Bruno Damião Areias Gandres

MAINTENANCE PRACTICES IN SOFTWARE ENGINEERING

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering advised by Professor Marília Curado, Eng. Ricardo Gonçalves and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

DEPARTMENT OF INFORMATICS ENGINEERING

Bruno Damião Areias Gandres

Maintenance Practices in Software Engineering

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Marília Curado, Eng. Ricardo Gonçalves and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

Acknowledgements

I would like to express my sincere gratitude to my Dissertation advisors, Prof. Marília Curado and Eng. Ricardo Gonçalves, for their invaluable guidance and support throughout my work. Their expertise and encouragement helped me to complete Dissertation.

I am grateful to Critical Software for providing me with the opportunity to conduct my internship and for all of the resources and support they provided. I would like to extend a special thanks to Eng. Ricardo Silva, Eng. Bruno Madureira and Eng. José Caetano, who went above and beyond to help me with my work and also to Eng. João Catalão and Eng. João Leal for the integration and good mood every day.

I would also like to thank my colleagues at DEI. In particular, I would like to thank Diogo Filipe, Francisco Bugalho, Henrique Teixeira, José Gomes, José Reis, Pedro Marques and Nuno Silva (and more) for all the memories and support over these five years.

I would also like to thank my family for their love and support during this process. Without them, this journey would not have been possible.

Abstract

With the increasing need to develop software that remains operational over the years, the role of software maintenance has grown in importance. Critical Software (CSW) has been playing a major role in the analysis, development, and maintenance of software, being involved in different sectors of the industry and achieving various certification objectives, thus underscoring the quality of the products it develops.

The main goal of this work is to migrate a component of one of the major systems developed by CSW, from Thorntail to Quarkus technology, using maintenance practises in order to keep the component up to date, since Thorntail reached its end-of-life. The use of legacy technology can bring problems of security and compatibility, and the role of software maintenance is to solve these problems.

This work outlines the steps taken to implement the migration process in accordance with the maintenance process. It gives a real insight into the problems and challenges that are faced in software maintenance work. A study is conducted to introduce the concepts of software maintenance. Additionally, an examination is performed to evaluate a potential alternative to Thorntail within the project's context. The presented work details the entire research and development process undertaken throughout the internship, highlighting the changes and improvements that were made and challenges/problems faced during the migration of the component. The obtained results show that the main objective has been met, maintaining the good functionality after the migration being this objective illustrated in the successful execution of unit and system tests.

Keywords

CSWSYS component, Software Maintenance, Thorntail, Quarkus, Reflection, System Tests

Resumo

Com a crescente necessidade de desenvolver software que se mantenha operacional ao longo dos anos, o papel da manutenção de software tem vindo a crescer em importância. A CSW tem tido um papel preponderante na análise, desenvolvimento e manutenção de software, estando envolvida em diferentes sectores da indústria e atingindo vários objectivos de certificação, evidenciando assim a qualidade dos produtos que desenvolve.

O principal objetivo deste trabalho é migrar um componente de um dos principais sistemas desenvolvidos pela CSW, da tecnologia Thorntail para a tecnologia Quarkus, utilizando práticas e técnicas de manutenção de forma a manter o componente atualizado, uma vez que o Thorntail atingiu o seu fim de vida. A utilização de tecnologia *legacy* pode trazer problemas de segurança e compatibilidade, e o papel da manutenção de software é resolver estes problemas.

Este trabalho apresenta os passos que foram realizados para implementar a migração seguindo o processo de manutenção. Também dá uma visão real dos problemas e desafios que são enfrentados num trabalho de manutenção de software. Foi efectuado um estudo para dar uma introdução aos conceitos de manutenção de software. Além disso, foi feito um estudo para avaliar uma possível alternativa ao Thorntail no contexto do projeto. O trabalho apresentado detalha todo o processo de pesquisa e desenvolvimento realizado durante o estágio, destacando as mudanças e melhorias realizadas e os desafios/problemas enfrentados durante a migração do componente. Os resultados obtidos mostram que o objetivo principal foi cumprido, mantendo a boa funcionalidade após a migração, sendo este objetivo ilustrado na execução com sucesso de testes unitários e de sistema.

Palavras-Chave

CSWSYS component, Manutenção de Software, Thorntail, Quarkus, Reflexão, Testes de Sistema

Contents

1 Introduction **1**
1.1 Project Context . . . 1
1.2 Objectives . . . 2
1.3 Structure of the document . . . 3

2 Practices in Software Maintenance **5**
2.1 Software Life Cycle Processes . . . 5
2.2 Software Maintenance . . . 6
2.2.1 Maintenance Processes . . . 6
2.2.2 Types of Software Maintenance . . . 8
2.2.3 Techniques for Maintenance . . . 10
2.3 Software Maintenance Measurement . . . 12
2.3.1 Size . . . 12
2.3.2 Complexity . . . 13
2.3.3 Quality . . . 13
2.3.4 Understandability . . . 13
2.3.5 Maintainability . . . 14
2.4 Handling Maintenance in Agile Projects . . . 15
2.4.1 Scrum . . . 15
2.4.2 Kanban . . . 17
2.4.3 Differences between Scrum and Kanban . . . 18
2.4.4 Scrumban . . . 18
2.5 Technology Stack . . . 20
2.5.1 Why it is important to maintain a Tech Stack? . . . 20
2.5.2 Technology Stack of CSWSYS . . . 20
2.5.3 Thorntail Background . . . 21
2.5.4 Thorntail Alternatives . . . 21
2.5.5 Comparison . . . 25
2.5.6 Final Thoughts . . . 27

3 Project Background and Approach **29**
3.1 Project Background . . . 29
3.2 Project Requirements . . . 30
3.2.1 Functional Requirements . . . 32
3.2.2 Non-Functional Requirements . . . 32
3.3 CSWSYS Architecture . . . 33
3.4 Approach . . . 34
3.4.1 Approach . . . 35

3.4.2 Initial Plan . . . 37
3.4.3 Planning . . . 37
3.4.4 Risk Plan . . . 39

4 Development, Integration of the solution and Testing **45**
4.1 Environment Setup . . . 45
4.2 CSWSYS Structure and Organization . . . 47
4.2.1 Test Approach . . . 49
4.3 Dependencies Changing . . . 49
4.3.1 Before Migration . . . 51
4.3.2 After Migration . . . 52
4.4 EJB's and Annotations . . . 57
4.4.1 @Health . . . 57
4.4.2 @Stateless . . . 58
4.4.3 Eager Instantiation Beans . . . 58
4.4.4 Private Members . . . 60
4.5 Configuration Files . . . 62
4.6 Unit Tests . . . 66
4.6.1 Replace PowerMock . . . 67
4.6.2 Delete Whitebox . . . 69
4.6.3 Unit testing with Constructor Injection . . . 72
4.7 Regression System Testing . . . 74
4.7.1 Test Environment Setup . . . 75
4.7.2 Defect Fixing . . . 76
4.8 Final considerations . . . 80

5 Conclusion **83**

Appendix A Show and Tell **93**

Appendix B Regression System Testing execution reports **139**

Acronyms

ADA Attribute Anomaly Detect.

API Application Programming Interface.

BOM Bill of Materials.

CDI Contexts and Dependency Injection.

CSW Critical Software.

DAO Data Access Object.

DCC Data Communications Company.

DSP Data Service Provider.

EOL End of life.

JEE Java Enterprise Edition.

JIT Just-in-time.

JPA Java Persistence API.

JVMs Java virtual machines.

KPIs key performance indicators.

LTS long term support.

MTTR Mean Time To Restore.

NFRs Non-functional requirements.

POC proof of concept.

POM Project Object Model.

RSS Resident Set Size.

SDK Software Development Kits.

SLAs Service Level Agreements.

SMIP Smart Metering Implementation Programme.

SOA Service-Oriented Architecture.

SRD System Requirements Definition.

SRS Software Requirements Specifications.

SRVs Service Requests Variant.

SSH Secure Shell.

SSS System/Subsystem Specification.

STS Smart Technology Solutions.

UML Unified Modeling Language.

VM virtual machine.

WIP work-in-progress.

List of Figures

2.1 Maintenance Process (from - [6]) 7
2.2 Modification Request (adapted from [9]) 10
2.3 Spring Boot Vs Quarkus Metrics (from - [47]) 26

3.1 Requirements Hierarchy 31
3.2 Existing Project Architecture 33
3.3 Initial planning for the first semester 37
3.4 First-semester plan 38
3.5 Second-semester initial plan 38
3.6 Second-semester 39
3.7 Risk Matrix (from - [52]) 40

4.1 SDK - Project Structure 46
4.2 CSWSYS Java version 46
4.3 settings.xml 47
4.4 Smart Technology Solutions (STS) Checkstyle 47
4.5 Thorntail Dependencies example 49
4.6 First errors after Java version change 50
4.7 Thontail Bill of Materials (BOM) 51
4.8 Thorntail Metrics 52
4.9 Quarkus Dependency 53
4.10 Quarkus Version 53
4.11 Jandex Maven Plugin 54
4.12 jakarta.el dependency 54
4.13 Microprofile Dependencies 55
4.14 Health import change 58
4.15 Listening for startup event (from - [60]) 58
4.16 Bean in an observer of the StartupEvent (from - [59]) 59
4.17 Startup Annotation (from - [59]) 59
4.18 Startup Approach 59
4.19 Field Injection 61
4.20 Constructor Injection Fields 61
4.21 Constructor Injection 61
4.22 Constructor Injection Error 62
4.23 JDBC driver Error 63
4.24 server-config.yml 63
4.25 Quarkus - Application.yaml 64
4.26 HibernateORM code adaptions 65
4.27 Config property path example 65

4.28 ConfigProperty Variable . . . 65
4.29 Reflection Warning . . . 67
4.30 PowerMock dependency . . . 68
4.31 Mockito dependency replace . . . 68
4.32 PowerMock PrepareForTest Annotation . . . 68
4.33 Unit Test Changes - initMocks() . . . 69
4.34 Unit Test Changes - mockStatic . . . 69
4.35 Mockito-inline dependency . . . 69
4.36 setInternalState Inject Field . . . 70
4.37 Replace of setInternalState with setter . . . 70
4.38 invokeMethod replace . . . 70
4.39 Private Method . . . 70
4.40 Public Method . . . 71
4.41 Package-private Method . . . 71
4.42 Unit Test with package-private method . . . 72
4.43 PostConstruct visibility . . . 72
4.44 setUp function . . . 73
4.45 Close function . . . 73
4.46 Mock Registred in Thread error . . . 74
4.47 Virtual Environment Libraries . . . 75
4.48 Thorntail Startup Logs . . . 77
4.49 Expected message log change . . . 77
4.50 Quarkus Message Log example . . . 77
4.51 SystemTest Error . . . 78
4.52 Old Check Database Connection function . . . 78
4.53 Quarkus Scheduled Annotation . . . 79
4.54 Database Connection State Logs . . . 79
4.55 Database Connection Down Logs . . . 79

A.1 1st Show and Tell . . . 108
A.2 2nd Show and Tell . . . 126
A.3 3rd Show and Tell . . . 138

B.1 Test Report . . . 142
B.2 Test Report . . . 146
B.3 Test Report . . . 150

List of Tables

2.1 Scrum - Pros and Cons (adapted from[24]) 16
2.2 Kanban - Pros and Cons (adapted from[29]) 17
2.3 Scrum vs Kanban (adapted from [30]) 18
2.4 Scrumban Pros and Cons (adapted from [32]) 19
2.5 Quarkus - Pros and Cons (adapted from [42],[43]) 23
2.6 SpringBoot - Pros and Cons (adapted from [44]) 24
2.7 JBoss - Pros and Cons (from [49]) 25
2.8 Quarkus vs Spring Boot Comparison (adapted from [42]) 25

3.1 CSWSYS component static analysis 30
3.2 Risks table 43

4.1 Overall dependency changes 56
4.2 Thorntail CSWSYS component static analysis 80
4.3 Quarkus CSWSYS component static analysis 80
4.4 Overall CSWSYS Changes statistic 81

Chapter 1

Introduction

Software maintenance plays a very important role in the world of systems and applications. A software system is created with the intention of being used for many years, and this implies that it must be compatible with the constant change and advancement of technology.

Continuous software maintenance will guarantee good performance, fewer issues, and the ability to adapt to changes. According to statistics, when it comes to software, 60% of the cost is associated with maintenance. From the overall maintenance costs 60% are spent on improving the solution [1]. If the software is not regularly maintained, the costs of this process can be extremely large, and sometimes the company cannot afford these costs. Software maintenance enhances the growth of the software, and cannot be considered as an option, but as an essential part of the software development.

1.1 Project Context

Critical Software (CSW) provides systems, software and data engineering services for safety, mission and business-critical applications, helping to ensure clients meet the most demanding quality standards for safety, performance and reliability. Its data engineering products and services also provide clients with the information they need to manage their important assets, helping them to achieve better business performance.

CSW was founded in 1998 by a multidisciplinary team, including PhDs in specialised fields of information technology. CSW competitive strengths and advantages lie in three major factors: software quality, technology innovation, and global vision. It supports customers across diverse markets, including Telecom, the public sector, Energy, Finance, Industry, Aerospace, transportation, and defence.

CSW's core competencies cover a wide range of areas, including system planning and analysis, system design, verification, development, integration and maintenance. The company has been operating in mature markets since 1998, with

NASA as its first client, and has offices in Portugal, the United Kingdom, Germany and USA. The Energy sector is one of the sectors where Critical Software has been excelling by providing business critical systems development services to several leading companies in their sectors of activity and in markets with high maturity in the Energy sector such as United Kingdom, Switzerland, Italy and Denmark.

CSW has a division called Smart Technology Solutions (STS) which is responsible for a project that is embedded within the energy sector. This is where the term "UK SMIP" comes in. Smart Metering Implementation Programme (SMIP), "is an energy-industry led programme which aims to roll-out approximately 53 million smart electricity and gas meters to domestic properties and non-domestic sites in Great Britain. The aim of SMIP is to provide consumers with “real time” information on their energy consumption" [3]. There are two types of smart meters: first and second generation, which are referred to as SMETS1 and SMETS2 smart meters, respectively. SMETS stands for Smart Metering Equipment Technical Specification. "When energy companies began installing smart meters several years ago, they fitted first generation (SMETS1) smart meters" [4].

Critical Software is currently working with an entity called Data Communications Company (DCC), which is the "digital spine of the energy system" [2] and is responsible for managing the smart-meters communication. CSW provides software development, testing, and products to the UK SMIP since early 2014, and has since then contributed some major software components for this infrastructure. A large infrastructure like this obviously needs an enormous amount of maintenance. A system with so many users and such an impact on their lives must always be operational. In this project, software maintenance is essential since any modifications that need to be made must be made quickly and without affecting the user.

This work will focus in one of the major systems developed by CSW for the UK SMIP. Due to the sensitivity of this system in the UK SMIP, it was decided to refer to it throughout the document as CSWSYS. Therefore, for the purposes of this work, the CSWSYS is the software system developed and maintained by CSW for the UK SMIP and **CSWSYS component** the component which is part of CSWSYS and the focus of the work.

