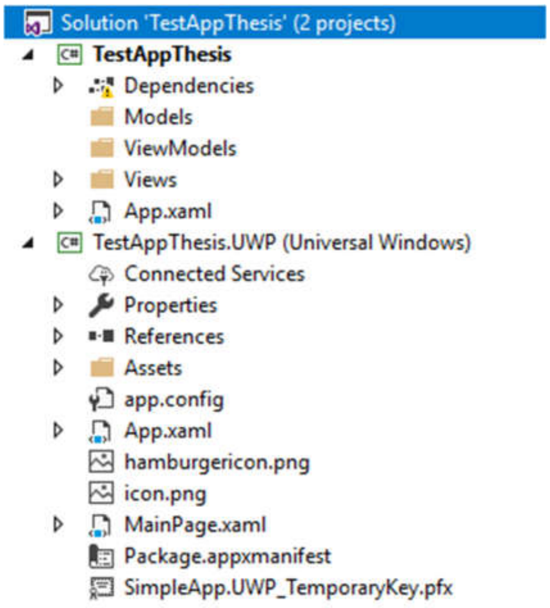
Several configuration settings of the advanced configuration form are interlinked. For example, the remote file log option presented in Figure 5.6 would be grayed out and become unable to be selected if FDAS access was disabled in the *Remote Files* tab.



☑ Enable Logging
General:
Minimum Level: Information
☑ Local File Log
Log Path: ./log.txt
Minimum Level: Information
Maximum File Size Bytes: 5242880
☑ Remote File Log
Minimum Level: Information
☑ Database Log
Minimum Level: Information

Figure 5.6: Data management wizard's advanced configuration form.

Once the developer finishes configuring the skeleton application, he or se should finish the generation process by pressing the *Ok* button in the general configuration form. Once the generator has finalized, the developer will be presented the generated skeleton application. At this point, the generated skeleton application will contain a functional login page, if such page was selected. Skeleton applications without a login form will directly present an empty main page instead. Figure 5.7 contains a screen capture of the solution files generated by the wizard. As shown in this figure, two projects were created, a multi-platform library, named *TestAppThesis,* which is imported by the *TestAppThesis.UWP* windows application. Skeleton applications with navigation supporting multiple pages will display no additional pages at this time.



```
Solution 'TestAppThesis' (2 projects)
  TestAppThesis
    Dependencies
    Models
    ViewModels
    Views
    App.xaml
  TestAppThesis.UWP (Universal Windows)
    Connected Services
    Properties
    References
    Assets
    app.config
    App.xaml
    hamburgericon.png
    icon.png
    MainPage.xaml
    Package.appxmanifest
    SimpleApp.UWP_TemporaryKey.pfx
```

Figure 5.7: Finalized skeleton application that was generated with the data management application wizard.

Regarding the creation of the skeleton application, the data management application wizard and control application wizard are almost identical. The main difference until this point is that data management applications will use the AppBase2020 framework, while control applications will use the AppBaseControl2020 framework.

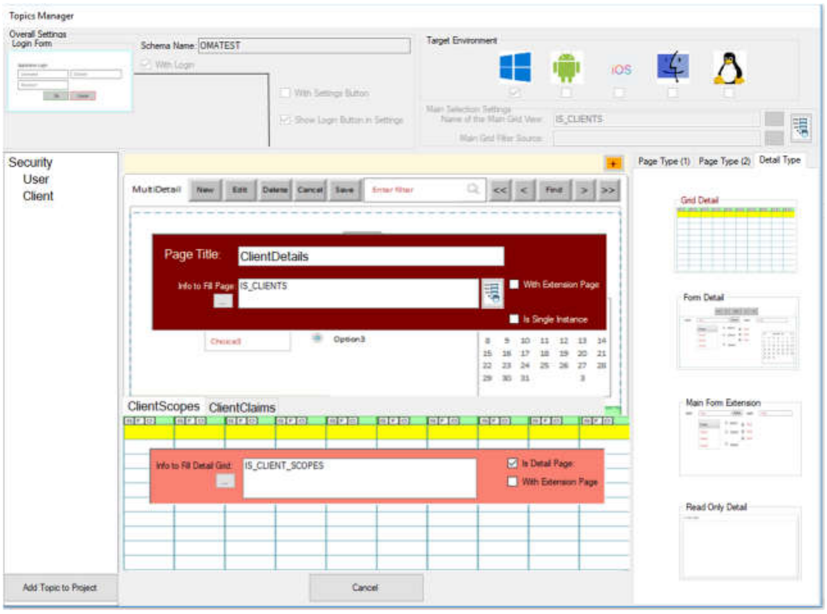
5.4.2 Adding pages into a skeleton application

After a skeleton application is created, it contains only the main page (besides the login and configuration optimal pages). Following the guidelines defined in the product line architecture, the developer must only create new pages through the AddTopic template.

When adding new files into projects, the IDE allows developers to select many file templates, such as code files (classes, interfaces, enumeration files), or other files (XML, JSON, etc). The wizard extention adds a new file template type named "AddTopic". The "AddTopic" template allows developers to select a page type from a set of pre-defined pages, configure it, and add it to the project.

All pre-defined pages except for the blank page are configurable. The blank page creates an empty page, without any logic associated to it, so that developers can define the visual elements, as well as domain layer code for the page. For all other pre-defined pages, developers are not expected to change their code once they are generated by the "AddTopic" wizard component. The configurable pre-defined pages are generated with the supporting code necessary for their operation. For example, configurable pages that manage repository data will use the RDAS library classes, which are initialized by the AppBase2020 framework. Because the configurable page's classes are defined in the framework, if the configurable page is required to obtain data, filter it, and manage insertions and deletions, all code will be added to the project, without any other developer action.

For example, Figure 5.8 presents the configuration of the client application management form. In the figure, the table containing client information is being configured to appear in the top part of the form, while several detail grids are being set to allow for the management of the scopes, secrets, and claims accessible to clients.



Schema Name: OMATEST

Name of the Main Grid View: IS_CLIENTS

Page Title: ClientDetails

Info to Fill Page: IS_CLIENTS

☐ With Extension Page

☐ Is Single Instance

Info to Fill Detail Grid: IS_CLIENT_SCOPES

☑ Is Detail Page

☐ With Extension Page

Figure 5.8: Generation of the skeleton application page using the AddTopic Template.

Figure 5.8 contains a screen capture of the "AddTopic" template when applied to an application using the Epic template navigation. The developer must define the page sets, and then add one page to a page set. This process defines how this page will be navigated to. Afterwards, developers choose the template of the page. Each page template has its own configuration parameters. Once all configuration parameters are set, the developer selects the "Add Template" option, and the new page, and all necessary support classes will be added to the project.

Figure 5.9 contains the project after the addition of a data grid page which contains three detail grids. The main page and each one of its detail sub-pages display data from a single table or view of the repository. As can be seen, files containing MVVM pattern classes for the multi-detail data grid page have been created (all files except PropertySettings.xaml and App.xaml have been added).

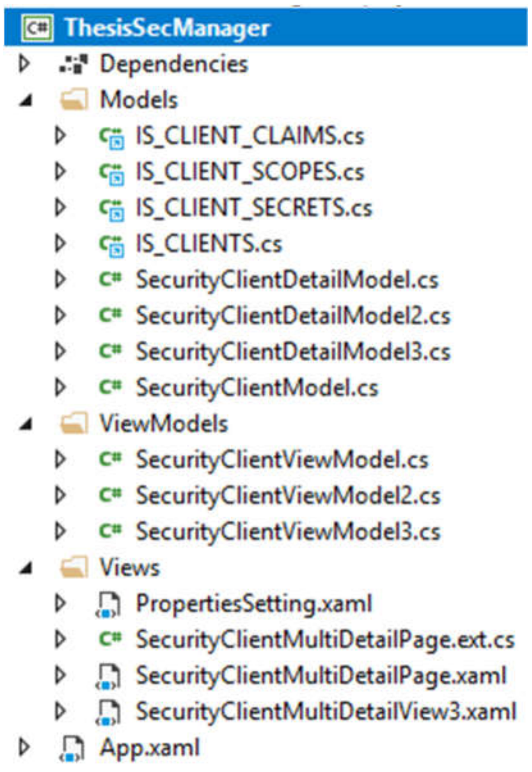

```
ThesisSecManager
  Dependencies
  Models
    IS_CLIENT_CLAIMS.cs
    IS_CLIENT_SCOPES.cs
    IS_CLIENT_SECRETS.cs
    IS_CLIENTS.cs
    SecurityClientDetailModel.cs
    SecurityClientDetailModel2.cs
    SecurityClientDetailModel3.cs
    SecurityClientModel.cs
  ViewModels
    SecurityClientViewModel.cs
    SecurityClientViewModel2.cs
    SecurityClientViewModel3.cs
  Views
    PropertiesSetting.xaml
    SecurityClientMultiDetailPage.ext.cs
    SecurityClientMultiDetailPage.xaml
    SecurityClientMultiDetailView3.xaml
  App.xaml
```

Figure 5.9: Project after the addition of a configurable multi-detail data grid.