1.2 Objectives

The primary objective of this internship is to migrate the CSWSYS component from Thorntail to Quarkus following the software maintenance process as this is a maintenance task. Thorntail has reached its end-of-life, and Quarkus has been chosen as the alternative technology for its replacement.

Given that the CSWSYS component uses Thorntail as it application server, it is crucial to maintain all the current requirements after the migration. Therefore, it is essential for the CSWSYS component to retain its capability to receive and validate service requests. It is the component responsible within the CSWSYS to

perform this task.

By the end of the internship, the CSWSYS component should be successfully implemented using Quarkus, and both unit and system tests should be conducted to ensure proper functionality.

In summary, this internship aims to achieve the following objectives:

- Understand the significance of software maintenance in the software life cycle and study the process involved in maintaining software. Identify the different types of maintenance and the techniques employed;
- Implement all the necessary modifications in Quarkus, adhering to the software maintenance process;
- Validate and test the migrated component to ensure its functionality is free from faults and meets the already defined requirements.

1.3 Structure of the document

This document is structured as follows:

- **Chapter 1** contains the detailed objectives of the internship, an overview of the work and the company;
- **Chapter 2** contains the study of software maintenance and the analysis of alternative technologies to proceed in the migration of the component;
- **Chapter 3** contains the existing project requirements, the existing architecture and the approach to implement Quarkus;
- **Chapter 4** contains the developed work, approaches and challenges in order to complete the migration of the CSWSYS component;
- **Chapter 5** contains the conclusions, summarises the developed work, the main limitations and, finally, future work.

Chapter 2

Practices in Software Maintenance

Before diving into analysis of the possible solutions, it is important to understand the importance of software maintenance in the software life cycle and some concepts within the software maintenance world.

This chapter reviews the fundamental concepts of the software maintenance. Section 2.1 introduces the processes involved in the software life cycle. Section 2.2 provides detailed information about the software maintenance phase, including a description of its phases, the various types of software maintenance that exist, and some software maintenance techniques. Section 2.3 addresses the main software maintenance measurements. How maintenance is handled in agile projects is analyzed in section 2.4. In section 2.5, the importance of maintaining a technology stack is highlighted. Furthermore, a connection is made between the significance of maintaining the tech stack and the current CSWSYS's tech stack. This connection addresses the problem and rationale behind the necessity of migrating to Quarkus.

2.1 Software Life Cycle Processes

Developing software is not just about starting to write code, it needs to follow a set of structured processes that enable the production of high-quality, low-cost software in the shortest possible production time. There is a set of technical processes defined in ISO 12207:2017 (which is a comprehensive framework for organisations to implement software projects in a professional and well-planned way), that cover all steps of a software life cycle: starting with business or mission analysis, moving on to the system/software requirements definition, defining the architecture, implementing and integrating the solution, validating the solution, and putting it into operation. Once in operation, the product is in the maintenance phase until it is disposed [7]. In addition to these, there are other processes referenced in the ISO/IEC/ IEEE 12207:2017.

The next section will present the maintenance processes in the context of software development on which this work focuses.

2.2 Software Maintenance

The software life cycle does not end when the product is delivered to the customer. The software product requires significant change and improvement. Once in operation, defects are uncovered, operating environments change, and it is necessary to maintain the product to keep it updated and bug-free.

Software maintenance is an integral part of the software life cycle and is not less important than software development. It is not just about fixing bugs; it involves keeping the software secure and scalable. The maintenance process ends when the software product is finally retired. It is important to understand how software maintenance is handled, starting by identifying the processes.

2.2.1 Maintenance Processes

In the context of software engineering, as mentioned in **Software Life Cycle Processes** (2.1), software maintenance is essentially one of the many technical processes. The maintenance process contains the activities and tasks necessary to modify a software product while keeping its integrity.

According to ISO/IEC/IEEE 14764, which is a document that provides guidance for the maintenance of software, based on the maintenance process defined in ISO/IEC/ IEEE 12207:2017, the maintenance life cycle begins with **Process Implementation** (2.2.1.1) where a planning for maintenance is done and ends with the **Retirement** (2.2.1.6) of the product. As mentioned above, the objective of this process is to modify an existing product (already delivered to the client) in order to make enhancements while preserving its integrity. The maintenance process, as represented in the Figure 2.1, is composed of the following processes [6]:

1. Process Implementation;
2. Problem and Modification Analysis;
3. Modification Implementation;
4. Maintenance Review/Acceptance;
5. Migration;
6. Retirement.

2.2.1.1 1 - Process Implementation

The process of maintaining software starts with process implementation. The maintenance plan and procedures must be developed during this phase in order to be put into action during the maintenance process. The strategy to be used to maintain the system should be documented in the maintenance plan, while the

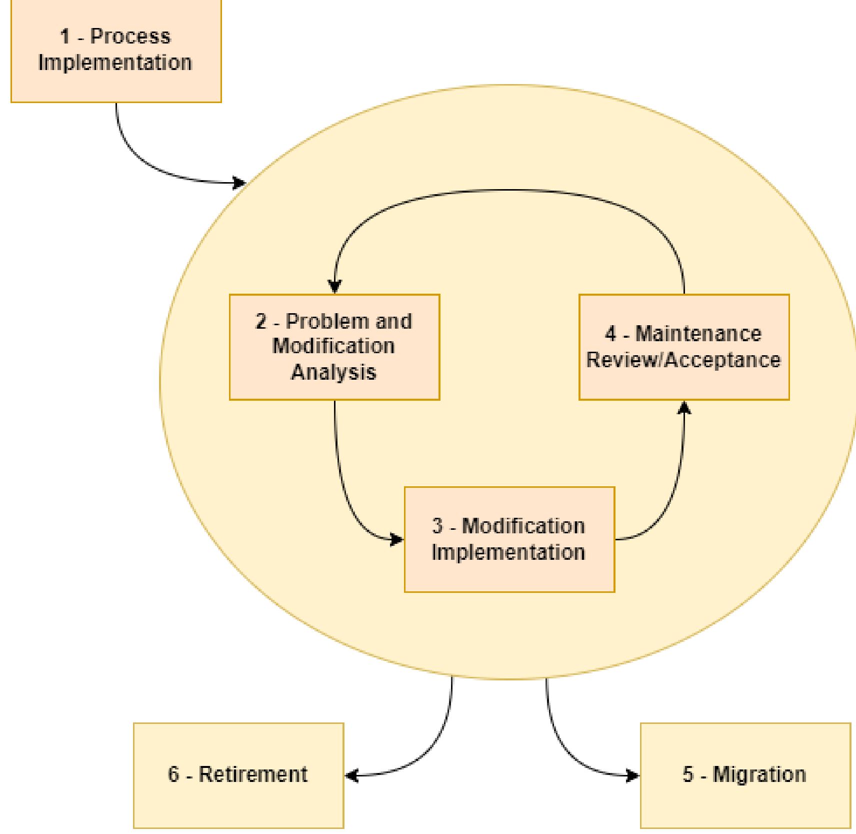


Figure 2.1: Maintenance Process (from - [6])

maintenance procedures should give a more specific approach on how to accomplish the maintenance [6]. In general, this phase is more focused on planning the maintenance to be done on the system.

2.2.1.2 2 - Problem and Modification Analysis

As can be seen in Figure 2.1, this phase and the following ones (2.2.1.3 and 2.2.1.4), are called iteratively when a modification request arises [6]. This phase involves the analysis and identification of problems and modifications requests. In order to assess the potential impact of any proposed changes, an analysis of the impact of the changes is also performed. This phase also includes classifying the type of maintenance to be performed, which will be discussed in more detail in the next sub-section.

2.2.1.3 3 - Modification Implementation

This phase involves developing and implementing the modifications to the software product [6]. In the first instance, analysis is performed to identify the elements to be modified (and document this information), as well as the areas that will be impacted by the modifications and the documentation that needs to be updated. In a second instance, the implementation of the identified modifications begins. It is also necessary to ensure that the original requirements are not

affected by the changes.

2.2.1.4 4 - Maintenance Review/Acceptance

During the maintenance review/acceptance phase, the primary objective is to ensure the accuracy of system modifications and their adherence to authorized standards, employing the appropriate methodology [6]. This phase involves a comprehensive examination of software changes in order to verify their compliance with the requirements and specifications set forth during the analysis and implementation phase, while also ensuring that they do not negatively impact the overall functionality of the software.

2.2.1.5 5 - Migration

During the software life cycle, it is sometimes required to make changes in order to function in a new environment. This phase involves moving the software to a new environment, and this requires performing some tasks such as, planning the migration, through a migration plan. The migration plan covers the requirements analysis, the migration tools, migration execution and migration verification [6]. This phase is important as it helps solve problems such as changing business requirements, compatibility issues, the end-of-life of the old environment, or taking advantage of the new features and capabilities of the new environment.

2.2.1.6 6 - Retirement

When a software product has reached the end of its useful life, it must initiate the retirement process [6]. To determine whether it is beneficial to retain or retire the outdated technology and whether a replacement software should be adopted, an analysis should be conducted. The retirement of software necessitates careful planning through the development of a retirement plan that evaluates the retired requirements, the impact of retiring the software, identifies potential replacements (if applicable), and outlines all essential actions required to execute the retirement successfully.

2.2.2 Types of Software Maintenance

Maintenance is necessary to make sure that the software product continues to satisfy user requirements, but this process may consume a considerable amount of life cycle costs. The analysis of the type of maintenance to be performed helps to understand the costs. There are five types of maintenance [6] that can be performed, and will be detailed bellow:

- **Preventive Maintenance**
  Preventive Maintenance is the modification of a software product after delivery to detect and correct latent faults in the software product before they

become operational faults [5]. In general, preventive maintenance encompasses any type of intention-based activity that allows to forecast upcoming problems and prevent maintenance problems before they occur [10], such as code reviews and testing. Preventive Maintenance is undertaken to improve maintainability of the software [14] and is about Software reegineering (e.g. Data restructuring, Code restructuring);

- **Corrective maintenance**
  Corrective maintenance refers to modifications needed by actual errors in a software product. It is a reactive modification of a software product performed after delivery to correct discovered problems [6]. It involves fixing errors or bugs as quickly as possible to restore normal operation of the software. Corrective maintenance is an important part of the software development process, as it helps to ensure that the software is functioning correctly and is in accordance with requirements. It gathers a set of tasks such as: understanding the system, evaluate the problem, repair code and testing;

- **Adaptive Maintenance**
  Refers to the modification of a software product performed after delivery to keep a software product usable in a changed or changing environment [5]. This type of maintenance performs a set of tasks such as: understanding the system, defining the adaption requirements, developing preliminary and detailed adaption design, code changes and testing [8]. Adaptive maintenance is an important part of the software life cycle as it allows for the system to remain operational when the platforms on which it is running change. This is achieved by integrating the software systems to new platforms, environments, languages and third-party applications;

- **Perfective Maintenance**
  Refers to the modification of a software product after delivery to provide enhancements for users, improvement of programme documentation, and recoding to improve software performance, maintainability, or other software attributes [5]. It aims to apply changes in the system in order to increase some of its functional and non-functional quality characteristics [10]. This type of maintenance includes understanding the system, diagnosing and defining requirements for improvements, and, finally, testing [8]. These changes (adding or deleting features) may arise as a result of new problems and new ideas and keep the software in line with the users needs as well as the constant growth of the technology.

- **Emergency Maintenance**
  It is an unscheduled modification performed to temporarily keep a software product operational pending corrective maintenance [6]. This type of maintenance is usually not planned and for that reason is inefficient and expensive. It is typically performed under time pressure, as the goal is to fix the problem as quickly as possible to minimize the impact on users.

Figure 2.2 represents the types of maintenance and when to use them.

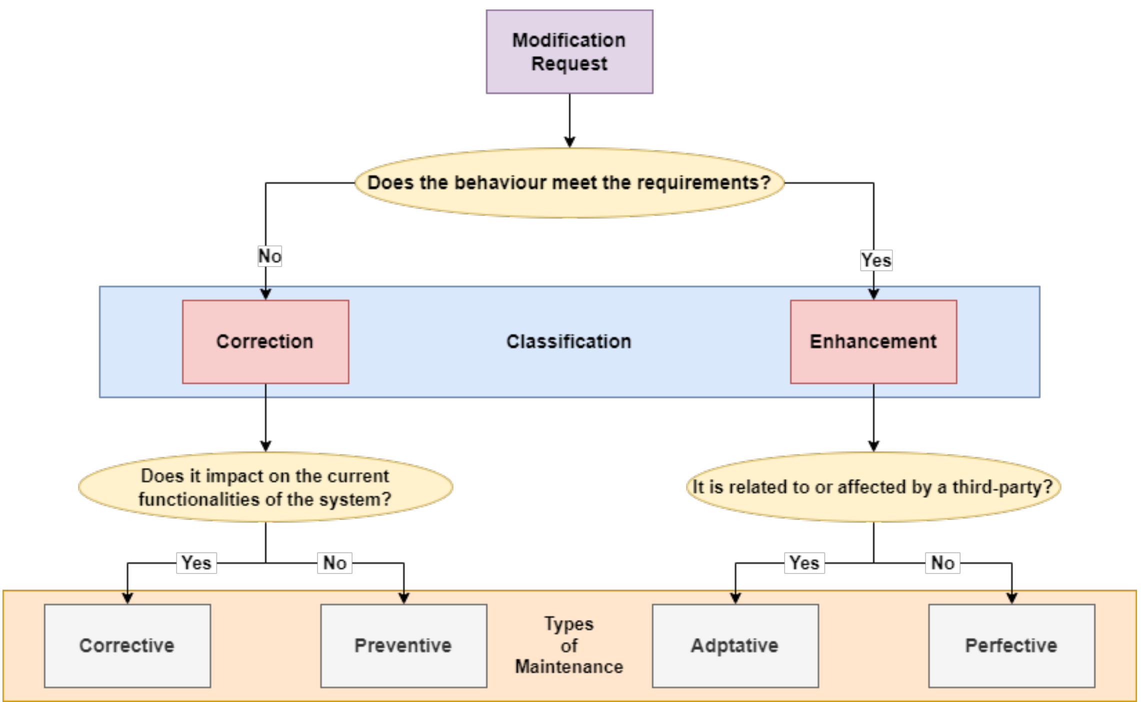


Figure 2.2: Modification Request (adapted from [9])

In general, each type of software maintenance has its own goals and approaches. When a modification request arrives, if the software behaviour does not meet the requirements, the request is classified as a software correction and identified as corrective or preventive maintenance. Corrective maintenance involves fixing errors or bugs in the software. Preventive maintenance involves performing activities to prevent future problems with the software.

On the other hand, if the software behaviour meet the specifications, when the modification request comes in, the request is classified as an enhancement and identified as adaptive or preventive maintenance. Adaptive maintenance involves making changes to the software to ensure it continues to function correctly in a changed environment. Perfective maintenance involves making improvements to the software to enhance its performance or functionality. Emergency maintenance is performed on an urgent basis in response to a critical problem or failure in the software.

2.2.3 Techniques for Maintenance

The types of software maintenance that can be performed were covered in the previous sub-section: Corrective (defects), Preventive (to make future maintenance tasks easier), Adaptive (changes in environment) and Perfective (accommodate new features or delete). However, before beginning any change, it is necessary to comprehend the product as a whole as well as the impact of the change.