The most noticeable difference between the data management application wizard and the control application wizard is the selection of pre-defined configurable pages available to each wizard. While the data management wizard only contains pre-defined pages related to accessing and managing the CNAO repository, the control wizard contains pre-defined pages for accessing remote files, and to graph data received from an OPC-UA server. Therefore, by selecting the appropriate wizard, adding and configuring all predefined pages necessary, and adding and modifying several blank pages, developers of the configuration and support environment are able to develop applications for the environment.

Chapter 6

Towards certification of control system applications

The CNAO facility performs hadrontherapy, hence it is governed by medical regulation. As the facility maintains a particle accelerator for the treatment of cancer, several safety standards have to be complied at various levels of the accelerator. In this work, however, we will approach only the safety standards related to medical software.

The control system software is classified as medical software, therefore, it has to comply with medical software standards. All software that has been developed with the intent of incorporation into a medical device is classified as medical device software [99][100]. As a consequence, the accelerator control system follows the regulatory standards for medical software. Under the regulatory environment for medical software in member states of European Union, certification is accredited on the basis of complying with the development process defined in several accepted standards. These standards are the IEC 62304+Amendment1[99], [100], ISO 14971[101], and ISO 13485[102].

The IEC 62304 standard, and its 2016 amendment, define a framework composed of processes to be performed during the lifecycle of the medical software, from development to product maintenance [99]. This standard has been adopted by the member states of the European Union, as well as by the American FDA agency. In these standards processes are composed of tasks that span from the software's development planning into the software release, and continuing through the maintenance stage. Figure 6.1 illustrates the activities required for medical software certification. This project focused on strategies for compliance with the lifecycle processes defined in the IEC 62304.

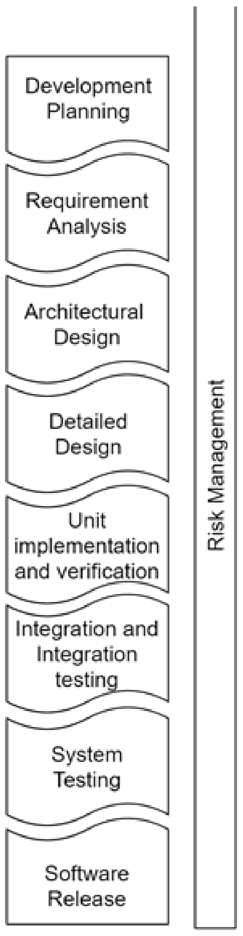


Figure 6.1: Applicable activities for complying with the IEC 62304/Amd2016 and IEC 14971 standards up until the software release. Information from [99].

Medical software is graded in a system as safety class A, B, or C, depending on whether it can cause harm to patients or operators. Under this system, class A software represents no risk of harm to patients or operators [99]. The software's classification determines the certification activities that have to be performed, with a large impact on the amount of verification and risk management activities necessary to be carried out during the certification process [103]. During the requirements analysis for the configuration and support environment, it was noted that currently existing applications of the configuration and support environment were certified under the safety class A. This was because applications in this environment do not directly interface with the accelerator, and that several software and hardware safety mechanisms exist to assure the safe treatment outside the environment.

Since certification is based around complying with the processes defined in the certification standards, applications that share software components are still required to perform the certification tasks independently. Rushby [104] examines the notion of modular certification in the aviation software domain, where software elements certified in isolation and then their integration would be certified using only the modules properties in isolation. Rushby notes that such modular certification is unfeasible currently, among other reasons, because of the way failures may propagate [104].

In the literature, several authors relate the experiences and lessons from the compliance with the software lifecycle processes. However, many of these works focus on the processes themselves, as well as the quality assurance procedures, such as [105][106]. Meanwhile, other works propose architecture strategies for improving the validation and certification efforts of critical software that reuses components [107], [108]. Among these, a noteworthy work by Land et al. [109] compiled and listed several currently used industry practices for enabling component reuse.

When developing the product-line architecture for configuration and support applications, we attempted to apply principles and strategies recommended by the certification standards, as well as previous academic works. Because all software applications of the legacy configuration and support environment has been classified as Class A, which is the lowest risk class, this work focuses on efficient compliance with the development lifecycle processes (specified by IEC 62304).

Section 6.1 presents several architectural choices present in the product-line architecture that were interned to facilitate the certification process. Afterwards, section 6.2 discusses each choice, highlighting their justifications, as well as the expected results.

6.1 Designing towards reusable certification

The aim of this phase of the project was to evaluate and steer the development of the new configuration and support environment tools towards aiding the future certification process. During the design of the upgraded environment, two tactics were implemented with the objective of helping the certification. Firstly, reusable software elements that are to be integrated into the future environment applications were defined as reusable components. Secondly, we have defined a product-line architecture containing several patterns aimed at emphasizing quality attributes such as testability.

Component based software engineering is a field of study that attempts to promote software reuse through applications which are developed with components. In component-based software engineering, a component is defined by Councill et al. [110] as a software element that can be deployed and used in applications without requiring modification, while also fulfilling certain standards of interaction and composition. During this project, we developed the reusable software elements, which are presented in chapter 4, while trying to emphasize several concepts of component-based development. Namely, by focusing on a defined interface, designed by the project, that each software element adheres to, and additional user documentation provided for each library, to inform application developers of any characteristic not present in the interface.

As part of this project, the following software operations were designed as reusable components.

- Communication with other control system devices, such as the repository database, filesystem access, and communication with third level control system equipment, via the OPC-UA protocol. The libraries performing these tasks have been described in section 4.1 to 4.3.
- User and client authentication and authorization using the OpenID Connect standard [78], as explained in section 4.4.
- Other operations such as system logging and configuration, presented in section 4.5 and 4.6.

The design decisions in the application architecture heavily impacts the quality attributes of the final application [56]. A quality attribute, as described by Bass et al. [56] as “a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders”, such as testability, and scalability.

In this project, we decided to focus on the testability quality attribute in the architecture. In order to achieve this, the product line architecture defines several patterns and design decisions to enforce the separation of application layers, and to encourage loose coupling of software elements. For example, the product line architecture describes each software layer and several software libraries that should be used in each layer.

The Presentation Layer follows the MVVM [111] software pattern. The MVVM pattern decouples presentation layer code from domain classes, and allows model and view model classes to be fully unit tested [111], thus aiding the verification process.

We have developed the AppBase2020 set of frameworks, which are described in detail in Section 5.3, to support the product line architecture. The frameworks provide base classes for the MVVM elements and perform initialization of the standard services. This is done to assist developer in complying with the reference architecture, as well as segregate the application domain classes from the services used by them.

In order to make the applications development easier and more reliable, developers are provided with an out-of-the-box base application, by a configurable wizard generator, which is explained in detail in Section 5.4.

6.2 Discussion

After presenting the decisions taken in order to assist the medical software certification of the future configuration and support environment applications, we discuss the expected impact of these decisions. Whenever possible, we attempt to trace arguments given to recommendations present in the literature, or in the standards themselves[3].

In this section, we first discuss the impact of the component based design choices in the certification process. Later, we categorize and argue the expected benefits of designing final applications according to a product line architecture.

The IEC 62304/Amd1 certification standard emphasizes that, when defining the software architecture, strategies for component segregation should be used to avoid unsafe interactions between them. Additionally, when credible arguments can be given to justify that the architecture separates components into separate software items in a correct manner, these software items may receive a different software safety class from the system [99]. The certification process has different levels of granularity depending on safety class, and some certification activities may even not be required for low safety levels [103]. By implementing commonly used software operations into reusable components, these can be implemented using principles of component based engineering as software units, and are often of the lowest safety class (class A). Thus, several certification tasks are not mandatory for these units, such as the task 5.4.2, entitled "Develop detailed design for each software unit" [99]. Additionally, we argue that certification artefacts produced from developing and certifying reusable components can be used as the basis for their counterparts in the final application, such as unit-test verification (activity 5.5), and establishment of a software maintenance plan (activity 6.1).

From our review of the certification standards, the separation of applications into several testable components has no effect on which system of system wide tasks are required. Because the scope of most certification tasks is the software system as a whole, the benefits of the loose coupling, component based approach are limited to a minority of certification tasks.