The following techniques help analyse and understand the behaviour of the software product:

- **Program Comprehension**
  Analysing old software systems has become an important task in software maintenance. Legacy systems have been subject to extensive analysis in order to make improvements and modifications. If these systems are not changed, they will become unusable. So the first step is to understand the software by using **program comprehension**. Programmers spend considerable time reading and understanding programmes in order to implement changes. Code browsers are key tools for programme comprehension and are used to organise and present source code [5]. If the system does not have good documentation, it can be an obstacle to performing this task, consuming much more time than necessary.

- **Reengineering**
  Reengineering is defined as the examination and alteration of software to reconstitute it in a new form, which includes the subsequent implementation of the new form. It is often undertaken not to improve maintainability but to replace ageing legacy software [5]. This can range from fully automatic approaches to manual reimplementations, including restructuring techniques, formal transformations, injecting component technologies, and replacing old user interfaces or database technologies [11]. Moving from an old legacy system to a new one is called software reengineering.

- **Reverse Engineering**
  Reverse engineering is the process of analysing software to identify the software's components and their inter-relationships and to create representations of the software in another form or at higher levels of abstraction [5]. It is a process of recovering the design, requirements and functions of a product from an analysis of its code [12]. Reconstructs concepts such as system architecture and business rules from the source code and from the documentation. This process does not modify the software or create a new one. Considering the time devoted to program understanding, this process offers real scope for reducing maintenance costs. The understanding obtained through reverse engineering can support the implementation of change through techniques such as forward engineering, restructuring, and reengineering [14].

In the sub-section 2.2.2, we covered different types of maintenance, and each type has its own approach. But there is an activity that is common to all, which is program comprehension. Before starting any modification to the system, using corrective, adaptive, preventive, perfective, and emergency maintenance, it is crucial to understand the software.

Reverse engineering is also a useful technique in a corrective change because it makes identifying defective program components easier. Through data-flow and control flow diagrams, charts can help identify and trace the required changes. For adaptive and perfective changes, the reverse engineering technique presents a vision that facilitates the understanding of the system components, making it possible to understand where the new requirements fit and how they relate to the existing components [14]. In the case of preventive maintenance, the graphical

representations that reverse engineering provide help with future maintenance. Reengineering also prevents foreseen maintenance problems.

2.3 Software Maintenance Measurement

The purpose of measurement is to collect, analyze and report information in order to support a good management and to demonstrate the quality of the software product. It is the process of using metrics to track and evaluate the effectiveness and efficiency of the software maintenance process.

There are several reasons to measure software. Software measurement allows for the evaluation of the use of different tools, libraries, and methods so that when the time comes to make a decision, the choice is the one that fits best. Through the measurement, the maintainer can also determine whether or not the maintenance is achieving the goal. If the goals are not met, then corrective actions can be taken. Sometimes there is room to improve the characteristics of the software, but without objective measures, it is difficult to assess such improvements.

There are several measures that maintainers may need to take. These measures can be derived from the attributes of the software, the maintenance process, and personnel. However, the source code is the most commonly used source of measures because it is sometimes the only information about the system available. There may be documentation available, but it may be out of date or inaccurate [14]. Therefore it is possible to take some measures from source code such as:

- Size;
- Complexity;
- Quality;
- Understandability;
- Maintainability.

2.3.1 Size

As mentioned before, the source code is usually the main source of measurements, so one of the commonest ways of measuring the size of a program is by counting the number of lines of code [14]. The focus is on the number of lines that have been added or modified during the maintenance process. This process makes it easier to determine the effort associated with the changes made.

2.3.2 Complexity

One of the major problems that software maintainers face is dealing with the increasing complexity of the software source code that they have to modify. A system may be complex because of poor programming practices or maintainability concerns were not taken into account during development [15]. "The more complex a program is, the more likely it is for the maintainer to make an error" [14]. As the system's complexity grows, it can become increasingly challenging to maintain a complete understanding of it. Complexity may have a direct impact upon maintenance costs. The more difficult a highly complex application system is to maintain, the more billable hours it will accumulate in the course of the maintenance and enhancements which consume most of a system's life cycle costs [15].

There are two types of complexity metrics: Cyclomatic complexity and Cognitive complexity. Cyclomatic complexity is based the code structure and on the execution flow (number of linearly independent paths). A function with a cyclomatic complexity below 10 can be considered simple and testable, while a cyclomatic complexity greater than 20 suggests an overly complex function [80]. On the other hand, cognitive complexity focuses on human understandability. It takes into account both the code structure and cognitive factors, such as recursive calls. This metric indicates how easy or difficult it is for a human being to comprehend the purpose and behavior of a piece of code, aligning with the concept of understandability (sub-section 2.3.4). A cognitive complexity of 15 is a good metric for a understandable code [80]. SonarQube [78] is a tool that allows to measure these complexity indexes.

2.3.3 Quality

Software quality is an important consideration in the maintenance of a software product [6]. It is possible to evaluate the software quality in two ways: Product quality and Process Quality.

Product quality is measured by the number of change requests received after the system has been put into operation. These requests can indicate the degree of user satisfaction. Another way to measure the product's quality is by the number of faults detected after the software system becomes operational. The process quality can be tracked by two variables: schedule and productivity. Firstly, the schedule is obtained through the planned and actual work time, and the productivity is obtained by the number of lines of code that have been modified or added by the effort required to make the modification (from analysing the modification to the implementation) [14].

2.3.4 Understandability

"Understandability is the concept that a system should be presented so that an engineer can easily comprehend it" [16]. Program understandability is the ease

with which the program can be understood, that is, the ability to determine what a program does and how it works by reading its source code and accompanying documentation [14]. As the complexity of a program increases, its understandability tends to decrease. Understandability is a crucial factor in software engineering. The ease of understanding does not depend only on the source code but also on the available documentation, maintenance processes, and maintenance personnel.

A system can be understood if it is: Complete - the system must contain a set of sources that contain all the key information; Concise - The source code should not have an excessive amount of detail, so the developer can focus on the task; Clear - Consistency and good practices should be used. Having clear documentation makes the process of understanding the application easier, saving extra effort. Organised - The software documentation and source code should be easily located in order to facilitate the cross-referenced information [17].

Setting a code style can be useful as it improves the readability of the code by using a consistent style, making it easier for developers to understand and maintain the code. A well-defined code style often includes best practices and guidelines for writing clean, efficient, and maintainable code. By adhering to the code style, developers are more likely to follow these best practices, resulting in higher code quality.

2.3.5 Maintainability

Maintainability is defined as the capability of the software product to be modified [13]. The maintainability of a system may be used as an indicator of its quality. According to ISO 9126, there are four maintainability sub-characteristics:

- **Analysability:** The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified [13];

- **Changeability:** The capability of the software product to enable a specified modification to be implemented [13];

- **Stability:** The capability of the software product to avoid unexpected effects from modifications of the software [13];

- **Testability:** The capability of the software product to enable modified software to be validated [13].

Maintainability can also be measured by Mean Time To Restore (MTTR), and this includes both repair time and testing time. By improving the effectiveness of the repair teams and processes, the objective is to reduce this number as much as possible.

2.4 Handling Maintenance in Agile Projects

Agile methodologies have been gaining popularity over the years, "due to the fact of producing high-quality software systems" [20], and have been adopted to deal with uncertainty in software development projects. The main advantage of using Agile methodologies is not just the fast delivery but also the flexibility to accept future changes during the life of software development and the active participation of the customer throughout the development lifecycle. Many agile methodologies have been developed so far, which help in developing and maintaining the software at a lower cost [18]. The agile software life cycle does not have a specific moment for maintenance, and as presented previously, software maintenance is a vital phase of the software life cycle.

Agile methods have in mind software development in general, not necessarily software maintenance. Therefore, the question that arises is where does Agile fit in the world of software maintenance projects. Maintenance projects, by their nature, are event-driven, which means they are reactive to events such as when a failure in the system occurs, when a change is requested by a customer... It is difficult to plan the work for the life of a software product in production. The Agile method of choice for software maintenance projects has to be picked or tailored keeping in mind these specific characteristics of maintenance projects.

Agile methods include common characteristics such as an interactive development process, small teams working closely together, light documentation, close customer involvement, frequent testing, knowledge transfer through openness, and a focus on a high quality of code and product [19]. Studies have highlighted that Agile methods can help speed up the process and improve code quality, team motivation, and communication between members of the maintenance team, and an interactive lifecycle can be seen as advantageous for maintenance due to the short-term nature of the work [19]. A common characteristic is light documentation, and maintenance engineers do not like less documentation because they rely on it to understand the system [20]. Some agile methodologies will be presented in the following sub-sections in order to determine which are best suited for software maintenance.

2.4.1 Scrum

Scrum is an agile framework for managing and completing complex projects. This agile framework helps teams manage and structure their work through a set of practices and principles. It is based on three pillars: transparency, inspection, and adaptation [23].

The Scrum framework includes specific roles, events, and artifacts that help to align the team and guide the process. The key roles in Scrum are the Product Owner, Scrum Master, and Development Team [23]. The Product Owner is the member of the Scrum Team who ensures that the group produces the most valuable product possible [23]. The Scrum Master is responsible for ensuring the team is following the Scrum process and for helping the team and organisation

be as effective as they can be. Finally, the development team is responsible for completing the work and creating the product.

The events that constitute framework scrum:

- **Sprint** - Sprint is a time-boxed short cycle which contains all the other scrum events. A new sprint starts immediately after the conclusion of the previous sprint [23];
- **Sprint Planning** - Event where the team meets to plan the work for the upcoming sprint [23];
- **Daily Scrum** - A short stand-up meeting where the team reviews progress toward the sprint Goal [23];
- **Sprint Review** - Event held at the end of the sprint where the scrum team demonstrates the work completed during the sprint to stakeholders and what has changed in their environment [23];
- **Sprint Retrospective** - The Scrum Team gets together during this event to reflect on the last sprint and identify areas for improvement [23].

Scrum artifacts includes:

- **Product Backlog** - A prioritized list of what is needed to improve the product;
- **Sprint Backlog** - A highly visible list of work that is the Developer's plan for the Sprint;
- **Increments** - "Small pieces of work that serve as concrete stepping stones toward the Product Goal" [23].

Table 2.1: Scrum - Pros and Cons (adapted from[24])

Pros	Cons
Teams can accomplish project deliverables more rapidly and effectively with the help of Scrum.	Scrum often leads to scope creep, due to the lack of a definite end-date
Short sprints enable changes based on feedback a lot more easily.	Scrum requires experienced and high-level individuals.
The individual effort of each team member is visible during daily scrum meetings.	Team members can become frustrated by daily meetings.
Scrum ensures efficient time and money management.	It is challenging to implement the Scrum framework with large teams.
The team gets clear visibility through scrum meetings.	Implementing quality is challenging until the team goes through an aggressive testing process.

It is important to note that these are general pros and cons of Scrum, as with any framework it is success relies on the team and organization culture and its ability to adapt the framework to their own needs.

2.4.2 Kanban

As Scrum, Kanban is also an agile framework. Real-time capacity communication and complete work transparency are required with this framework [26]. Kanban is based on principles of visualising the flow of the work and limiting the work-in-progress (WIP).

The key elements of Kanban are:

- **Kanban Board** - A Kanban board is a tool designed to help visualize work, limit work-in-progress, and improve the efficiency and flow of the work [27];
- **Work in progress (WIP) limits** - Limits set the maximum amount of work that can exist in each status of a workflow. This helps to control and balance the flow of work and identify Bottlenecks [28];
- **Pull-based system** - Work is pulled through the process as capacity becomes available.

Although Kanban itself does not prescribe specific meetings, there are several types of meetings commonly used in Kanban in order to facilitate improvement: Daily Stand-up, which is a short daily meeting where team discuss progress, challenges and plans for the day; Replenishment Meeting, which focuses on replenishing the backlog of work; Retrospective Meeting, which is a reflective session held after a specific period where the team discusses what went well, what did not go well, and ideas for improvement. Also aims to identify lessons learned, celebrate sucesses and define actions/experiments in order to enhance team performance; Delivery Planning which focuses on understanding requirements, expectations, and coordinating efforts.

Table 2.2 will present the advantages and disadvantages of Kanban.

Table 2.2: Kanban - Pros and Cons (adapted from[29])

Pros	Cons
Kanban boards are easy to visualise and keep everyone on the same page..	There are no timing parameters, kanban boards have no dates (no timeframes are associated with each phase).
Limits the amount of work in progress (WIP) in order to ensure quality, when the columns start getting too long, indicate a disconnect between resources.	Kanban Boards can become very complex and confusing, and if not updated may not reflect reality.
Flexible, so the work to be done can be rearranged and reprioritized .	Teams struggle to focus on priorities since there are no clearly defined, tight responsibilities.

2.4.3 Differences between Scrum and Kanban

Both Scrum and Kanban belong to Agile and Lean methods, where the focus is to quickly respond to customer requests. Both are highly adaptive and based on highly collaborative and self-managing teams [33]. They are not completely incompatible, there is even a framework called Scrumban that takes the best of each and merges them. Scrumban will be discussed separately in the next subsection. Table 2.3 present the main differences between both agile frameworks.

Table 2.3: Scrum vs Kanban (adapted from [30])

	Kanban	Scrum
Roles	There are no pre-defined roles for a team.	Each team member has a predefined role (Scrum Master, Product Owner, Team Member)
Due Dates	Continuos delivery.	Deliverables are determined by sprints.
Delegation & Prioritization	Uses a “pull system,”that allows team members to only “pull” new tasks once the previous task is complete.	Uses a “pull system” however an entire batch is pulled for each iteration.
Change policy	Allows for changes to be made during the project,allowing for iterations and continuous improvement prior to the completion of a project.	Changes during the sprint are strongly discouraged.
Best Application	Best for projects that are reactive to events and do not have a consistent or planned workload.	Best for teams with stable priorities that may not change as much over time.

In summary, Scrum is a framework that is focused on time-boxed iterations and delivering working software, while Kanban is focused on visualizing the flow of work, managing and improving processes. Both frameworks can be effective in their own right and they can also be combined in a hybrid approach called Scrumban.

2.4.4 Scrumban

Scrumban is a definition of Scrum + Kanban [31] and as mentioned before, is an agile integration of Scrum and Kanban. Merges the structure and predictability of Scrum with Kanban’s flexibility and continuous workflow. Scrumban can help a team benefit from the prescriptive nature of Scrum and the freedom of Kanban to improve their processes [25].

In Scrumban, the teams use the time-boxed sprints of Scrum to plan and prioritise their work and the Kanban board to visualise and manage the flow of work. The teamwork is organised in small iterations and tracked with the help of a visual board. When it becomes necessary to decide which user stories and tasks should be completed in the next iteration, on-demand planning sessions are conducted. The Kanban work-in-progress (WIP) limit is intended to keep iterations

short [25].