Regarding the usage of a product line architecture, we argue that its adoption and enforcement reduces effort required for certification. The importance of deep knowledge of the system's architecture in the certification process is highlighted several times, such as in the annex B of the IEC 62304 standard, where it is stated that, concerning software safety classification, risks associated with each software item can only be determined once the software architecture "defines the role of the software item in terms of its purpose and its interfaces with other software and hardware items" [99]. Furthermore, in the context of the activity 7.1, which is the analysis of software contributing to hazard situations process activity, the standard guidelines point out that hazardous situations can only be fully identified once the software architecture has been designed, and, only then, proposed risk control measures can be completely evaluated [99].

[3] This chapter is based on the conference proceedings published entitled "Certification of Component-Based Particle Therapy Software" [112], written by the authors as part of this work.

By defining the product line architecture and enforcing it during the design of applications, developers have access to a ‘used-and-tested’ architecture, accompanied by several internal documents explaining its usage, interaction between components and validation procedures. As developers design configuration and support environment applications using the enforced general architecture, the arguments provided to justify the safe interaction between developer modules and reusable components should be written using artefacts from the product line documentation and previous applications as basis [112].

Chapter 7

Results and evaluation

7.1 Introduction

The development of final applications to be used in CNAO's control system is not the focus of this work, however, the development of pilot applications has several benefits. Firstly, pilot applications can showcase the capabilities of the environment, and thus guide and motivate developers. Secondly, by developing pilot applications, we are able to test the tools and libraries developed, allowing us to improve them based on the knowledge obtained.

Section 7.2.1 presents the main pilot application developed during this work. This application, generated with the data management application wizard, allows repository administrators to manage repository data which is used to configure the environment's identity provider. Additionally, the early results of the LabVIEW integration project are described in section 7.2.2, alongside its pilot application, named EasyLoader.

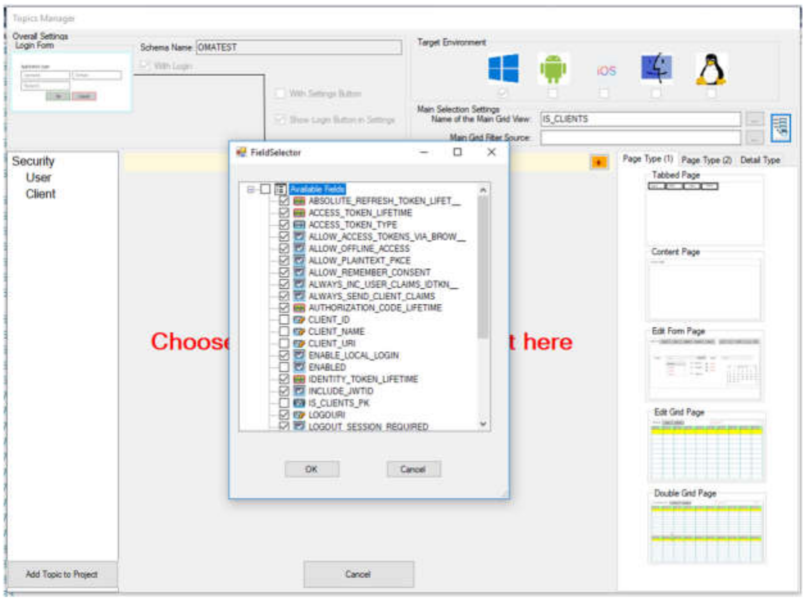
Later, in section 7.3, we evaluate the capabilities of the upgraded configuration and support environment in relation to the legacy environment. In this evaluation, we analyze the added features, and their impact in facility's control system. Afterwards, we evaluate each component individually, analyzing the effects of the design decisions.

7.2 Pilot applications

7.2.1 Security configuration application

A pilot configuration and support environment application was developed as part of this work. During this work, a large amount of applications was developed, such as prototype applications for testing architecture strategies presented in Section 3.3, and executable tools for aiding the development process, such as the RDAS class generator presented in Section 4.1.5. The pilot application differs from the previously mentioned application for two reasons: Firstly, the pilot application has a different audience, as it is to be used by control system operators. Secondly, the pilot application fully follows the product line architecture requirements, and therefore consumes the services developed, and is originally generated by the wizard generator.

In addition to serving its primary purpose in the hands of control system operators, the pilot application is intended to serve as an example application for future developers of the configuration and support environment. The pilot application, named “CNAO OIDConfig”, is a configuration application that allows control system administrators to define protected resources and set user authorization to the protected resources in the repository. This information is later loaded by the CNAO identity provider (see Section 4.4.3).


Schema Name: OMATEST

Name of the Main Grid View: IS_CLIENTS

Main Grid Filter Source:

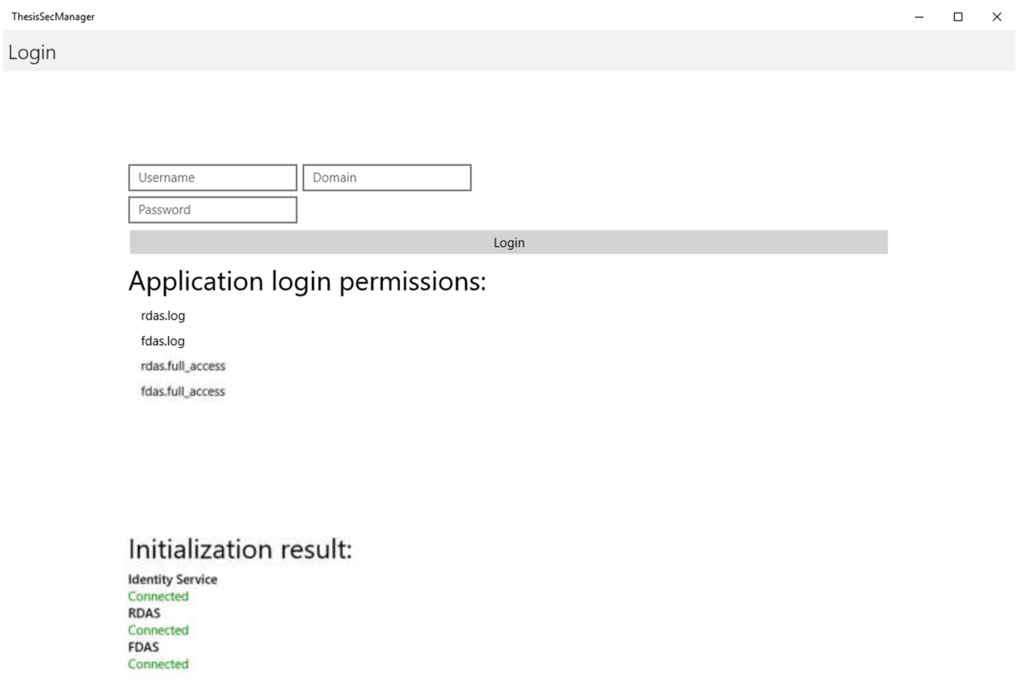
FieldSelector

Available Fields
- [✓] ABSOLUTE_REFRESH_TOKEN_LIFET__
- [✓] ACCESS_TOKEN_LIFETIME
- [✓] ACCESS_TOKEN_TYPE
- [✓] ALLOW_ACCESS_TOKENS_VIA_BROW__
- [✓] ALLOW_OFFLINE_ACCESS
- [✓] ALLOW_PLAINTEXT_PKCE
- [✓] ALLOW_REMEMBER_CONSENT
- [✓] ALWAYS_INC_USER_CLAIMS_IDTKN__
- [✓] ALWAYS_SEND_CLIENT_CLAIMS
- [✓] AUTHORIZATION_CODE_LIFETIME
- [ ] CLIENT_ID
- [ ] CLIENT_NAME
- [ ] CLIENT_URI
- [✓] ENABLE_LOCAL_LOGIN
- [ ] ENABLED
- [✓] IDENTITY_TOKEN_LIFETIME
- [✓] INCLUDE_JWTID
- [ ] IS_CLIENTS_PK
- [✓] LOGOURI
- [✓] LOGOUT_SESSION_REQUIRED

Figure 7.1: Generation of the skeleton application for CNAO OIDConfig displaying selection of the fields to be displayed in the selection form.

The CNAO OIDConfig application was generated with the Data Management application wizard (presented in Section 5.4). The Data Management wizard allows for the insertion of several configurable data management forms, which were extensively used. Figure 7.1 illustrates the design of the client management form of the application, where the client fields to be displayed to users were chosen. The usage of configurable multi-grid edit forms allows developers to manage data of several related database tables in the same form.

After the skeleton application was generated, several features were added to the generated code in order to finalize the application. Namely, single instance edit forms were added to the application in the previously blank sections, one of each primary entity. Additionally, since a test version of the generator was used, a few alterations to the generated application are still required for bug-fixing.



Application login permissions:

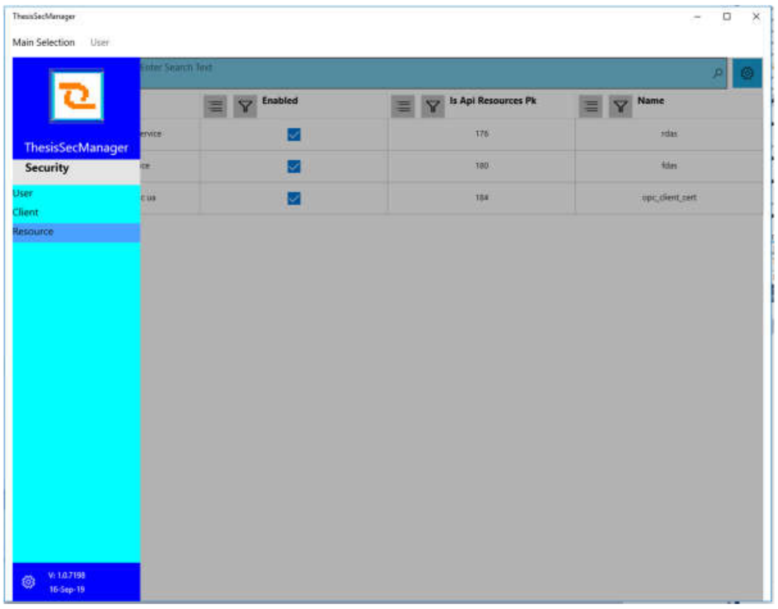
rdas.log
fdas.log
rdas.full_access
fdas.full_access

Initialization result:

Identity Service
Connected
RDAS
Connected
FDAS
Connected

Figure 7.2: CNAO OIDConfig login page.

In the final application, the initial page presented to users is the Login page, illustrated in Figure 7.2. Because the application requires access to the repository, only users with permission to manage repository data are able to proceed. In addition, the Login page displays the authorization scopes that the application accesses on behalf of the user, as well as the status of the services it depends on.

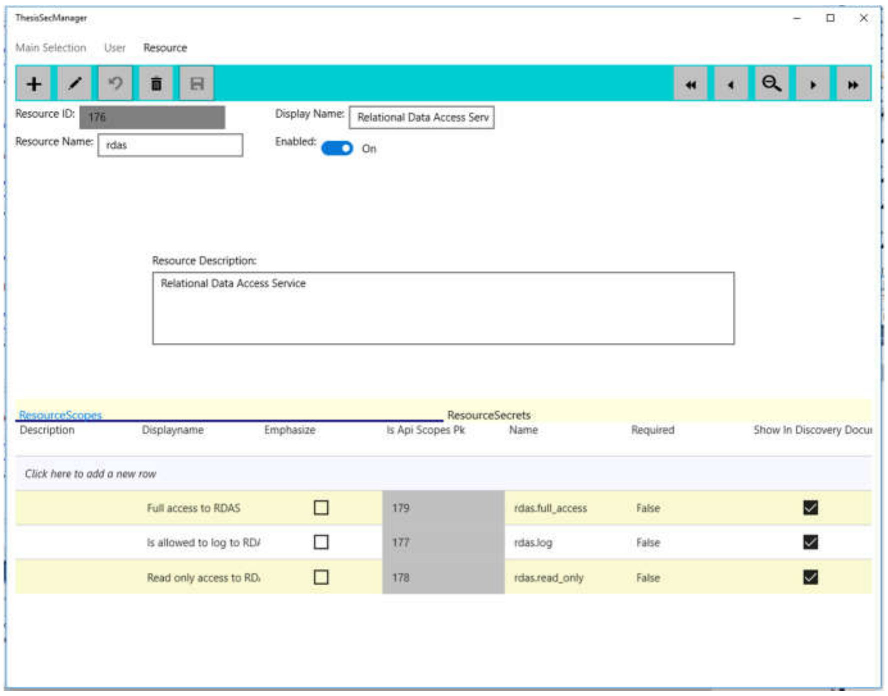


	Enabled	Is Api Resources Pk	Name
ervice	✓	176	rdas
ice	✓	180	fdas
c ua	✓	184	opc_client_cert

Figure 7.3: CNAO OIDConfig's navigation form.

Figure 7.3 displays the application with an open navigation tab, allowing users to navigate to data management forms for users, clients, and protected resources. In the figure, the protected resource selection form is present, although greyed out. The selection form presents a read only grid containing the currently configured protected resources, allowing the operator to select which one to edit. Figure 7.4 illustrates the edition form of a protected resource, which is accessed from the navigation form. In the protected resource management form, the operator can edit the protected resource's data. This form also allows users to navigate between the managed resources by selecting the navigation arrows. In the lower half of this form, the grid displays and allows the maintenance of all the scopes of the currently selected resource. It is also possible for operators to, by selecting the appropriate tab, set the grid to display instead the resource secrets.

Finally, the CNAO OIDConfig application logs several events to the repository, as well as to a remote device, using the RDAS and FDAS service respectively.



Resource ID: 176
Display Name: Relational Data Access Serv
Resource Name: rdas
Enabled: On

Resource Description:
Relational Data Access Service

ResourceScopes
ResourceSecrets

Description	Displayname	Emphasize	Is Api Scopes Pk	Name	Required	Show In Discovery Docu
Click here to add a new row						
	Full access to RDAS	☐	179	rdas.full_access	False	☑
	Is allowed to log to RD/	☐	177	rdas.log	False	☑
	Read only access to RD.	☐	178	rdas.read_only	False	☑

Figure 7.4: CNAO OIDConfig protected resource management form.

7.2.2 Integration with LabVIEW: EasyLoader pilot application

In this section, we present the preliminary results of the integration with the LabVIEW environment project. This project was spearheaded by control system developers of the LabVIEW environment, aided by the PhD candidate and the PhD candidate's supervisor. The goal of this project was to integrate the standard services developed in this work, allowing them to be consumed by LabVIEW applications of the upper levels of the control system.

The current scope of the project so far is the integration of the repository access service, which is the RDAS server. The details of this project have been described in Section 4.7. As part of the integration project, the Query.lvlib LabVIEW library was developed, which interacts with the RDAS server via HTTP protocol, and allows applications to request a pre-defined set of parametrized queries to be executed.

After the development and testing of the Query.lvlib library, the LabVIEW environment developers can use the library in newly developed applications for the facility's control system. In this section, we present the first of such applications to be developed, named EasyLoader. First, we describe the context and objectives of the EasyLoader application, which are to be used in the accelerator's daily quality assurance procedure. Later we present the user interface of the application.

EasyLoader – Context and usage

The EasyLoader application is used as part of the accelerator's beam quality assurance procedures, when the geometric characteristics of the accelerator beam are no longer adequate. A procedure called beam steering is performed daily during the quality assurance tests of the accelerator. This procedure calculates the geometric characteristics of the beam produced by the accelerator and evaluates whether the beam meets the requirements for treatment of patients. In CNAO, the geometry of the produced beam varies over time, mostly due to environmental conditions effects on the accelerator components [96].

If the beam characteristics are less than desirable, the geometry calculated by the steering procedure is used by correction calculation software developed by CNAO's optical physicists. This software calculates the necessary corrections to be applied to the accelerator's magnets in order to return the beam geometry to the desired parameters. The correction calculation software's output is a XLS file containing the corrections to be applied to the equipment, which are named setpoints [96].

In the initial operation of the control system, the files containing the new setpoints were manually transferred by operators into the destination equipment using the file transfer protocol (FTP). Later, additional features were desired, such as keeping a record of all previous setpoints used, the ability to detect and recover transfer errors, and rollback features. In order to achieve this, the initial version of the Easy Loader application was designed.

The EasyLoader application was then developed to provide an interactive, graphical application for operators to obtain the XLS files, and transfer them to power supply controllers, while keeping track of changes in the control system's repository. The application obtains the needed XLS files through a shared folder in a remote network drive, which is accessible to several workstations. EasyLoader then accesses the power supply controllers that require new setpoint values through a custom protocol on top of a TPC-IP connection. Finally, Easy Loader accesses the repository through the Query.lvlib library, which communicates with the RDAS server. An allocation view that maps each software component into its respective control system equipment is presented in Figure 7.5 [96].