Table 2.4: Scrumban Pros and Cons (adapted from [32])

Pros	Cons
Bottlenecks can be identified and reduced by limiting the product backlog and each stage of the workflow.	Team managers have less control, in Scrumban there are not specific roles on development team. This approach can cause some confusion and disorganization.
Enforces transparency, by taking advantage of the visual features of kanban (same board). Each stage is visible not only to the development team but to all stakeholders.	It is relatively new, and because of that, there aren't many established processes, and the documentation is not very extensive.
Everyone can see what work is being done, which helps to save time and prevents the duplicated work.	It is difficult to monitor each element's effort and contribution because each member selects their own tasks, and there are not mandatory daily meetings to track.
Continuos Workflow, by setting limits in work-in-progress (WIP). Also no daily stand-ups reduce the levels of stress and frustration.	

As mentioned before, maintenance projects are event-driven, and there is an exigency to react to a change request or to fix bugs that are unpredictable. Scrumban and Kanban enters the picture at this point. When it comes to maintenance, there are some details in the different methodologies that may be better suited to the unpredictable nature of changes. While Scrum is better suited when work is prioritised in batches and priorities do not change in less than one sprint duration, Kanban is more suitable for work where there is a large degree of variability in priority. Due to the nature of maintenance tasks, the concept of working on a continuous delivery model like Kanban works better compared with sprint delivery [33]. Scrum's inflexibility during sprints and inability to adapt to changes make it a poor methodology for software maintenance. On the other hand, Kanban focuses continuous delivery. All incoming tasks are listed on the Kanban board (considering the WIP limits) and can be assigned a priority based on their importance and urgency [33]. As the scrumban also has the flexibility of kanban it is also an excellent tool for managing ongoing project maintenance chores.

In the Critical Software (CSW) project, initially scrumban was used because there was no product owner and there were still sprints, but recently the Kanban methodology was adopted, and as we have seen, it is the most appropriate methodology for software maintenance.

2.5 Technology Stack

The last section examined how agile projects handle software maintenance, and technology stacks often play a role in determining the tools and processes that teams use to implement agile methodologies. Agile methodologies prioritize flexibility and adaptability, and the choice of technology stack can have a significant impact on a team's ability to meet these goals. Firstly, it is essential to define the technology stack concept.

A technology stack (sometimes also called Tech Stack) is a set of technologies used to build and run an application [35]. Typically this comprises of programming languages, frameworks, a database, front-end tools, back-end tools, and applications connected via APIs.

The technology stack for an agile project needs to be carefully chosen to ensure that it is well-suited to the needs of the project. The difficulty of maintaining and scaling the application and how the application will behave now and in the future depends on the chosen tech stack [34].

2.5.1 Why it is important to maintain a Tech Stack?

The technology stack of a project is one of the main elements that needs to be constantly maintained because the smooth running of the project depends on it. Software maintenance and the tech stack are related in that the maintenance of a software system or application can be impacted by the technologies in the stack.

In terms of compatibility, if the software uses technologies that are no longer supported or are incompatible with other technologies in the stack, it can be more difficult to maintain. On the other hand, the software can also be affected in terms of performance, since using outdated or inefficient technologies can make the software slower or use more resources than it really needs. Using outdated technologies can also impact the security of a software system, since these technologies can make the software more vulnerable to attacks. Continuously maintaining the tech stack and updating to newer and more secure technologies helps protect the software from these threats.

Regularly maintaining both software and technology stack is important for ensuring that the software is stable, efficient, and secure. In the next section 2.5.2, will be presented the tech stack that is currently being used within the project.

2.5.2 Technology Stack of CSWSYS

The tech stack that is currently being used is:

- Thorntail as building framework;
- MySQL as database language;

- Java 8 as the development Language;
- Docker;
- Maven 3.6.

Thorntail will be the technology covered in the following sub-section since it is the target of the work. Firstly, a description of Thorntail will be made, followed by a description of the problem that this technology has and, finally, possible solutions.

2.5.3 Thorntail Background

Thorntail is a MicroProfile certified framework, designed to facilitate the development of cloud-native applications by leveraging the power of Enterprise Java components. Based on WildFly Java application server, providing a reliable foundation for the creation of small, stand-alone microservice-based applications [36].

Thorntail was the application upon which the whole CSWSYS business-level code was deployed. The historic reason for selecting Thorntail in 2018 was driven by the lack of available Java EE-compatible application servers that offered a single-step deployment setup, which is particularly attractive for a containerized orchestration platform.

The Thorntail project has reached its End of life (EOL), May 31, 2021 [37]. And as we saw in section 2.5.1, using legacy software can bring problems and vulnerabilities to the system. Therefore, it is necessary to maintain the tech stack updated and replace Thorntail with a long term support (LTS) technology, and that technology needs to have the same or better performance than Thorntail. Since Thorntail is the focus of the study, the other technologies in the CSWSYS tech stack will not be addressed.

2.5.4 Thorntail Alternatives

In this sub-section, different alternatives will be discussed in order to replace Thorntail that follows to the same standards as CSW.

2.5.4.1 Quarkus

An alternative to Thorntail suggested by RedHat was Quarkus. Quarkus is a full-stack, Kubernetes-native Java framework made for Java virtual machines (JVMs) and native compilation, optimizing Java specifically for containers and enabling it to become an effective platform for serverless, cloud, and Kubernetes environments [38]. Designed to work with HotSpot VM and GraalVM (a universal virtual machine for running applications written in a number of languages, including Java and JavaScript), for native compilation of the application.

The main feature is the use of extensions, which work like regular Maven dependencies. These include the dependency itself along with the appropriate configuration files that are needed for use in native images. Quarkus applications are optimised for low memory usage and fast startup times, because this framework was built around a container-first philosophy.

GraalVM is a polyglot virtual machine (VM) capable of compiling Java code (and many other hosted languages) into Java bytecode that is quicker and more efficient than the regular JVM thanks to Just-in-time (JIT) compilation and a set of optimizations. An important feature of GraalVM is the ability to compile code into lightweight Native Images. These images are generally much quicker to start up due to being extremely optimized. Native images also have a lower memory footprint than traditional images, due to an in depth analysis of the code at compile time that removes unused code and the compilation of the code that will effectively be required.

To achieve the fast startup times and low memory usages, the applications are optimised in the following ways:

- **Build Time Processing**
  Quarkus operates on the principle of performing extensive processing during build-time, akin to what traditional frameworks do during runtime. This includes tasks such as configuration parsing, classpath scanning, and feature toggling based on classloading [40]. By conducting thorough processing at build-time, only the necessary classes are included at runtime, eliminating the loading of unused classes into the production JVM. During build-time, Quarkus prepares the initialization of all application components, processes metadata, reduces memory usage, and facilitates faster startup times [40];

- **Reduction in Reflection Usage**
  Reflection is a built-in feature in the Java programming language that enables a running Java program to inspect and manipulate its own internal properties, commonly referred to as "introspection" [41]. Quarkus tries as much as possible to avoid reflection. By reducing reliance on reflection operations, Quarkus achieves shorter boot times and reduced memory usage, enhancing overall efficiency and performance [40];

- **First-Class Support for GraalVM Native Images**
  An important feature of GraalVM is the ability to compile code into lightweight Native executable. When this happens, the boot time is much faster and can run with a much smaller heap than a standard JVM. Also, the native compiler uses aggressive dead-code elimination techniques to only embed the parts of the JVM and classes that are absolutely required by your application [40];

- **Native Image Pre-Boot**
  The maximum frameworks are pre-booted during native compilation. This way, the native executable has already run most of the startup code and serialized the result into the executable, providing faster startup [40].

Table 2.5: Quarkus - Pros and Cons (adapted from [42],[43])

Pros	**Cons**
Good and simple documentation on web. Almost every solution can be found in its community pages.	The installation of the Graal VM can be challenging.
Quarkus significantly reduces the application boot time as compared to alternative frameworks. The reasons for this are the build-time metadata processing and the use of GraalVM or Substrate VM to create standalone native images.	Lack of Solution on Forums: The Quarkus community forum does not always have the answer to every problem because Quarkus is relatively new.
Quarkus provides fast since it can automatically detect changes made to Java and other resource/configuration files, and transparently re-compile a nd deploy the changes.	Highly-optimized build process for native images takes a long time.
JAX-RS is used as a foundation, which is a well known enterprise standard	

Developing Java applications with Quarkus offers a significant performance boost as its main advantage. However, the drawbacks arise from the optimizations executed during the compilation process and due to it being a more recent framework.

2.5.4.2 Spring Boot

Spring Boot is a popular open-source Java framework designed to simplify the development of enterprise applications and microservices [42], supporting Java, Kotlin and Groovy and JIT compilation. It is used to build production-ready applications with features such as: starters and auto-configurations. Spring Boot provides an opinionated approach to building Java applications, which helps to simplify the development process by providing a set of conventions and defaults. It provides an easy way to get started with a new Spring-based application and it provides a simple way to create and run a production-ready application.

Spring Boot has three core capabilities that make developing with the Spring Framework faster and easier:

- **Autoconfiguration**
  One of the advantages of SpringBoot is the automatic configuration of the application. Applications are initialised with pre-set dependencies, which do not need to be configured manually. Spring Boot will attempt to automatically configure the application based on the jar dependencies provided [44];
- **Opinionated approach**

Based on the needs of the project, Spring Boot takes an opinionated approach to adding and setting starter dependencies. The required third-party dependencies are implicitly packaged by Spring Boot and provided as starter packages. [44]. This feature is just a method of managing all required dependencies in one place and efficiently making use of them.

This approach can be beneficial because it helps to simplify the development process by providing a set of conventions and defaults that are known to work well together, which can save time and effort. However, this approach can also be a disadvantage for developers that have specific requirements that do not align with the conventions and defaults provided by Spring Boot;

- **Standalone applications**
  Spring Boot includes and embedded web server. The developers are not required to setup a separate servlet container and deploy an application to it. It is possible create standalone applications that run on their own, without relying on an external web server [45].

Table 2.6: SpringBoot - Pros and Cons (adapted from [44])

Pros	**Cons**
Extensive documentation and community support.	Lack of control - Spring Boot creates a lot of unused dependencies, resulting in a large deployment file.
Reduced amounts of source code.	The complex and time-consuming process of converting a legacy or an existing Spring project to a Spring Boot application. When there is a framework conversion, the integration will be greater.
Eases the dependency and comes with Embedded Servlet Container.	Not suitable for large-scale projects. Spring boot is built focusing on micro-services. Not recommended for building monolithic applications.
No need for XML configuration, Spring boot provides the option to use either XML configurations or annotations. Sometimes, developers choose annotations to avoid waste extra time of writing code.	
Autoconfiguration.	
The ability to create standalone applications.	

2.5.4.3 JBoss/Wildfly

Red Hat's Jboss supports the WildFly open-source application server software (also known as JBoss AS). It offers a comprehensive Java Enterprise Edition (JEE)

stack, making it an excellent choice for developers building enterprise Java applications. JBoss includes various technologies, such as Enterprise JavaBeans (EJB), that are valuable for building robust and scalable applications [49].

JBoss supports Jakarta EE (formerly known as Java EE), a set of specifications for building enterprise-grade Java applications. This compatibility ensures that developers can leverage the latest Jakarta EE technologies and benefit from the extensive ecosystem of Jakarta EE web-based frameworks available. JBoss's compatibility with these frameworks simplifies development and integration processes, providing developers with a wide range of tools and options.

Table 2.7: JBoss - Pros and Cons (from [49])

Pros	Cons
Support for Java EE.	It is slow.
Ease of server administration and configuration management.	Need more flexibility in pricing.
Reduction of DevOps resource requirements and support for rapid scalability.	User interface need more improvement.
It being a Java based product is pretty straightforward.	Needs to support the Open Services Gateway initiative (OSGi).
Easy to manage since you can deploy the applications on the JBoss server.	The solution could improve by providing more integration.
Easy to use from a developer´s perspective.	Initial setup is challenging.

2.5.5 Comparison

In this sub-section some comparisons between main technologies will be made in order to determine the best option. Table 2.8 presents some of the main differences between Quarkus and Spring Boot:

Table 2.8: Quarkus vs Spring Boot Comparison (adapted from [42])

Comparison	Quarkus	Spring Boot
Dependencies	Uses Contexts and Dependency Injection (CDI)	Uses a robust Dependency Injection Container (DI)
Boot time	Faster boot times	Slower boot times than Quarkus
Memory consumption	Lower memory consumption due all optimizations	Higher memory consumption
Maturity	New Framework	More mature
Community Support	The community doesn't always have the solution to the problem	Excellent documentation and community support

Quarkus is a more recent framework than Spring Boot, which brings more innovation. However, the community does not always have all the solutions for the problems. It offers quick startup times in comparison with Spring Boot and a positive development experience, since is similiar with other frameworks such as Thorntail. On the other hand, Spring Boot is a well-known and reliable framework with many features and security measures. Has a huge community support that knows the answer to any question and has excellent documentation. When compared with Quarkus, it has higher memory consumption and slower startup times. The main issue with Quarkus, due to its recentness, is that it has fewer options and a smaller pool of specialised engineers to answer the questions, which results in longer investigation times.

According to the test data [47], Quarkus is often better than Spring Boot, in line with what was previously mentioned. The Figure 2.3 shows the values resulting from the test [47]. "To test both implementations, we'll use Wrk to perform the test, and its metrics report to analyze our findings. Also, we'll use VisualVM to monitor the applications' resource utilization during the test execution" [47]. All tests were performed on a machine with the following specifications:

- **Memory:** 64 Gb;
- **Processor:** AMD Ryzen 9 5900HX with Radeon Graphics x 16;
- **Graphics:** NVDIA GeForce RTX 3080 Laptop GPU / AMD RENOIR;
- **Disk Capacity:** 2.0 Tb;
- **OS Name:** Fedora Linux 36 (Workstation Edition);
- **OS Type:** 64-bit;
- **GNOME Version:** 42.3;
- **Windowing System:** Wayland.

Metrics	Spring Boot JVM	Quarkus JVM	Spring Boot Native	Quarkus Native
Startup Time (s)	1,865	1,274	0,129	0,110
Build Artifact Time (s)	1,759	5,243	113	91
Artifact Size (MB)	30,0	31,8	94,7	80,5
Loaded Classes	8861	8496	21615	16040
CPU usage max (%)	100	100	100	100
CPU usage average (%)	82	73	94	92
Heap Size Startup (MB)	1048,57	1056,96	-	-
Used heap start-up (MB)	83	62	12	58
Used Heap max (MB)	780	782	217	529
Used heap average (MB)	675	534	115	379
RSS Memory startup	494,04	216,1	90,91	71,92
Max threads	77	47	73	42
Requests per second	7887,29	9373,38	5865,02	4932,04

Figure 2.3: Spring Boot Vs Quarkus Metrics (from - [47])

The startup time, as expected, was faster in both cases for Quarkus. Quarkus build time was much quicker in the case of native images, but in the case of JVM,

it was significantly longer than Spring Boot, since Spring Boot makes good use of JVM handling and JIT compilation [77]. Regarding CPU usage, overall, Quarkus obtained better results than Spring Boot, although there was not a big difference. The JVM consumes more memory than the native code in both technologies. In both cases, the number of classes loaded was lower for Quarkus. The ahead-of-time compilation strategy used by Quarkus while building the application gives the benefit of loading only the necessary classes. Also, for both JVM and Native options, the Resident Set Size (RSS) memory is lower for Quarkus.