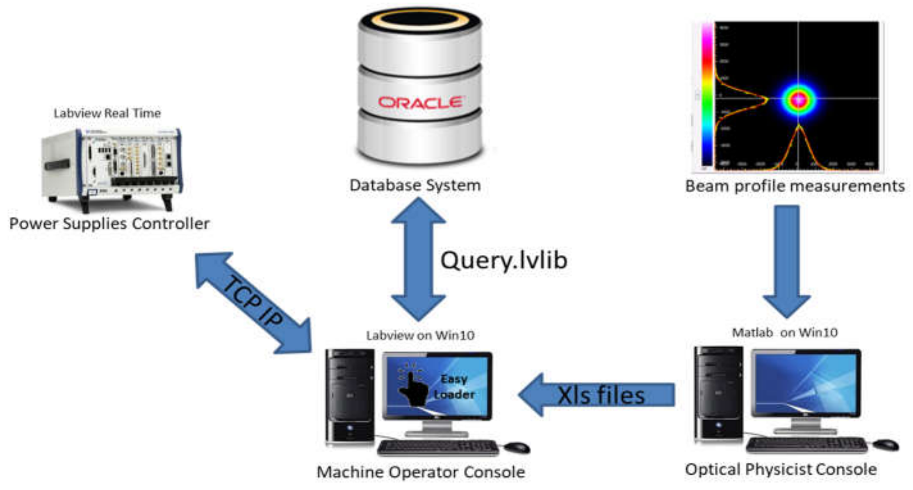
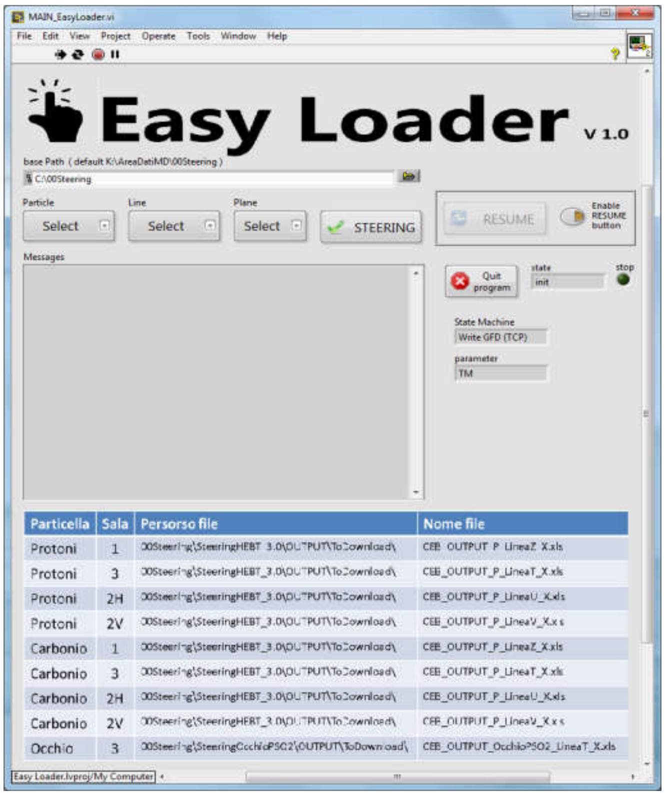


Figure 7.5: Diagram illustrating Easy Loader application as well as the control system components it communicates with. Extracted from [96].

The EasyLoader application can be launched by control system operators from the control room's workstations. The user interface of the application is displayed in Figure 7.6. The user interface has been designed to minimize the available number of actions that must be performed by control system operators. By doing so, the LabVIEW development group aimed at reducing the time to perform the correction upload, as well as reducing the probability of operator error [96].



base Path (default K:\AreaDatiMD\00Steering)
C:\00Steering

state: init

State Machine: Write GFD (TCP)

parameter: TM

Particella	Sala	Persorso file	Nome file
Protoni	1	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaZ_X.xls
Protoni	3	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaT_X.xls
Protoni	2H	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaU_X.xls
Protoni	2V	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaV_X.xls
Carbonio	1	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaZ_X.xls
Carbonio	3	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaT_X.xls
Carbonio	2H	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaU_X.xls
Carbonio	2V	00Steering\SteeringHEBT_3.0\OUTPUT\ToDownload\	CEB_OUTPUT_P_LineaV_X.xls
Occhio	3	00Steering\SteeringOcchioPSO2\OUTPUT\ToDownload\	CEB_OUTPUT_OcchioPSO2_LineaT_X.xls

Figure 7.6: EasyLoader user interface.

Because all the operations performed are saved in the repository and in a log file, the operator can later review the procedures performed and rollback the steering procedure if the procedure does not produce a desirable result.

The EasyLoader application requests two queries types to the repository via the RDAS server, which have been developed by the configuration and support environment developers, in the process explained Section 4.4.6.

7.3 Evaluation

In this section, we evaluate the configuration and support environment in comparison to the legacy environment. First, we evaluate the individual components developed during this work, which are the services, libraries, and tools. In cases where these components have replaced previous ones from the legacy environment, we evaluate them in comparison to their legacy counterparts. Later, the evaluation scope broadens to the environment itself. This evaluation compares the newly available features developed for applications of the environment, and how these features impact the control system. Additionally, we present our estimation of the impact on several quality attributes, based on the design tactics defined in the product line architecture.

7.3.1 Component evaluation

The majority of the time allocated for this project was spent designing and developing several software components to be used by several applications of the upgraded environment. These reusable components include several web services, libraries, and development tools.

In cases where the scope of the newly developed components was identical or constitute an extension of those of their legacy counterpart, comparisons are made taking account any additional features, as well as changes in the quality attributes of the developed components. For those components which were not designed based on a legacy counterpart, and thus added new features to applications of the environment, we instead evaluate them based on the impact of these features.

RDAS

The repository data access service, which is composed of the RDAS server and client, performs a similar role as the previous repository access application library. The main difference is that the previous solution was implemented as an application library, and so, all environment applications were accessing the repository by itself. Previously, these applications would have to be given the access credentials to the repository. Additionally, only applications developed in the software programing environment targeted by the library (C# desktop applications, running on top of the .Net Framework CLI) could access the repository.

The new RDAS server allows all HTTP enabled CNAO applications to access the repository, provided they have the authorization to do so. While applications in the upgraded environment should use RDAS client library to access the RDAS service, any application can access the service via the HTTP protocol. This has been shown in the LabVIEW integration project, where LabVIEW applications are being allowed to access the repository via the RDAS service.

Using the RDAS client library, applications can now access the repository using object-oriented models due to the developed ORM features. Previously, the repository access library returned generic matrix objects results, which then had to be cast into the correct types.

Previously, as mentioned in section 4.1, the SIprod.DataComponent library performed presentation layer operations. By not depending on the presentation layer, developers can more easily unit test domain layer application classes. Therefore, the usage of the RDAS library over the legacy repository access library should improve the testability and maintainability of the environment's application.

FDAS

The FDAS service was developed due to the incompatibility with the legacy solution for file sharing with mobile devices, which relied on remote drives and the FTP protocol. The main advantage of the FDAS service over the legacy solution is the greater degree of control allowed to developers over the files exposed and permissions required. The FDAS service relegates client authorization to the Identity Provider, which can check for application as well as user permissions.

Additionally, because the API to access and expose remote files is now owned and maintained by CNAO, the control system, it can be maintained according to the needs of the facility.

Identity Provider

There were several changes in authentication and authorization in the environment upgrade. Previously, authentication was performed by the application itself, which evaluated the user's credentials through CNAO's LDAP service. In the upgraded environment, the process is no longer performed by the applications, as resource providers require access tokens signed by the identity provider to be delivered by clients. If the resource providers did not verify the user's authentication, then other applications inside the local network could maliciously access these resources without permission.

In the upgraded environment, client and user permissions are set in the repository. Therefore, developers are no longer required to decide the permissions of their applications during development, as these are set in the repository by a control system administrator. This allows administrators to change client permissions without needing to perform updates to the applications.

By employing an identity provider to perform authentication and authorization, the control system becomes more complex. If the identity provider fails, the environment's applications will no longer be able to access resources until the identity provider is restored. However, by centralizing this operation, the identity provider can be made to log every resource authorization request. In order to improve the reliability issues caused by introducing a centralized component that performs access control, several tactics can be implemented in the future, such as redundancy. Currently, several control system components use redundancy techniques to improve their reliability, such as the timing system.

While not currently foreseen, the identity provider could allow the services (such as RDAS server, and FDAS server) to be exposed outside of CNAO's local network, allowing several applications to function outside of the facility's premises. This is due to the fact that the resource providers delegate the access control to the OpenID Connect identity provider, and OpenID Connect is a widely adopted authentication protocol.

Overall, we believe that the implemented changes in access control offer the control system a greater degree of security and maintainability, through the usage of a widely used, standardized protocol. However, because the solution requires a dedicated component (the identity provider), the solution may impact the system's reliability. In production scenarios, addressing possible reliability issues can be performed by using redundancy techniques.

OPC-UA Communication

The CTSIprodOPCUA.v2 library was almost entirely a port of its legacy counterpart. The main architectural change was the removal of presentation layer operations, which promotes interoperability and improves testability.

Logging Operations

The logging solution developed contains several new features that can make logging easier and more flexible. By permitting that applications of the configuration and support environment to use their standardized configuration file to define where events will be logged, we aim to achieve standardized remote logging in the environment.

In two of the logging endpoints developed in this work (repository and remote file logging), the application only delivers the logs to a server, which then decides how to store them. By doing so, these applications do not have to be changed or recompiled if, in the future, these servers are required to process the logs differently.

Configuration loading

The configuration loading features developed consist of an extension over the legacy solution, providing an equivalent interface, and adding object-oriented configuration loading. By standardizing the process of configuration setting, casting and validation, we expect that applications of the environment achieve better reliability.

Additionally, by providing an implementation to this commonly used operation, we hope to slightly shorten the future applications' development time.

7.3.2 Environment evaluation

The expected date for the integration of the first upgraded environment applications into the control system is the beginning of 2020. As a result, evaluating the upgraded environment, in comparison to the former, has been performed based on the newly developed environment features, and the expected quality attributes of applications.

The main improvement to the control system has been the addition of the mobile devices as client platforms of the control system. As defined in the product line architecture, in Section 3, applications and libraries target a common runtime environment that allows them to be executed in Windows workstations and Android mobile devices. Other control system components, which had dependencies only present in the workstations, such as direct access to the repository, and local network drives, now can access these as web services, allowing applications in mobile devices to consume them as well.

As mentioned previously, we argue that, overall, resource access control has been improved by the upgrade. The adoption of an authentication and authorization standard that is widely adopted in the industry improves application security over implementing a custom authentication workflow.

The product line architecture developed for this work heavily emphasizes the testability quality attribute. In order to facilitate application testing, the architecture dictates the usage of several software patterns. These patterns include MVVM, which separates presentation code from domain code, dependency injection to separate the instantiation of software elements from their usage, as well as implementing commonly used operations into standard service libraries. Applications in the legacy environment were also designed to promote testability, through the implementation of several standardized operations as reusable libraries. In conclusion, with the usage of the software patterns required, we argue that the upgraded environment improves the testability of final applications.

During this project, we also analyzed the medical software certification process, and tailored the product line architecture to aid the certification process. As a result, we expect that the upgraded environment's applications that follow the guidelines provided in the product line architecture, to be more easily certifiable than their legacy counterparts. However, due to time constraints, this will have to be empirically evaluated after the end of this work.

The new wizard generator is a more ambitious automatic code generation solution than the previously existing solution. Currently, we have two wizard generators in the final stages of development, as well as a large amount of pre-defined configurable pages, which can be added into applications. While further development is still required for the generators, the initial results are encouraging.

Finally, the technological upgrade of the environment was necessary to phase out technologies which were expected to have limited support in the next years. By using more recent development platforms and technologies, developers of the control system will have access to a greater amount of commercial and open source libraries to choose from in the future.

CHAPTER 8

Conclusion and future work

8.1 Conclusion

Hadrontherapy is a very important treatment option for patients with cancer. Certain varieties of cancer, such as radio-resistant cancers, or those that surround critical organs are usually better treated with hadrontherapy rather than radiotherapy. Control system software is essential for the operation of the large particle accelerators required for hadrontherapy.

The goal of this work was the development of tools, libraries, and services necessary for the future applications of the upgraded configuration and support environment of CNAO. The major features added to the upgraded environment are the integration of mobile devices, and the update of the technologies of the development environment. In order to do so, the legacy environment was reviewed, and requirements were analyzed. Based on these requirements, a product line architecture was designed by defining the scope of future applications, defining the services and libraries available to them, and providing an overview of their architecture.

Afterwards, several services were designed and developed to support the future environment's applications. These services allow applications to access other control system components and expose their operations through a RESTful API. Several libraries were also designed and developed for these applications, and can be classified in two categories: libraries that perform communication with services, and libraries that implement commonly required operations.

The preliminary usage of the services and their respective client libraries has yielded favourable results. Prototype applications in all devices targeted have been able to consume the services defined in this work. By remaining separate from applications of the environment, the services can undergo maintenance independently for adding new features or fixing issues.

Since the beginning of this work, the internal demonstrations of the services have led to the approval of a separate project, performed by the control system's LabVIEW development group. This project, which has been performed in cooperation with this work, aims at allowing LabVIEW applications on the control system's first levels to be integrated with several of the configuration and support environment's services.

In order to assist the developers to implement the architectural components required by the product line architecture, a set of frameworks were developed. These frameworks, and their supporting libraries, perform the initialization of applications, initialize their service library components, and provide several base implementations of software patterns that applications must use.

In order to simplify the usage of each framework, a wizard generator Visual Studio extension has been developed to generate configurable skeleton applications. These skeleton applications are fully functional configured base applications for the developer to build upon. Additionally, using the data management wizard generator, we have developed a pilot application, showcasing the features of future applications in the new environment.

Later, analysis of the medical software certification process, as well as the expected benefits from adopting the product line approach, was presented. Since control systems use a large amount of custom software, the effort of producing the documentation and testing for certification requires a large portion of development time. Consequently, research focused on designing control system software focusing on testability and reusability allows the developers to more easily perform the required testing.

In conclusion, this work has demonstrated the viability of a product line architecture focused approach when performing technological upgrade of a medical accelerator control system. Additionally, the service-oriented architecture principles used in the definition of the environment’s infrastructure have been successful and are already spreading to other development groups in the facility.

8.2 Future work

The totality of the configuration and support environment upgrade project has a bigger scope than this work. While the work presented in this thesis had the goal of developing tools, services, and components for the upgraded environment, the total project encompasses the development of the environment’s applications, as well as performing their respective medical software certification tasks.

The authors expect the remaining members of the configuration and support environment to continue after the end of this work. In the near future, several applications of the legacy will have to be reviewed, and the total amount of new required applications will have to be analyzed by the control system stakeholders. During the development, metrics can be gathered, and the product line approach can be further evaluated. Particularly, metrics such as required number of hours for development and certification, as well as number and severity of errors encountered during testing would provide insight on the impact of the architecture.

There are also several other tasks in adjacent topics that could be carried out as projects on their own. Further work can be performed in bringing service-oriented architecture designs to accelerator control systems. Research on further architectural tactics aimed at enabling services to possess real-time behavior and assuring the dependability of control system services would be useful. As mentioned in chapter 2, in control systems with several levels, each level is usually characterized by their real-time constraints. Applicable tactics for ensuring different real-time behaviors onto the services could be compiled into a reference architecture. The reference architecture would then be consulted by developers in order to understand the features that would be needed in order to fit their control system environment's constraints.

Regarding security, extending the OPC-UA protocol in order to support the delegation of authentication to OpenID Connect identity providers could be carried out. By using authorization tokens, authenticated users and clients communicate securely to other devices using the OPC-UA protocol. Currently, there is an OPC-UA conformance profile that denotes OPA-UA enabled devices which are able to use tickets from another access control standard (Kerberos) as proof of user authentication [113]. Because the OpenID Connect standard allows for the authentication of both clients and users, identity tokens provided by identity providers could be used as proof of client and user authentication.

Alternatives that would not require extensions to the OPC-UA protocol to be made can also be researched. Further research into providing certificate authority capabilities to an identity provider can be done, extending the work presented in Section 4.4.6. This would allow further integration between web services and OPC-UA enabled devices, as the authentication process could be used to obtain access tokens and the required certificates. Alternatively, the development of a certificate authority that delegates authorization to an identity provider is also an option. Because the certificates would have to be transferred from the certificate authority to OPC-UA clients, study on additional measures for securing the certificates is essential to these two approaches.

References

[1] U. Amaldi, “History of hadrontherapy in the world and Italian developments,” *Riv. Medica*, vol. 14, no. 1, 2008.

[2] J. H. Lawrence, P. C. Aebersold, and E. O. Lawrence, “Comparative effects of x-rays and neutrons on normal and tumor tissue,” *Proc. Natl. Acad. Sci. U. S. A.*, vol. 22, no. 9, p. 543, 1936.

[3] U. Amaldi *et al.*, “Accelerators for hadrontherapy: from Lawrence cyclotrons to linacs,” *Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip.*, vol. 620, no. 2–3, pp. 563–577, 2010.

[4] R. S. Stone, “Neutron Therapy and Specific Ionization,” *Am. J. Roentgenol. Radium Ther.*, vol. 59, pp. 771–785, 1948.

[5] D. Schulz-Ertner, O. Jäkel, and W. Schlegel, “Radiation therapy with charged particles,” in *Seminars in radiation oncology*, 2006, vol. 16, no. 4, pp. 249–259.

[6] U. Amaldi and G. Kraft, “Radiotherapy with beams of carbon ions,” *Reports Prog. Phys.*, vol. 68, no. 8, pp. 1861–1882, Jul. 2005.

[7] W. Maciszewski and W. Scharf, “Particle accelerators for radiotherapy: Present status and future,” in *Astroparticle, Particle And Space Physics, Detectors And Medical Physics Applications*, World Scientific, 2004, pp. 402–410.

[8] A. J. Lennox, F. R. Hendrickson, D. A. Swenson, R. A. Winje, and D. E. Young, “Proton linac for hospital-based fast neutron therapy and radioisotope production,” 1989.