Regarding JBoss/WildFly, while it provides a comprehensive JEE stack with various features and services, it can sometimes encounter performance problems. This performance concern is mentioned as a disadvantage of using JBoss (Table 2.7). On the other hand, Quarkus, known as one of the fastest technologies for developing Java applications, surpasses JBoss in terms of performance. Quarkus is specifically designed to optimize and improve the performance of Java applications.

2.5.6 Final Thoughts

After the comparisons of the alternatives of Thorntail in sub-section 2.5.5, Quarkus generically performed better than Spring Boot, yet Spring Boot also proved to be a good option to replace Thorntail. Due to the fact that Spring Boot has an opinionated approach, problems can arise when choosing this technology. On the other hand, Quarkus is built on top of known enterprise standards such as CDI, JAX-RS, and many more. Developers do not need to learn new APIs or rewrite the software's code. They can write an application that is based on CDI, JAX-RS, or Java Persistence API (JPA), for example, and optimise it by changing the runtime to Quarkus [50].

In addition to the above analyses, in 2021, CSW conducted a proof of concept (POC) regarding the migration of other components that are part of CSWSYS to Quarkus and to Wildfly and reached the following conclusions: It is possible to migrate to both Application Servers. From a software development perspective, the migration to Quarkus is significantly easier than the migration to Wildfly. Although there are some incompatibilities between Thorntail and Quarkus due to Quarkus's lack of EJBs, these have been demonstrated to be circumvented. The deployment of Wildfly is very different from the current deployment model followed by CSW. Very significant improvements were seen in Quarkus with reference to memory usage and startup times. Also, some improvements were seen in Quarkus with reference to transaction duration.

Critical Software had some reservations regarding Spring Boot, due to their approach regarding Java Enterprise Specifications, which could lead to technology lock-in. Quarkus was the choosen technology to replace Thorntail.

Chapter 3

Project Background and Approach

This chapter start by giving a project background, details the existing project requirements, the existing project architecture and the approach to tackle the objectives defined for this internship. The approach includes the high-level steps defined that summarize the work to be done, a risk analysis and the plan of activities divided by each semester.

3.1 Project Background

CSWSYS is composed of several components, each serving different purposes. The entire CSWSYS system functions as a dual-control validation system for communication between the DSP (Data Service Provider) and the smart meters. The smart meters only receive commands after they have been validated by CSWSYS. Each component within CSWSYS exposes a REST web service, enabling communication with the component (although some endpoints may be hidden). When a service request from the DSP arrives at CSWSYS, the CSWSYS component takes responsibility for handling the request.

The CSWSYS component offers low-level detection of all commands sent to the devices, ensuring that any command sent to the meter is either non-supply affecting or has been properly authorized by a supplier. The service request arrives in XML format, and the first action taken by the CSWSYS component is to validate the XML schema against a predefined schema. If the received XML schema does not comply with the predefined schema, the validation fails, an exception is raised, and the request is denied. Additionally, there are defined limits (upper/lower limits) that serve to validate the values within the XML. If a value falls below or exceeds the defined limits, the request is rejected. This validation process is known as Attribute Anomaly Detect (ADA). Once validated, the request is inserted into the database. If the validation fails, the request is rejected.

It is important to note that CSWSYS does not send commands directly to the smart meters. The smart meters only receive service requests from the DSP after CSWSYS has validated and authorized the commands. All the components within CSWSYS are independent of each other. During the maintenance work,

the other components remain unchanged. Since they communicate via IP, having components running on different technologies (e.g., some in Thorntail and the CSWSYS component in Quarkus) does not cause any issues. Changing the CSWSYS component will not impact the other components, as they operate independently.

In the last sub-section it was concluded that Quarkus was the best option to replace Thorntail for the CSWSYS component. But choosing Quarkus implies first upgrading the Java version (from 8 to 11). It will be necessary to assess the existing Thorntail project to understand its structure, dependencies, and functionality, analyse the dependencies used in Thorntail, and migrate them to the corresponding Quarkus dependencies. As mentioned before, there are some incompatibilities between Thorntail and Quarkus due to Quarkus's lack of EJBs, which will need to be updated, as well as the imports, annotations, and specific source code that differ from Quarkus. Currently, the CSWSYS component has one hundred and three unit tests, and it is expected that they will remain unchanged at the functional level. The unit tests will be a factor in validating the changes, indicating that the methods maintained functionality as they returned the expected result to pre-defined input data. Table 3.1 represents a static analysis of the CSWSYS component with Thorntail implemented.

Table 3.1: CSWSYS component static analysis

	Classes	Unit Tests	Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Blank Lines
CSWSYS Component	88	103	5720	3782	66%	944	994

This analysis will help to assess differences between Thorntail and Quarkus, and track some measures like the number of lines (sub-section 2.3.1), and the effort in the end of the migration. It is not expected that these metrics will rise much, as they may indicate bad signs of migration. Using the SonarQube tool, we were able to obtain complexity metrics for the CSWSYS component. In terms of cyclomatic complexity, the CSWSYS component has a total score of 237, which is the sum of complexities across all 74 files. The highest cyclomatic complexity value for a single file is 26. In terms of cognitive complexity, the component has an overall score of 103, with 16 being the highest cognitive complexity score among all files. In general, the complexity within the CSWSYS component's files is low. Since no source code changes (functional changes) are expected, it is anticipated that the complexity of the component will remain unchanged.

3.2 Project Requirements

In this section it will be presented the existing CSWSYS component requirements. Either the functional and non-functional requirements of the component had already been defined, and with the migration of the component, it is desirable that they remain unchanged. However it is important to understand how these

requirements specifications were developed and what are the defined requirements.

The diagram present in Figure 3.1 illustrates the requirements hierarchy from the top level System Requirements Definition (SRD) to the Software Requirements Specifications (SRS) produced by CSW.

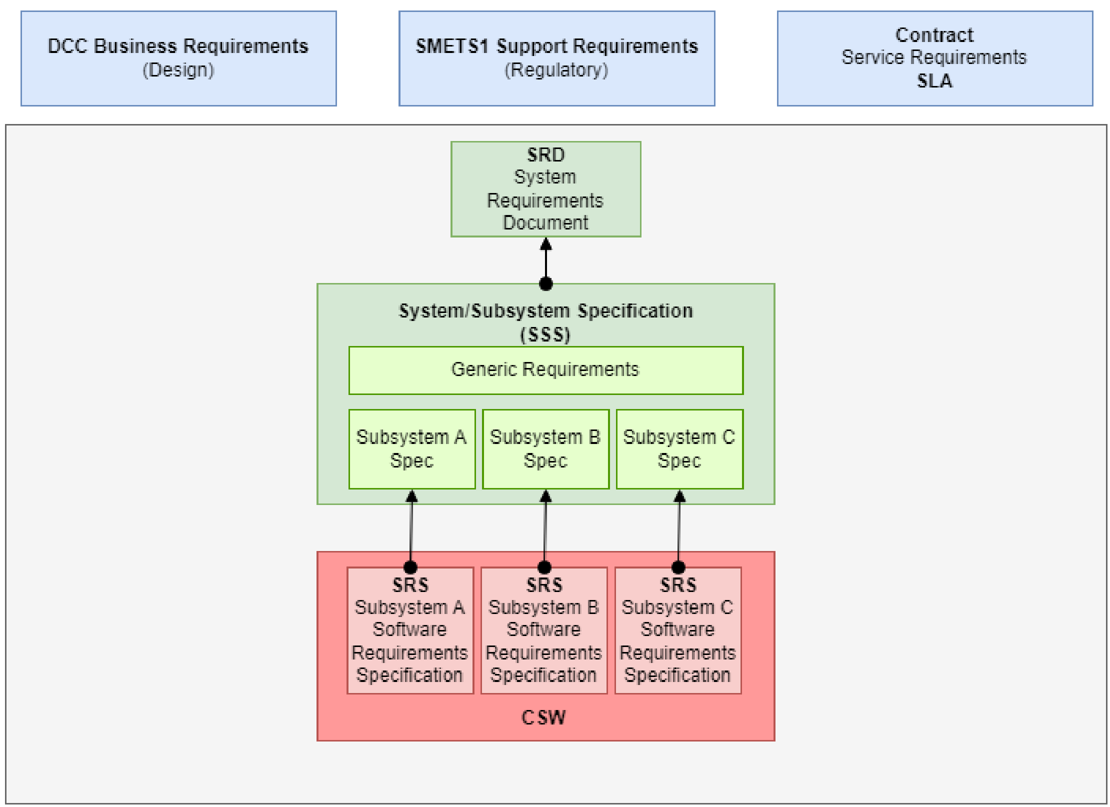


Figure 3.1: Requirements Hierarchy

Firstly, at the highest level, we have the System Requirements Document from the client, which describes the behaviour and the features of a system. At the lowest level, software requirements are developed by CSW according to specifications provided by the client. These SRS are in User Stories format.

Beyond the software requirements, there are some requirements that must be met. Business Requirements, in order to follow design guidelines to ensure that the solution architecture is compliant with standards. SMETS1 Support Requirements that are more at the regulatory level. Furthermore, there are also Service Level Agreements (SLAs)/ key performance indicators (KPIs) which impact the application - these SLAs/KPIs are present in the contractual obligations that CSW needs to fulfil and they flow down into System/Subsystem Specification (SSS)/SRS.

3.2.1 Functional Requirements

Functional requirements represent statements of services that the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. A function is a defined objective or characteristic action of a system or component and a functional requirement specifies a function that a system or system component must be able to perform.

An example of Functional Requirements of the CSWSYS component, are:

- Receive, validate and persists a service request into Database;
- Export monitoring data to management agents;
- Provide information to determine if the service is running.

3.2.2 Non-Functional Requirements

Non-functional requirements (NFRs) are a set of specifications that describe the system's operation capabilities and constraints and attempt to improve its functionality [21]. Non-functional requirements can be classified into several categories. The following apply to the project:

3.2.2.1 Dependability Requirements

The dependability of a system is a judgement about the user's trust in that system. It reflects the extent of the user's confidence that it will operate as expected and that it will not 'fail' in normal use [22]. The following dependability requirements are the already defined non-functional requirements regarding the CSWSYS component:

- **Availability:** Ability of the system to deliver the services when requested. It is important that the system remains available as long as possible so that it can respond to all requests.

  The service shall support high availability. Permit maintenance of system component while the service remains online. The system should meet a target Availability level of 99,990%;

- **Maintainability:** specify the ease of repairing the system after a failure has been discovered or changing the system to include new features. The mean time to restore the system (MTTR) following a system failure must not be greater than 6 hours;

- **Performance**: Defines how fast a software system or a particular piece of it responds to certain users' actions under a certain workload;
  The system shall enable the Service Provider to meet a response time of 4 seconds;

- **Scalability** The service shall scale to support large volumes of service requests - 30000 services requests per second.

CSW is not able to test and assure these requirements alone. Assurance of these requirements requires integration with other systems in the customers test environments. However, the application needs to be designed and technology (such as language and tech stack) needs to be chosen that are capable of fulfilling these requirements.

There are some performance tests carried out by CSW for this application, but they only assure partially these requirements. Therefore, in general, performance testing and test assurance for non-functional requirements are outside of the work discussed on this document.

3.3 CSWSYS Architecture

In this section will be presented the existing architecture of the CSWSYS application and as in the previous sub-section, it is desirable that the architecture remain intact after the migration is complete.

The architecture of a software system can be represented in various ways, and in the case of CSW, the chosen method is Unified Modeling Language (UML), which provides a visual representation of the architecture, design, and implementation of complex software systems. The diagram presented in Figure 3.2 illustrates the CSWSYS architecture. On the left side, highlighted in green, are the external systems that interact with CSWSYS. On the right side, highlighted in yellow, is the CSWSYS component that forms the solution. The orange component represents the focus of this study, and the other components represent the data sources involved.

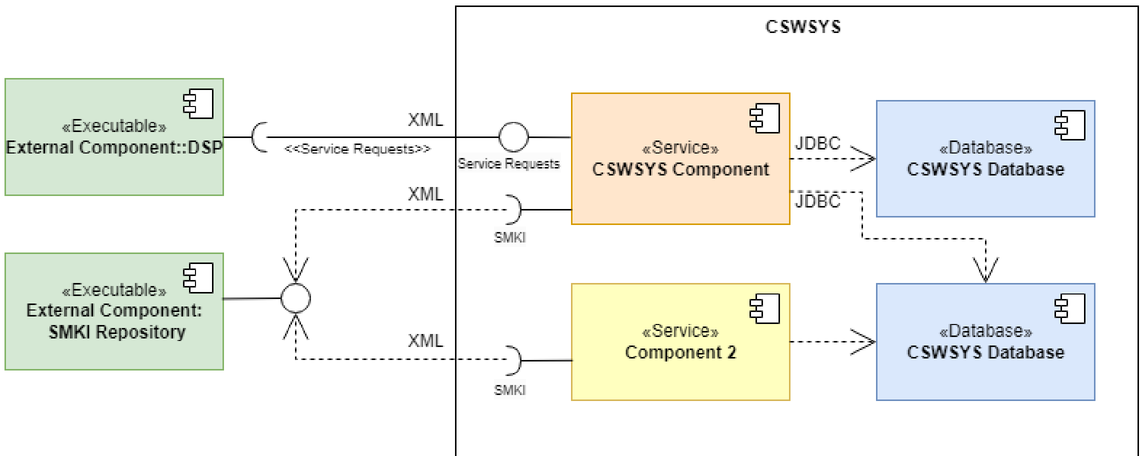


Figure 3.2: Existing Project Architecture

There are additional components within the CSWSYS architecture; however for confidentiality reasons, they are not represented, leaving only the component

that will be migrated and other components about which the CSWSYS component require information.

The CSWSYS component (orange) is responsible for receiving Service Requests Variant (SRVs) from the Data Service Provider (DSP) component, validate those service requests and manage their persistence. The validation includes XML schema compliance. If any validation fails, the service request is rejected.

Initially, CSWSYS was monolithic, but due to the nature of the project and the need to be stateless, it went from monolithic to Service-Oriented Architecture (SOA). As we can see from the Figure 3.2, the architecture of CSWSYS is composed by many services. SOA is a way of software development that makes software components (services) reusable. These services can be integrated into new applications because they have the capacity to communicate with other services across platforms and languages. This method provides more flexibility than the monolithic method.

The structure of SOA is based on "loose coupling" concept. This means that services can be called without know how the service is implemented and do not require complex point-to-point integration as is the case in a monolithic architecture [51].

3.4 Approach

A set of objectives was defined for this internship and to accomplish all of them, an approach consisting of the following steps was selected. Since this work is a maintenance job involving the migration of a component, the approach will follow the process outlined in sub-section 2.2.1, and as this is a change of environment, the adaptive maintenance type discussed in sub-section 2.2.2 will be used. This means that, initially, there is a phase of analysing how everything works and the necessary modifications, followed by a modification implementation phase, and ending with the testing and validation phase. Since the components are independent from each other, the work will be focused on the CSWSYS component, applying the changes to the source code of the component.