[9] Particle Therapy Co-operative Group, “Particle therapy facilities in clinical operation (last update: April 2019).” [Online]. Available: https://www.ptcog.ch/index.php/facilities-in-operation. [Accessed: 16-May-2019].

[10] D. Ungaro, A. Degiovanni, and P. Stabile, “LIGHT: A Linear

Accelerator for Proton Therapy,” in *North American Particle Accelerator Conf.(NAPAC’16), Chicago, IL, USA, October 9-14, 2016*, 2017, pp. 1282–1286.

[11] A. Degiovanni *et al.*, “A Cyclotron+ Linac Complex for Carbon Ion Therapy,” in *talk in workshop on" Physics for Health in Europe"*, 2010.

[12] IBA, “IBA Worldwide - Shaping the future of proton therapy.” [Online]. Available: https://iba-worldwide.com/proton-therapy. [Accessed: 25-Jul-2019].

[13] Varian, “Proton Therapy | Varian Medical Systems.” [Online]. Available: https://www.varian.com/oncology/solutions/proton-therapy. [Accessed: 25-Jul-2019].

[14] S. Peggs, T. Satogata, and J. Flanz, “A survey of hadron therapy accelerator technologies,” in *2007 IEEE Particle Accelerator Conference (PAC)*, 2007, pp. 115–119.

[15] Advanced Oncotherapy, “AVO | Proton Therapy Specialist.” [Online]. Available: https://www.avoplc.com/. [Accessed: 03-Jul-2019].

[16] TERA, “TERA - Fondazione per Adroterapia Oncologica.” [Online]. Available: http://www.tera.it/. [Accessed: 03-Jul-2019].

[17] R. Mueller, “Control Systems for Accelerators: Operational Tools,” 2016, pp. 629–670.

[18] “ICALEPCS - About.” [Online]. Available: https://www.icalepcs.org/icalepcs.html. [Accessed: 22-May-2019].

[19] A. Daneels and W. Salter, “What is SCADA?,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 1999.

[20] B. Galloway and G. P. Hancke, “Introduction to industrial control networks,” *IEEE Commun. Surv. tutorials*, vol. 15, no. 2, pp. 860–880, 2012.

[21] B. Kuiper, “Issues in accelerator controls,” in *Proceedings of International Conference on Particle Accelerators*, 1991, pp. 602–211.

[22] M. E. Thuot and L. R. Dalesio, “Control system architecture:

the standard and nonstandard models," in *Proceedings of International Conference on Particle Accelerators*, 1993, pp. 1806–1810.

[23] J. Zhang, T. R. Johnson, V. L. Patel, D. L. Paige, and T. Kubose, "Using usability heuristics to evaluate patient safety of medical devices," *J. Biomed. Inform.*, vol. 36, no. 1, pp. 23–30, 2003.

[24] R. Štefanič, R. Tavčar, J. Dedič, J. Gutleber, and R. Moser, "Timing System Solution for Medaustron; Real-Time Event and Data Distribution Network," in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2011.

[25] L. R. Dalesio, A. J. Kozubal, and M. R. Kraimer, "EPICS architecture," in *Proceedings of International conference on accelerator and large experimental physics control systems*, 1991.

[26] J. Chaize, W. Klotz, J. Meyer, M. Perez, and E. Taurel, "TANGO: An Object Oriented Control System Based on CORBA," in *Proceedings of International conference on accelerator and large experimental physics control systems*, 1999, pp. 475–479.

[27] M. Stal, F. Buschmann, and R. Meunier, *Pattern-oriented Software Architecture—A System of Patterns*. Wiley, 1996.

[28] L. R. Dalesio *et al.*, "The experimental physics and industrial control system architecture: past, present, and future," *Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip.*, vol. 352, no. 1–2, pp. 179–184, 1994.

[29] M. Knott, D. Gurd, S. Lewis, and M. Thuot, "EPICS: A Control System Software Co-Development Success Story," *Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip.*, vol. 352, no. 1–2, pp. 486–491, 1994.

[30] W. P. Mcdowell *et al.*, "Standards and the design of the Advanced Photon Source control system," in *Proceedings of International conference on accelerator and large experimental physics control systems*, 1991.

[31] H. Michi and V. Steve, *Advanced CORBA®programming with C++*. Addison-Wesley, 1999.

[32] A. Götz *et al.*, “The future of TANGO,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2007.

[33] A. Götz *et al.*, “The TANGO CONTROLS collaboration in 2015,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2015.

[34] E. Taurel *et al.*, “TANGO a CORBA based Control System,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2003.

[35] National Instruments, “What is LabVIEW?” [Online]. Available: http://www.ni.com/en-us/shop/labview.html. [Accessed: 05-Jun-2019].

[36] National Instruments, “PXI Systems - National Instruments.” [Online]. Available: http://www.ni.com/en-us/shop/pxi.html. [Accessed: 05-Jun-2019].

[37] Siemens, “SCADA System SIMATIC WinCC V7.” [Online]. Available: https://w3.siemens.com/mcms/human-machine-interface/en/visualization-software/scada/pages/default.aspx. [Accessed: 05-Jun-2019].

[38] Vista Control Systems, “Vista Control Systems Homepage.” [Online]. Available: https://www.vista-control.com. [Accessed: 31-May-2019].

[39] B. Mannix and T. Gray, “Vista Controls Vsystem at the ISIS pulsed neutron facility,” 2007.

[40] ISIS Neutron and Muon Source, “Accelerators and Targets: People.” [Online]. Available: https://www.isis.stfc.ac.uk/Pages/Accelerators-and-Targets-People.aspx. [Accessed: 31-May-2019].

[41] A. Daneels and W. Salter, “Selection and evaluation of commercial SCADA systems for the controls of the CERN LHC experiments,” *Proc. Int. Conf. Accel. large Exp. Phys. Control Syst.*, 1999.

[42] P. C. Burkimsher, “Jcop experience with a commercial scada product, pvss,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2003.

[43] National Instruments, “LabVIEW Datalogging and

Supervisory Control Module.” [Online]. Available: https://www.ni.com/en-us/shop/select/labview-datalogging-and-supervisory-control-module. [Accessed: 05-Jun-2019].

[44] J. Gutleber, A. Brett, R. Moser, M. Marchhart, C. T. de Matos, and J. Dedič, “The MedAustron accelerator control system,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2011.

[45] V. Aleinikov, I. Borina, A. Krylov, S. Pachtchenko, and K. Sychev, “Using LabVIEW to Build Distributed Control System of a Particle Accelerator,” in *Proceedings of International conference on accelerator and large experimental physics control systems*, 2017.

[46] S. Rossi, “The status of CNAO,” *Eur. Phys. J. Plus*, vol. 126, no. 8, p. 78, Aug. 2011.

[47] U. Amaldi and G. Tosi, “Per un centro di teleterapia con adroni,” *TERA 91/1 GEN*, vol. 1, 1991.

[48] Tera Foundation, *The Tera Project and the Centre for Oncological Hadrontherapy*. INFN-LNF Divisione Ricerca, 1995.

[49] Tera Foundation, *The national centre for oncological hadrontherapy at Mirasole*. 1997.

[50] L. Badano *et al.*, “Proton-Ion Medical Machine Study (PIMMS), 1,” 1999.

[51] P. J. Bryant *et al.*, “Proton-Ion Medical Machine Study (PIMMS), 2,” 2000.

[52] P. Mandrillon *et al.*, “Feasibility study of the EULIMA light ion medical accelerator,” in *Proceedings of the Third European Particle Accelerator Conference, Berlin*, 1992, vol. 92, pp. 179–181.

[53] S. Rossi, “The National Centre for Oncological Hadrontherapy (CNAO): Status and perspectives,” *Phys. Medica*, vol. 31, no. 4, pp. 333–351, 2015.

[54] L. Casalegno, M. Pezzetta, and S. Toncelli, “CNAO General Control System Organization Document - Unpublished internal document,” 2003.

[55] L. Casalegno, M. Pezzetta, and S. Toncelli, “Timing and

Signal Distribution Services Requirements Specification Document - Unpublished internal document," 2004.

[56] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Addison-Wesley Professional, 2003.

[57] S. Gioia, "Presentation, Supervision Procedures," 2016.

[58] M. Caldaram, L. Casalegno, A. Parravicini, M. Pezzetta, M. Pullia, and S. Toncelli, "Control System Short Overview - Unpublished internal document," 2004.

[59] L. Casalegno, "The CNAO RT Patient Scheduler," 2013.

[60] L. Casalegno, "Istruzione Operativa Treatment Monitor - Unpublished Internal Document," 2017.

[61] L. Casalegno, "Istruzione Operativa CNAOHistory - Unpublished Internal Document," 2019.

[62] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[63] P. Clements, "Software product lines," *Pract. patterns*, 2001.

[64] J. D. McGregor, "Software Product Lines," *Technology*, vol. 3, no. 3, pp. 65–74, 2004.

[65] S. Angelov, P. W. P. J. Grefen, and D. Greefhorst, "A classification of software reference architectures: Analyzing their success and effectiveness," *2009 Jt. Work. IEEE/IFIP Conf. Softw. Archit. Eur. Conf. Softw. Archit.*, pp. 141–150, 2009.

[66] E. Y. Nakagawa, P. O. Antonino, and M. Becker, "Reference architecture and product line architecture: A subtle but critical difference," in *European Conference on Software Architecture*, 2011, pp. 207–211.

[67] L. Casalegno, "Istruzione Operativa Trlinesloaderfast - Unpublished Internal Document," 2017.

[68] L. Casalegno, "Istruzione Operativa MTCM - Unpublished Internal Document."

[69] D. Krafzig, K. Banke, and D. Slama, *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005.

[70] L. Casalegno, M. Pezzetta, and S. Toncelli, "Standards, common rules and best practices in the development of the

CNAO control system - Unpublished internal document," 2004.

[71] Microsoft, "Active Directory Domain Services | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-domain-services. [Accessed: 17-Jun-2019].

[72] "Xamarin | Open-source mobile app platform for .NET." [Online]. Available: https://dotnet.microsoft.com/apps/xamarin. [Accessed: 19-Sep-2019].

[73] Microsoft, "Introduction to Portable Class Libraries (PCL) - Xamarin | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/pcl?tabs=windows. [Accessed: 18-Jun-2019].

[74] Microsoft, ."NET Standard | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/net-standard. [Accessed: 18-Jun-2019].

[75] Microsoft, "Introduction to DependencyService - Xamarin | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/app-fundamentals/dependency-service/introduction. [Accessed: 18-Jun-2019].

[76] C. Campbell, F. Cheung, D. A. Poza, R. Sharma, M. Vazquez, and B. Wastell, "Prism for the Windows Runtime for Windows 8.1," 2013. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=39042. [Accessed: 27-Mar-2019].

[77] Microsoft, "The Model-View-ViewModel Pattern - Xamarin | Microsoft Docs." [Online]. Available: https://docs.microsoft.com/en-us/xamarin/xamarin-forms/enterprise-application-patterns/mvvm. [Accessed: 19-Jun-2019].

[78] N. Sakimura, J. Bradley, M. Jones, B. de Medeiros, and C. Mortimore, "OpenID Connect Core 1.0 incorporating errata set 1," *The OpenID Foundation, specification*, 2014. [Online]. Available: https://openid.net/specs/openid-connect-core-1_0.html. [Accessed: 27-Mar-2019].

[79] Microsoft, “ADO.NET Overview | Microsoft Docs.” [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ado-net-overview. [Accessed: 21-Jun-2019].

[80] Devart, “ADO.NET Data Provider for Oracle with Entity Framework Support.” [Online]. Available: https://www.devart.com/dotconnect/oracle/. [Accessed: 21-Jun-2019].

[81] Oracle, “Oracle Instant Client - Free tools and libraries for connecting to Oracle Database.” [Online]. Available: https://www.oracle.com/database/technologies/instant-client.html. [Accessed: 21-Jun-2019].

[82] Connect2id, “LDAP user authentication explained | Connect2id.” [Online]. Available: https://connect2id.com/products/ldapauth/auth-explained. [Accessed: 08-Jul-2019].

[83] OpenID, “Welcome to OpenID Connect,” 2014. [Online]. Available: http://openid.net/connect/.

[84] D. Hardt, “The OAuth 2.0 authorization framework,” 2012.

[85] Justin Richer, “End User Authentication with OAuth 2.0 — OAuth.” [Online]. Available: https://oauth.net/articles/authentication/. [Accessed: 08-Jul-2019].

[86] Okta, “Authorization Code Grant - OAuth 2.0 Servers.” [Online]. Available: https://www.oauth.com/oauth2-servers/server-side-apps/authorization-code/. [Accessed: 08-Jul-2019].

[87] D. Baier, B. Allen, and other Github contributors, “IdentityServer4 Framework.” [Online]. Available: https://github.com/IdentityServer/IdentityServer4.

[88] “IdentityServer 4 Quickstart Samples.” [Online]. Available: https://github.com/IdentityServer/IdentityServer4/tree/master/samples/Quickstarts. [Accessed: 21-Aug-2019].

[89] IdentityServer4 development team, “Profile Service — IdentityServer4 1.0.0 documentation.” [Online]. Available: https://identityserver4.readthedocs.io/en/latest/reference/profileservice.html. [Accessed: 09-Jul-2019].

[90] J. Richer, “User Authentication with OAuth 2.0.” [Online].

Available: oauth.net/articles/authentication/.

[91] B. Allen and D. Baier, “Grant Types — IdentityServer4 1.0.0 documentation.” [Online]. Available: http://docs.identityserver.io/en/latest/topics/grant_types.html. [Accessed: 10-Jul-2019].

[92] “Identity Model Library.” [Online]. Available: https://github.com/IdentityModel/IdentityModel.OidcClient2. [Accessed: 21-Aug-2019].

[93] OpenID Foundation, “Certified OpenID Connect Implementations | OpenID.” [Online]. Available: https://openid.net/developers/certified/. [Accessed: 08-Jul-2019].

[94] Serilog contributors, “Serilog — simple .NET logging with fully-structured events.” [Online]. Available: https://serilog.net/. [Accessed: 14-Jul-2019].

[95] Serilog contributors, “Serilog — simple .NET logging with fully-structured events.” .

[96] S. Foglio, C. Viviani, and L. Casalegno, “Use of the CNAO Query.lvlib in the EasyLoader application.”

[97] R. E. Johnson, “Frameworks=(components+ patterns),” *Commun. ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[98] M. Fayad and D. C. Schmidt, “Object-oriented application frameworks,” *Commun. ACM*, vol. 40, no. 10, pp. 32–38, 1997.

[99] IEC, “IEC 62304:2006 - Medical device software -- Software life cycle processes,” May 2006.

[100] IEC, “IEC 62304:2006/Amd 1:2015 - Medical device software - Software life-cycle processes / Amd 1,” 2015.

[101] ISO, “ISO 14971:2007 - Medical devices -- Application of risk management to medical devices,” Mar. 2007.

[102] ISO, “ISO 13485:2016 - Medical devices -- Quality management systems -- Requirements for regulatory purposes,” Mar. 2016.

[103] N. Hrgarek, “Certification and regulatory challenges in medical device software development,” in *Proceedings of the 4th International Workshop on Software Engineering in Health Care*, 2012, pp. 40–43.

[104] J. Rushby, “Modular certification,” Menlo Park, CA, 2002.

[105] A. Höss, C. Lampe, R. Panse, B. Ackermann, J. Naumann, and O. Jäkel, “First experiences with the implementation of the European standard EN 62304 on medical device software for the quality assurance of a radiotherapy unit,” *Radiat. Oncol.*, vol. 9, no. 1, p. 79, 2014.

[106] P. Rust, D. Flood, and F. McCaffery, “Creation of an IEC 62304 compliant software development plan,” *J. Softw. Evol. Process*, vol. 28, no. 11, pp. 1005–1010, 2016.

[107] P. M. Nadkarni and R. A. Miller, “Service-oriented architecture in medical software: promises and perils,” *J. Am. Med. Informatics Assoc.*, vol. 14, no. 2, pp. 244–246, 2007.

[108] J. Hatcliff *et al.*, “Rationale and architecture principles for medical application platforms,” in *Proceedings of the 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems*, 2012, pp. 3–12.

[109] R. Land, M. Åkerholm, and J. Carlson, “Efficient software component reuse in safety-critical systems--an empirical study,” in *International Conference on Computer Safety, Reliability, and Security*, 2012, pp. 388–399.

[110] B. Councill and G. T. Heineman, “Component-based Software Engineering,” G. T. Heineman and W. T. Councill, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 5–19.

[111] J. Smith, “Patterns-wpf apps with the model-view-viewmodel design pattern,” *MSDN Mag.*, vol. 72, 2009.

[112] C. F. Afonso, L. Casalegno, and C. Larizza, “Certification of Component-Based Particle Therapy Software,” in *Proceedings of the IADIS International Conference e-Health 2019, Part of the IADIS Multi Conference on Computer Science and Information Systems 2019, MCCSIS 2019*, 2019.

[113] “‘User Token - Kerberos Server Facet’ Profile.” [Online]. Available: http://opcfoundation-onlineapplications.org/ProfileReporting/index.htm?ModifyProfile.aspx?ProfileID=32428471-43c8-4f67-ae40-099a6b475b51. [Accessed: 22-Aug-2019].