For this work, a variation of the Kanban methodology which is the methodology used within the team, was adopted, aligned with the approach used in this internship, which will be presented in the next sub-section. Two types of Kanban meetings were addressed during the internship: Daily meetings, which allowed the team to receive feedback from the team about problems and challenges that arose as the work progressed since they had more experience in the project; Retrospective meetings every two weeks in order to reflect on the work that has been done in the past few weeks. The remaining Kanban meetings, such as Replenishment meetings, were not taken into account as the tasks had been defined according to the plan. The planning for the second semester, which will be discussed in sub-section 3.4.3, took into account the approach presented in the next sub-section.

3.4.1 Approach

1. **Run the application with the Thorntail still in operation (legacy technology)**
   This step includes the installation of all dependencies and libraries that the application needs to run. Once the application is running, this is a good starting point to use the program comprehension, technique addressed in sub-section 2.2.3, to analyse the code and identify the pieces of code that need to be modified;

2. **Get familiar with Quarkus**
   As Quarkus is a recent technology, the next step is getting familiar with the Quarkus features and development processes. This step includes the implementation of the Quarkus sample application to understand how it works, to get to know the needed dependencies and to understand the needed migration tools;

3. **Define adaption requirements**
   After getting familiar with Quarkus and understand how it works, there are requirements to run Quarkus which differ from Thorntail (e.g. versions, libraries), as well as other configurations that need to be defined;

4. **Dependency changes**
   To start the migration itself, it is first necessary to handle the dependency changes by replacing the Thorntail dependencies with Quarkus dependencies and handle missing dependencies because, by removing Thorntail dependencies, some transitive dependencies will be missing, which can be added as needed. Some dependency versions should be updated because, with Java 11, some implementations became deprecated;

5. **Code changes**
   Some practices used in Thorntail will have to be changed, as well as imports. These changes will be made incrementally in small modules so that they can be made more easily. Variables covered in this section 2.3, such as code size and understandability, should be considered while modifying the code to avoid making it worse. The CSWSYS component has 3782 source code lines (Table 3.1), as it is expected to keep the number of final lines close to this.

6. **Verify migration through testing**
   Validate the implemented changes (where applicable) through unit tests and system tests, it is important to verify that it has been implemented correctly and that it does not negatively impact the existing functionality of the system. The CSWSYS component has currently 103 unit tests (3.1) and it is expected that these tests will continue to be passed so that the component maintains good functionality. These are two criteria of success ;

7. **Documenting the change**
   Update any necessary documentation to reflect the changes that have been made. This helps ensure that the system remains up-to-date and that future

changes can be made in a controlled and organized manner. As seen earlier, documentation is an important tool that helps in program comprehension, saving future effort.

3.4.2 Initial Plan

The figure 3.3, represents a first version of the work plan drawn up at the beginning of the semester, and which was amended during the course of the internship.

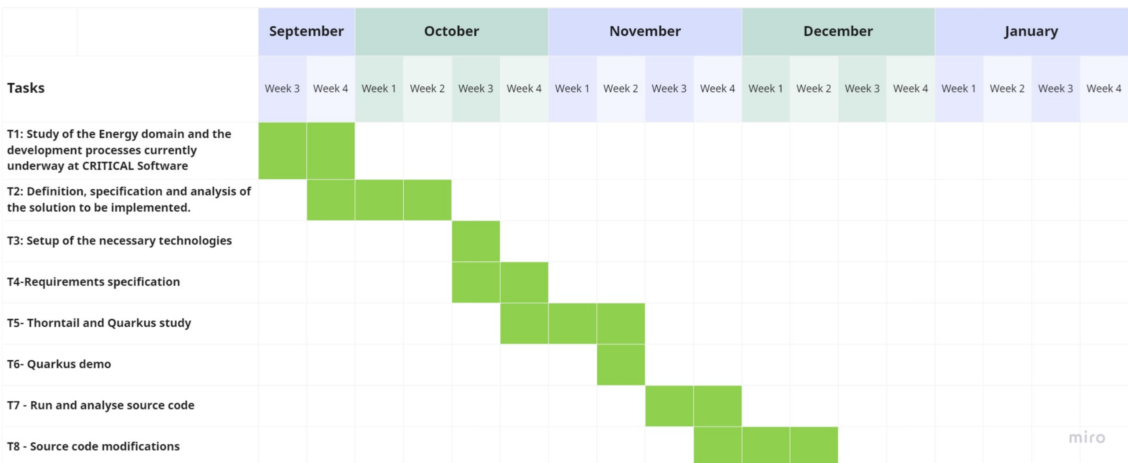


Figure 3.3: Initial planning for the first semester

The initial plan presented contained high-level tasks in the first semester that included the study of energy sector in the context of this work, definition and evaluation of the requirements, the analysis of solutions on the market, development and integration of the solution and validation of the solution, and in the second semester, the continuation of the last three. This initial plan was detailed in order to have concrete tasks to follow.

The first semester served to familiarize with the project since it is an extensive project, and to understand which norms and procedures are used within Critical Software in order to choose a solution that fits and follow these rules. The task of development and integration of solution and validation has been changed for the second semester since theses tasks require the understanding of the code.

3.4.3 Planning

The following Gantt diagrams contain the activities conducted in the first semester (Figure 3.4) as well as the plan for the second semester (Figure 3.5). In summary, throughout the first semester, the main activities consisted of: understanding the world of energy, more specifically the UK SMIP, studying and analysing the project, identifying the existing component requirements, analysing the technology to be changed and finding alternative solutions that are in line with the standards and procedures that are used by the CSW, and understanding the processes that are used in software maintenance.

Figure 3.4: First-semester plan

The Figure 3.5 represents a draft made for the second semester based on information accessible back then. The plan consisted, at the beginning of February, of the setup of necessary technologies to run the current CSWSYS application, the analysis of the source code to understand how the application works, the implementation of necessary code changes in order to migrate the application from Thorntail to the previously studied technology alternative, and finally the implementation of the validation strategy.

Figure 3.5: Second-semester initial plan

The Figure 3.6, represents the planning that was followed in the second semester. This plan differs from the first version (Figure 3.5) carried out previously. This plan was divided in two main tasks based on the defined approach to implement the migration (sub-section 3.4) previously discussed: Develop, integration of the solution, Solution Validation. And also, three presentations during the semester "Show and Tells".

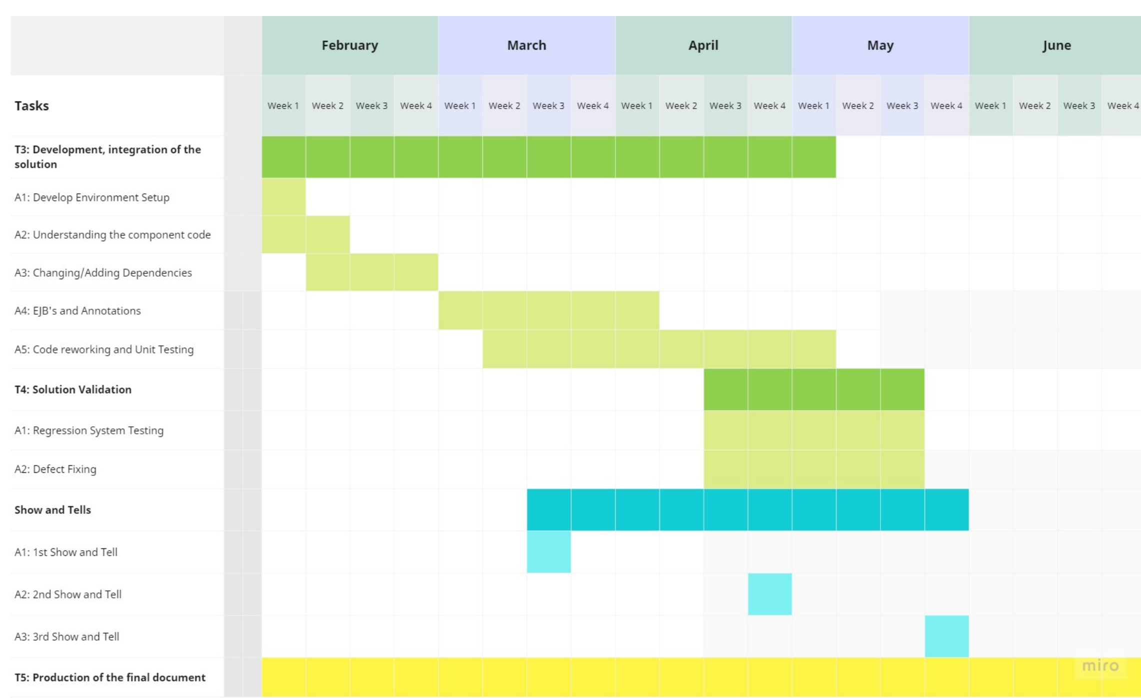


Figure 3.6: Second-semester

The task *Development, integration of the solution*, was divided into five main sub-tasks: Setup the environment in order to run the current CSWSYS project; familiarisation with the project and understand the structure and source code of the CSWSYS component; Changes at the dependency level; EJB's and annotation changes; Code reworking and unit testing. After the code changes, the next task was to validate the solution with regression system testing and defect fixing in order to fix some problems that arose during the execution of the tests. Finally some Show and Tells presentations were carried out during the semester. The goal of the "Shows and Tells" was to show the team the work that was being developed and some of the problems faced in order to receive some feedback and make improvements. The presentations can be found in Appendice A.

3.4.4 Risk Plan

Due to the nature of this work, a risk plan capable of summarizing the most relevant risks was conducted. This plan includes the risk statement, which is a concise description of a potential risk; the consequence, which is the potential negative impact that a software risk could have; the probability, which is the likelihood that a specific software risk will occur; the impact, which is the potential negative effect that a software risk could have; and the mitigation plan, which is a set of actions and strategies that are put in place to reduce the likelihood and impact of software risks. Nine risks were identified, which are detailed in the next sub-sections.

3.4.4.1 Risk Matrix

The Figure 3.7 represents the Risk Matrix. The categories ranges from low to high. The impact goes from Insignificant to Catastrophic and the probability goes from Very Unlikely to Very Likely. This helps to prioritize and analyse the risks of this work.

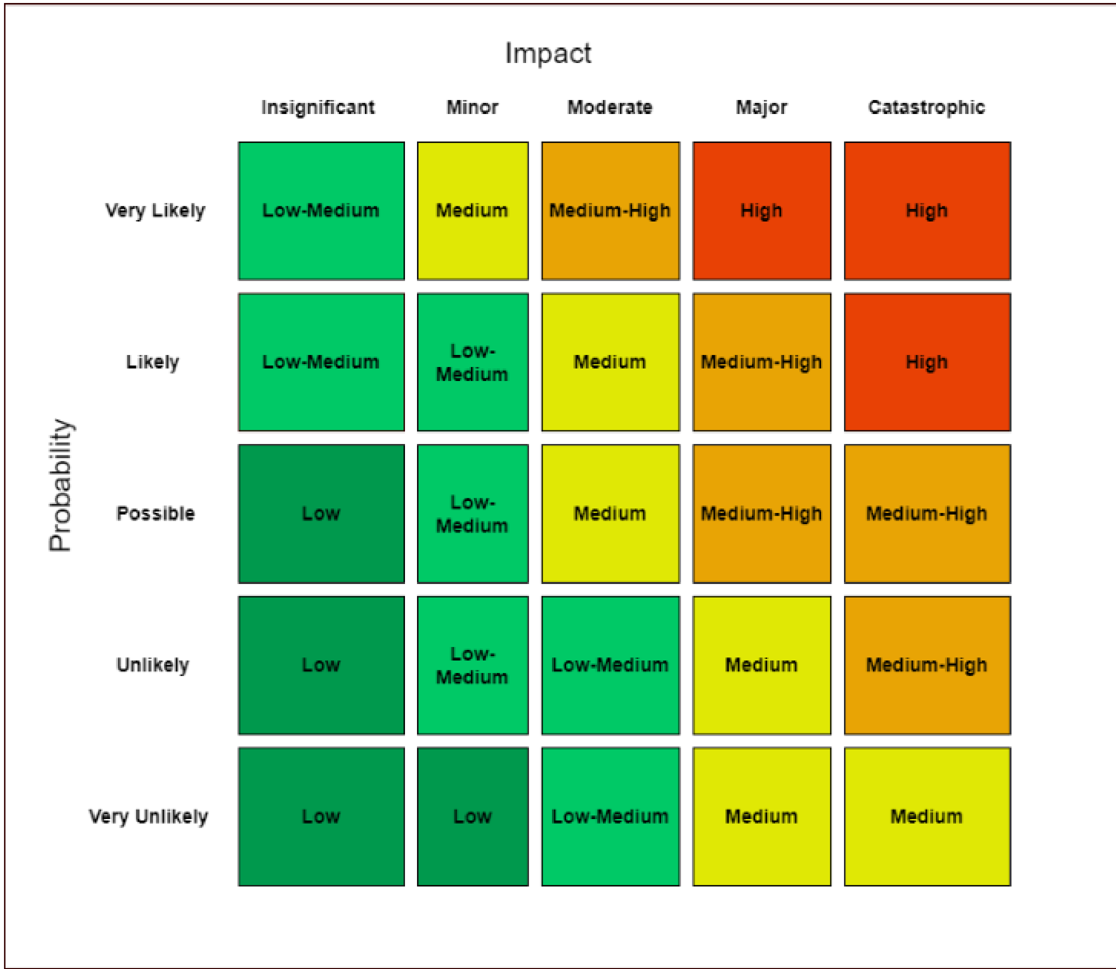


Figure 3.7: Risk Matrix (from - [52])

3.4.4.2 [RISK-01]

- **Risk Statement:** Lack of documentation.
- **Consequence:** The lack of available documentation make the process of understanding the source code more difficult.
- **Probability:** Unlikely.
- **Impact:** Moderate.
- **Mitigation Plan:** Ask for help from collaborators to help clarify details of the source code.

3.4.4.3 [RISK-02]

- **Risk Statement:** New technologies make estimations more difficult.
- **Consequence:** The lack of experience on Quarkus and Thorntail and on the systems itself make difficult to size the work and this can lead to delays on completing the code change and other foreseen problems during the work.
- **Probability:** Possible.
- **Impact:** Moderate.
- **Mitigation Plan:** Preparatory research work to get the code running and familiarisation with Quarkus.

3.4.4.4 [RISK-03]

- **Risk Statement:** Insufficient time to validate the whole component.
- **Consequence:** There may be features that have not been properly tested.
- **Probability:** Possible.
- **Impact:** Moderate.
- **Mitigation Plan:** Testing during code migration in order to minimise the amount of possible bugs.

3.4.4.5 [RISK-04]

- **Risk Statement:** Dependency Incompatibility can lead to delays in the work.
- **Consequence:** It may affect and delay the execution of the remaining tasks, leading to the objective not being met on time
- **Probability:** Possible.
- **Impact:** Moderate.
- **Mitigation Plan:** May require code changes or using previous versions in order to mitigate these incompatibilities.

3.4.4.6 [RISK-05]

- **Risk Statement:** The use of certain dependencies may contain vulnerabilities that compromise the integrity of the component.
- **Consequence:** It may affect and delay the execution of the remaining tasks, leading to the objective not being met on time

- **Probability:** Possible.
- **Impact:** Moderate.
- **Mitigation Plan:** Walkthrough of the different issues and weighting of resolution, according to the time available.

3.4.4.7 [RISK-06]

- **Risk Statement:** Using different dependencies can cause changes in boot performance, either positively or negatively.
- **Consequence:** There may not be time in the internship to complete the tasks and carry out this analysis with Performance tests.
- **Probability:** Likely.
- **Impact:** Moderate.
- **Mitigation Plan:** Performance and Load Tests in order to compare the performance against the Thorntail, and assess its behaviour. Out of scope.

3.4.4.8 [RISK-07]

- **Risk Statement:** Quarkus does not run on Java 8, so it is necessary to migrate to Java 11. All the code is based on Java 8.
- **Consequence:** The remaining components as they are in Java 8 will not be able to compile and will not work afterwards.
- **Probability:** Very Likely.
- **Impact:** Moderate.
- **Mitigation Plan:** Allocate more development time, focusing on code reworking.

3.4.4.9 [RISK-08]

- **Risk Statement:** With the change of dependencies (elimination of Thorntail and Java 8, replacement by Quarkus), there are implementations of these dependencies that no longer exist in the new versions.
- **Consequence:** Extensive code changes can impact the planning.
- **Probability:** Very Likely.
- **Impact:** Major.
- **Mitigation Plan:** Add more time on rework tasks, descope everything that is not essential.

3.4.4.10 [RISK-09]

- **Risk Statement:** In order to validate the changes it necessary to run the system tests as regression tests.
- **Consequence:** There may be no time to execute all the created system tests.
- **Probability:** Very Likely.
- **Impact:** Moderate.
- **Mitigation Plan:** Create a sub set of the system tests, just for the CSWSYS component, in order to not impact the plan and still validate the functionality of the component.

Table 3.2 represents the risks that occurred and the respective mitigation plan that was active.

Table 3.2: Risks table

Risk ID	Happened	Mitigation Plan
RISK-01		Not activated
RISK-02		Not activated
RISK-03		Not activated
RISK-04		Not activated
RISK-05		Not activated
RISK-06	X	Performance and Load Tests became Out of Scope.
RISK-07	X	Allocation of more time for code reworking.
RISK-08	X	Add more time to code rework tasks, descope Performance and Load tests.
RISK-09	X	Execution of a sub-set of the system tests regarding CSWSYS component.

Chapter 4

Development, Integration of the solution and Testing

This chapter aims to show the work developed in order to meet the main objective, which is to migrate the CSWSYS component from Thorntail to Quarkus.

It is divided into eight sections. The first section addresses the first phase of the work, which was to configure the setup to run the whole project. The second section is related to the structure and organisation of the CSWSYS.

From the third section on, the phase of implementing modifications begins, starting with the dependencies changing. The fourth section focuses on source code changes, followed by the configuration file changes section. Sections six and seven deal with unit tests and system tests, respectively. In the last section, there is an overview of the changes and the results of those changes.

4.1 Environment Setup

During the work, the company laptop was used, and this laptop has the following specifications:

- **Processor:** 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz;
- **RAM Memory:** 32,0 GB;
- **System Type:** 64-bit operating system, x64-based processor;
- **Graphics Card:** NVIDIA GeForce GTX 1650 with Max-Q Design.

In order to be able to build and run the CSWSYS, the following tools needed to be installed:

- Oracle VM Virtual Box;
- Ubuntu 22.04;

- IntelliJ IDEA 2022.3.2;
- Docker 23.0.0;
- Apache Maven 3.6.3;
- SDKMAN - Java 8;
- DBeaver 22.3.4;
- SoapUI 5.7.0;
- Git.

The virtual machine was configured with 16 GB of RAM, four processors, and 250 GB of storage. The SDKMan, which is a tool for managing parallel versions of multiple Software Development Kits (SDK), was used to install the latest version of Java 8, which is the version used within the project. The project is located inside Bitbucket, which is a Git-based source code repository hosting service [53]. The source code of CSWSYS and hence the CSWSYS component code was cloned from the main branch to the local machine using Secure Shell (SSH). After cloning the code, in IntelliJ, it was necessary to define which SDK should be used in order to execute the project. After installing Java 8, inside the project structure tab of IntelliJ, the SDK version 1.8 was used.

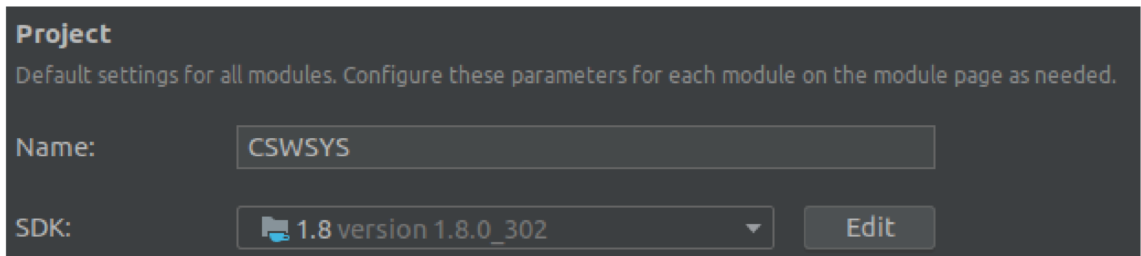


Figure 4.1: SDK - Project Structure

The default Java version was also defined to Java 8 (Figure 4.2):

```
Terminal:  Local  +
brunogandres@brunogandres:~/Desktop/Rep/dco$ java -version
openjdk version "1.8.0_302"
OpenJDK Runtime Environment (build 1.8.0_302-b08)
OpenJDK 64-Bit Server VM (build 25.302-b08, mixed mode)
```

Figure 4.2: CSWSYS Java version

While using Maven, most of the project-specific configuration is present in the pom.xml. Maven also provides a settings file, *settings.xml*, which allows to specify which local and remote repositories it will use. This file was copied to *user.home/.m2* directory as represented in Figure 4.3. The .m2 directory is where

Maven stores all the local dependencies in the machine. When the Maven build command is executed, the project dependencies are stored locally for future use [75]. In *settings.xml* credentials that allow to access the defined repositories are stored. Without this file, or if the credentials are wrong or out of date, the project will not compile, as it can not access to the repositories.

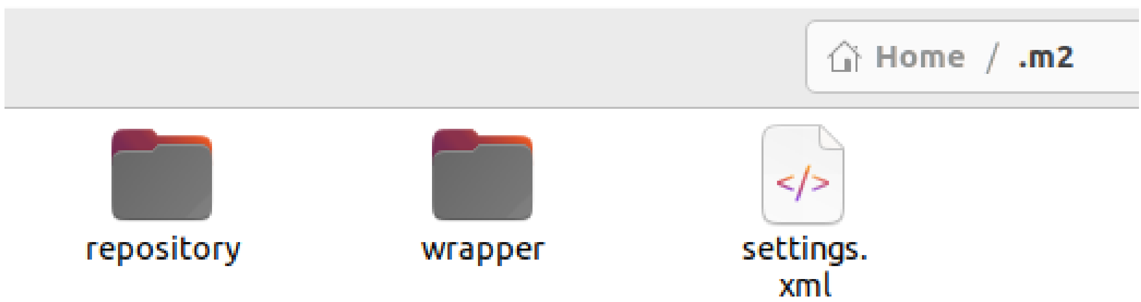


Figure 4.3: settings.xml

The Smart Technology Solutions (STS) division also has some code guidelines and checkstyle definitions defined, which are present in the *settings.xml* file. IntelliJ uses those guidelines in order to verify if the code complies with these guidelines and conventions.

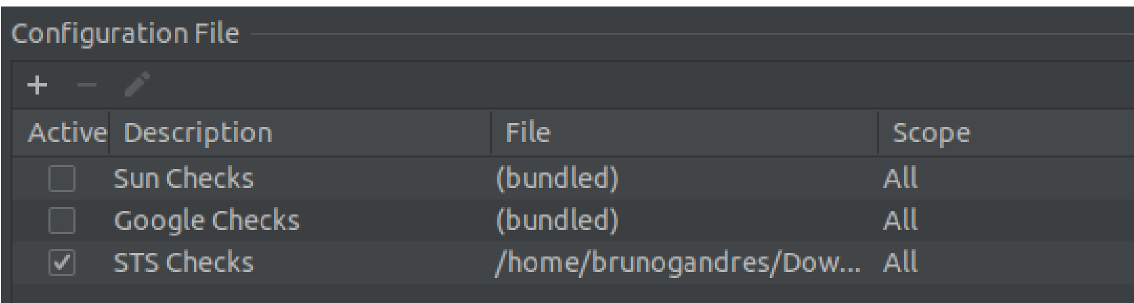


Figure 4.4: STS Checkstyle

When there are some checksytle errors, for example, unused imports or variables in the code, the code fails to compile, forcing the developer to use good code practices. Following a consistent coding style throughout the whole project (this includes indentation, naming conventions, and code formatting), as discussed earlier, improves the understandability of the code (sub-section 2.3.4) making it easier to read and to understand, saving effort when software maintenance tasks need to be carried out.

4.2 CSWSYS Structure and Organization

After completed the environment setup phase, the next step was to get familiar with the structure and organization of the CSWSYS project. The CSWSYS is composed by several modules, including the CSWSYS component. Although they

are independent components, some of them depend on common modules. Essentially, the structure of the project is organised:

1. Component Modules;
2. Distributable Module;
3. Parent Project Object Model (POM).

The project structure is composed by multiple maven modules and sub-modules. At the root of the project it is possible to find: ***pom.xml***, ***core*** module and ***dist*** module. The parent POM is an XML file that contains information about the project and configuration details used by Maven to build the project. Each module, contains also a POM with all needed dependencies. The **core** module, which is a library and should be included as a dependency as needed, holds common features for all application modules. The **distributable** (dist) module is responsible to build the final package of the application with all dependencies. Each module, with the exception of the distributable module has a list of sub-modules divided by features:

- **Common-api**: Holds common Application Programming Interface (API) to share between other modules;
- **Business-api**: Business API with services interfaces;
- **Business**: The implementation of all services on business-api;
- **Data**: Holds the relational data objects and any Data Access Object (DAO) services;
- **Webservices**: The implementation of Web Services.

Since this project uses Thorntail, it is expected that the POM of each submodule inside the CSWSYS component module contains Thorntail's dependencies.

Figure 4.5, illustrates an example of the most typical dependencies used in the CSWSYS project and also in the CSWSYS component. First, the *javax* dependency refers to the dependencies for Java 8, and second the *io.thorntail* dependency, the dependencies for Thorntail. This will be the initial stage of the work, and it is necessary to remove dependencies like these ones.

```
<!-- Java EE 7 dependency -->
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
</dependency>

<!-- Health check -->
<dependency>
    <groupId>io.thorntail</groupId>
    <artifactId>microprofile-health</artifactId>
</dependency>
```

Figure 4.5: Thorntail Dependencies example

4.2.1 Test Approach

For the CSWSYS project, including the CSWSYS component, the following definitions are being considered:

- **Unit Tests:** Each unit test is very limited in scope and test only one specific class;
- **System Testing:** System testing method refers to a range of test types focused on verifying that the complete, integrated engineered system behaves in compliance with its specified system requirements.

For this internship, the System Testing will work as Regression System Testing. Regression testing consists of re-running tests to ensure that previously developed and tested software still performs after changing from Thorntail to Quarkus. Regression testing allows the understanding of whether defects have been introduced or uncovered in unchanged areas of the software.

4.3 Dependencies Changing

Moving to the Modification Implementation stage (sub-section 2.2.1.3), this subsection addresses the dependency changes phase. As this work is an adaptive maintenance type, some adaptation prerequisites needed to be defined to migrate the CSWSYS component to Quarkus based on the Quarkus demo [74].

The first prerequisite, as previously mentioned, involves upgrading the Java version from eight to eleven. Additionally, it is necessary to import the Quarkus Bill of Materials (BOM) to replace the Thorntail dependencies, update the Dockerfile to enable Quarkus to run in JVM mode, and create a new application configuration file, as it differs from the one used in Thorntail. This section focuses

on analyzing the dependencies utilized in Thorntail and migrating them to their corresponding counterparts in Quarkus.

This phase began with the transition from Java 8 to Java 11. While Quarkus is compatible with Java versions higher than 11 (such as Java 17), it is generally not advisable to make such a significant leap between versions from a software maintenance vision. This could result in various compatibility issues that may not have a solution. Following the Java version upgrade, the initial and expected symptom was the deprecation of certain dependencies and associated libraries. Figure 4.6 displays the first encountered errors after switching the Java version. Notably, the removal of the package *javax.xml.bind.annotation* and others from Java SE 11 and JDK 11 [76].

```
 package javax.xml.bind.annotation does not exist
 package javax.xml.bind.annotation does not exist
 package javax.xml.bind.annotation does not exist
 package javax.xml.bind.annotation does not exist
cannot find symbol
```

Figure 4.6: First errors after Java version change

The initial approach to address this problem involved changing and replacing deprecated or missing dependencies in the root POM. However, since this project consists of multiple modules, and some modules that do not interact with the CSWSYS component still rely on Java 8, modifying the main POM, which contains dependencies for all components, is not a viable solution. If this were done, none of the components would function properly. To overcome this issue, the second approach involved modifying and removing dependencies specifically within the CSWSYS component's POM. Additionally, modules that do not interact with the CSWSYS component were also commented out. As a result, during project compilation, only the modules on which the CSWSYS component depends will utilize Java 11 without Thorntail's dependencies.

This approach presents a significant challenge: the inability to compile all the components simultaneously. While these components are independent, they rely on certain shared modules, such as the **core** module, which contains common features. Leaving the other components uncommented introduces Java incompatibilities and hampers ongoing development work, as making changes in each component becomes necessary. However, these changes fall outside the scope of the internship.

Initially, due to existing unit tests, the CSWSYS component could not compile with the changes being made, particularly the new Java version. As a workaround, the project was initially compiled with the flag: **mvn clean package -DskipTests**, which skips executing unit tests. Thus, our first goal was to successfully compile the project with the new dependencies and without running the unit tests.

4.3.1 Before Migration

Within the CSWSYS component's POM, the javax and Thorntail's dependencies were removed. In this sub-section, a brief description of each old dependency will be provided to understand its purpose. Additionally, explanations will be provided for the decisions made regarding these dependencies.

4.3.1.1 thorntail-bom

The **io.thorntail:bom-all** (Figure 4.7) is a reference to the BOM for the Thorntail project in the Maven build system. A Bill of Materials (BOM) is a file that defines a set of versions for a group of dependencies, making it easier to manage dependencies and ensure compatibility among them. The bom-all refers to a BOM that includes all the necessary dependencies for Thorntail. This is the main dependency of Thorntail which was removed.

```
    <!-- Thorntail -->
<!--    <dependency>-->
<!--      <groupId>io.thorntail</groupId>-->
<!--      <artifactId>bom-all</artifactId>-->
<!--      <version>${version.thorntail}</version>-->
<!--      <scope>import</scope>-->
<!--      <type>pom</type>-->
<!--    </dependency>-->
```

Figure 4.7: Thontail BOM

4.3.1.2 Javax (javaee-api)

Java EE (Java Platform, Enterprise Edition) is a collection of specifications and APIs (Application Programming Interfaces) that provide a platform for developing enterprise-level applications in Java. The Java EE API (javaee-api) is a reference implementation of these specifications, providing the necessary classes, interfaces, and methods for building Java EE applications. It is important to note that starting from Java EE 8, the Java EE specifications have been transferred to the Eclipse Foundation and renamed as Jakarta EE. Therefore, if the goal is to maintain Java EE functionality in Java 11 or later, Jakarta EE should be used instead. The Figure 4.5 illustrates the java dependency used inside the CSWSYS component.

4.3.1.3 io.thorntail - Microprofile Metrics

The **io.thorntail** is a Java library/framework that provides an implementation of the MicroProfile Metrics specification. MicroProfile is a project that aims to

enhance the development of microservices-based applications on the Java EE or Jakarta EE platform. MicroProfile Metrics (Figure 4.8), one of the specifications within MicroProfile, focuses on collecting and exposing metrics about the application's performance, health, and other relevant aspects. It provides a standardized way to monitor and measure various aspects of microservices, such as response times, error rates, throughput, and resource utilization.

```
Thorntail Microprofile Metrics
<dependency>
  <groupId>io.thorntail</groupId>
  <artifactId>microprofile-metrics</artifactId>
  <version>${version.thorntail}</version>
</dependency>
```

Figure 4.8: Thorntail Metrics

4.3.1.4 io.thorntail - Microprofile Health

MicroProfile Health, one of the specifications within MicroProfile, enables developers to create health checks for microservices applications. Health checks are routines that assess the state and availability of various components within the application, such as databases, external services, or internal dependencies. These checks provide insights into the overall health and readiness of the application. This dependency is commonly used inside the CSWSYS component, as the Figure 4.5 illustrates.

4.3.2 After Migration

The previous dependencies were changed by the following ones, in order to keep the same functionality of the application. Also an overall vision of the main dependencies that were changed before and after the migration will be handled in Table 4.1.

4.3.2.1 quarkus-bom

After removing Thorntail's dependencies, the next step was to add Quarkus dependencies. The first one was the **quarkus-bom**, which is a Maven BOM that centrally manages the versions of Quarkus dependencies in a project. This BOM replaced the previously discussed Thorntail BOM. By importing the quarkus-bom into the root POM, it is possible specify the desired Quarkus dependencies without explicitly mentioning their versions. The BOM ensures that all the listed dependencies use compatible and tested versions, minimizing the chances of version conflicts or compatibility issues. Version 2.16.1 was used for the quarkus-

bom. With this addition, it became possible to add Quarkus dependencies to the CSWSYS component.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>${quarkus.platform.group-id}</groupId>
      <artifactId>${quarkus.platform.artifact-id}</artifactId>
      <version>${quarkus.platform.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Figure 4.9: Quarkus Dependency

As we can see in Figure 4.10 the version were defined globally, in the beginning of the root POM in order to keep consistency and reduce the probability of making mistakes. Additionally, from a software maintenance perspective, this good practice allows to save time when versioning is carried out, as is not necessary the version of each single Quarkus dependency.

```
<quarkus.platform.artifact-id>quarkus-bom</quarkus.platform.artifact-id>
<quarkus.platform.group-id>io.quarkus.platform</quarkus.platform.group-id>
<quarkus.platform.version>2.16.1.Final</quarkus.platform.version>
<quarkus-plugin.version>2.16.1.Final</quarkus-plugin.version>
```

Figure 4.10: Quarkus Version

By utilizing a BOM, ensures that all dependencies within the project are using compatible versions. This reduces the risk of conflicts and compatibility issues that may arise when using different versions of dependencies. It also reduces the maintenance overhead, as it is not necessary to manually update version numbers across the modules. All the modules that import the BOM will automatically use the updated version.

4.3.2.2 jandex-maven-plugin

In the sub-section 2.5.5 a analysis of Quarkus was carried out and one of the main discussed advantages of Quarkus is its fast boot time. In order to achieve this, Quarkus moves steps like classpath annotation scanning forward from runtime to build-time. For this, it is necessary to announce all dependencies at build-time.

Quarkus heavily utilizes Jandex at build time, to discover various classes or annotations. When the plugin is enabled, it scans the bytecode of the application's classes and generates an index file containing metadata about these classes. This index file is then used by Quarkus at runtime to optimize class loading and

enhance performance. It is not necessary to interact directly with the plugin, as its functionality is transparently integrated into the build process when using Quarkus. To generate the index it was necessary to add the plugin (Figure 4.11) to the build file, in plugin section.

```
<plugin>
  <groupId>io.smallrye</groupId>
  <artifactId>jandex-maven-plugin</artifactId>
  <version>3.0.5</version>
  <executions>
    <execution>
      <id>make-index</id>
      <goals>
        <goal>jandex</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Figure 4.11: Jandex Maven Plugin

4.3.2.3 jakarta.el

The jakarta.el package (Figure 4.12) serves as the counterpart to the javax.el package in the Jakarta EE platform. It offers the same functionality for working with EL expressions but operates under the Jakarta EE namespace. With the transfer of Java EE technologies to the Eclipse Foundation and their rebranding as Jakarta EE starting from Java EE 8, package names have been modified to align with this transition. In this Java version, the javax namespace can still be used while retaining the jakarta functionality. However, in more recent versions, it is mandatory to switch the namespace from javax to jakarta.

```
<dependency>
  <groupId>org.glassfish</groupId>
  <artifactId>jakarta.el</artifactId>
  <scope>test</scope>
</dependency>
```

Figure 4.12: jakarta.el dependency

4.3.2.4 Eclipse Microprofile Health/Metrics

To replace the Microprofile health and metrics dependencies from Thorntail, two alternatives were considered. The first option was to utilize the dependencies provided by Quarkus itself, such as **quarkus-smallrye-metrics** [55] and **quarkus-smallrye-health** [56]. The second option was to use the Microprofile Metrics and Microprofile Health from the Microprofile project.

When comparing SmallRye from Quarkus and Eclipse MicroProfile from a software maintenance perspective, several factors need to be evaluated. SmallRye is tightly integrated with the Quarkus framework, offering a seamless development experience. This integration allows for efficient maintenance of Quarkus applications, leveraging the full capabilities of SmallRye within the Quarkus ecosystem. However, using SmallRye with Quarkus introduces a dependency on the Quarkus framework.

On one hand, this dependency may limit flexibility if, in the future, there is a need to migrate to a different framework or if the project requirements change. On the other hand, Eclipse MicroProfile promotes interoperability by providing standardized specifications and APIs. This simplifies maintenance efforts, as applications developed using MicroProfile can be easily migrated or integrated with other MicroProfile-compliant frameworks and tools. For these reasons, the second option was chosen (Figure 4.13).

```
<!-- Old -->
<!--        <dependency>-->
<!--            <groupId>io.thorntail</groupId>-->
<!--            <artifactId>microprofile-metrics</artifactId>-->
<!--        </dependency>-->
<!-- New -->
<dependency>
    <groupId>org.eclipse.microprofile.metrics</groupId>
    <artifactId>microprofile-metrics-api</artifactId>
</dependency>
```

(a) Microprofile Metrics

```
<!-- Health check -->
<!-- Old -->
<!--        <dependency>-->
<!--            <groupId>io.thorntail</groupId>-->
<!--            <artifactId>microprofile-health</artifactId>-->
<!--        </dependency>-->
<!-- New -->
<dependency>
    <groupId>org.eclipse.microprofile.health</groupId>
    <artifactId>microprofile-health-api</artifactId>
</dependency>
```

(b) Microprofile Health

Figure 4.13: Microprofile Dependencies

Other dependencies were also added and removed. The Table 4.1 represents an overall vision of the before and after regarding the dependencies that had been changed.

Table 4.1: Overall dependency changes

Before Migration	After Migration
<dependency> <groupId>io.thorntail</groupId> <artifactId>microprofile-health</artifactId> </dependency>	<dependency> <groupId>org.eclipse.microprofile.health</groupId> <artifactId>microprofile-health-api</artifactId> </dependency>
<dependency> <groupId>io.thorntail</groupId> <artifactId>microprofile-metrics</artifactId> </dependency>	<dependency> <groupId>org.eclipse.microprofile.metrics</groupId> <artifactId>microprofile-metrics-api</artifactId> </dependency>
<dependency> <groupId>org.glassfish</groupId> <artifactId>javax.el</artifactId> <scope>test</scope> </dependency>	<dependency> <groupId>org.glassfish</groupId> <artifactId>jakarta.el</artifactId> <scope>test</scope> </dependency>
<dependency> <groupId>mysql</groupId> <artifactId>mysql-connector-java</artifactId> </dependency>	<dependency> <groupId>io.quarkus</groupId> <artifactId>quarkus-jdbc-mysql</artifactId> </dependency>
<dependency> <groupId>org.mockito</groupId> <artifactId>mockito-all</artifactId> <scope>test</scope> </dependency>	<dependency> <groupId>org.mockito</groupId> <artifactId>mockito-core</artifactId> <scope>test</scope> </dependency>
Added	<dependency> <groupId>org.mockito</groupId> <artifactId>mockito-inline</artifactId> <scope>test</scope> </dependency>
<dependency> <groupId>org.powermock</groupId> <artifactId>powermock-api-mockito</artifactId> <scope>test</scope> </dependency>	Removed
<dependency> <groupId>org.powermock</groupId> <artifactId>powermock-api-mockito</artifactId> <scope>test</scope> </dependency>	Removed
Added	<dependency> <groupId>io.quarkus</groupId> <artifactId>quarkus-scheduler</artifactId> </dependency>
Added	<dependency> <groupId>io.quarkus.arc</groupId> <artifactId>arc</artifactId> </dependency>
Added	<dependency> <groupId>io.smallrye.config</groupId> <artifactId>smallrye-config-source-yaml</artifactId> </dependency>
Added	<dependency> <groupId>io.quarkus</groupId> <artifactId>quarkus-container-image-docker</artifactId> </dependency>
Added	<dependency> <groupId>io.quarkus</groupId> <artifactId>quarkus-hibernate-orm</artifactId> </dependency>

Some dependencies will be discussed in the following sections, as they have become necessary in the course of the changes.

4.4 EJB's and Annotations

The next phase of the work was focused on migrating the CSWSYS component code from Thorntail to Quarkus. This involves updating package imports, annotations, and any Thorntail-specific code to their equivalent Quarkus counterparts. The Quarkus documentation was also used to provide information on how to perform these updates. After replacing the dependencies discussed in the last section, some errors appeared due to missing implementations and deprecated annotations, such as:

- javax.ejb.Stateless;
- javax.ejb.Singleton;
- javax.ejb.Startup;
- javax.ejb.ScheduleExpression;
- org.eclipse.microprofile.Health.

As mentioned in sub-section 2.5.6, there are certain incompatibilities between Thorntail and Quarkus due to Quarkus not supporting EJBs. All annotations found in services and methods need to be modified, and every type of bean must have an annotation for it to be used. Bean classes without a bean-defining annotation are not detected, and Quarkus does not compile them. So, services with no annotations should be explicitly marked as @Dependent.

4.4.1 @Health

Starting with health checks, the annotation **@Health**, which was used within the project from the org.eclips.microprofile.Health import, became deprecated (Figure 4.14) [57]. Two options were available to replace @Health with the new dependency: @Liveness or @Readiness. A Health check for readiness allows third party services to know if the application is ready to process requests or not. For example, a readiness check might check dependencies, such as database connections. A Health Check for liveness allows third party services to determine if the application is running.

Since the CSWSYS component is responsible for receiving and processing service requests and has some database connections dependencies, the annotation @Readiness was the chosen, being the most suitable in this case.

```java
//Old
//import org.eclipse.microprofile.health.Health;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
//New
import org.eclipse.microprofile.health.Readiness;
```

Figure 4.14: Health import change

4.4.2 @Stateless

The @Stateless annotation is typically used in Java EE applications to indicate that a bean does not maintain any conversational state between method calls. However, since Quarkus is designed for microservices and serverless architectures, it does not support the use of @Stateless. Instead, the @ApplicationScoped annotation was used to achieve similar functionality. This annotation indicates that an object is created once per application. The @Stateless annotation was replaced with @ApplicationScoped. Furthermore, the @Stateless also automatically initiated database transactions. The @Transactional annotation was introduced in places where transactions were necessary.

4.4.3 Eager Instantiation Beans

Along the code, some @Startup annotations were found. This annotation mark a singleton bean for eager initialization during the application startup sequence [58]. In Quarkus, beans are instantiated lazily by default, which means they are created only when they are first requested. Since javax.ejb.Startup annotation does not exist in Java 11, it was necessary to find an alternative. In the Quarkus documentation, three solutions were present. To instantiate a bean eagerly could be through [59]:

- Declare an observer of the StartupEvent (Figure 4.15);

```java
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;

import io.quarkus.runtime.ShutdownEvent;
import io.quarkus.runtime.StartupEvent;
import org.jboss.logging.Logger;

@ApplicationScoped
public class AppLifecycleBean {

    private static final Logger LOGGER = Logger.getLogger("ListenerBean");

    void onStart(@Observes StartupEvent ev) {               1
        LOGGER.info("The application is starting...");
    }
```

Figure 4.15: Listening for startup event (from - [60])

- Use the bean in an observer of the StartupEvent (Figure 4.16);

```
@Dependent
class MyBeanStarter {

  void startup(@Observes StartupEvent event, AmazingService amazing, CoolService cool) { 1
    cool.toString(); 2
  }
}
```

Figure 4.16: Bean in an observer of the StartupEvent (from - [59])

- Annotate the bean with @io.quarkus.runtime.Startup (Figure 4.17).

```
@Startup 1
@ApplicationScoped
public class EagerAppBean {

   private final String name;

   EagerAppBean(NameGenerator generator) { 2
     this.name = generator.createName();
   }
}
```

Figure 4.17: Startup Annotation (from - [59])

The chosen approach for this replacement also considered software maintenance. The utilization of the StartupEvent (Figure 4.15) and **@Startup** (Figure 4.17) comes from the io.quarkus dependency. StartupEvent is an event class provided by Quarkus, enabling the observation of the application's startup phase. However, these approaches rely on the Quarkus dependency, which restricts flexibility and increases the effort required for future migration to another framework. To address these concerns, a more generic approach was adopted for the replacement. The startUp method is annotated with @Observes to indicate event observation. The observed event is @Initialized(ApplicationScoped.class), which denotes the initialization of the application scope. This method will be invoked during application startup, allowing for the execution of any necessary actions at that time.

```
no usages  brunogandres
protected void startUp(@Observes @Initialized(ApplicationScoped.class) final Object toStart) {
    LOG.debug("operation=startUp, start={}", toStart);
}
```

Figure 4.18: Startup Approach

There is no real difference between utilizing the StartupEvent from Quarkus and the @Initialized(ApplicationScoped.class). The discrepancy lies in the timing

that each is fired. The StartupEvent is consistently triggered after the @Initialized(ApplicationScoped.class) event, which itself is fired during the initialization of the application context. By incorporating this startUp method, it becomes unnecessary to modify all the @Startup annotations for eager instantiation, thereby saving effort in future endeavors.

With the Thorntail version, a **@Singleton** annotation was also used, but similarly as the startup dependency, the javax.ejb Singleton became deprecated and needed to be replaced. With @Singleton, only one instance is created in the whole application and does not terminate until the application is shut down [72]. This behaviour is similar to the @ApplicationScoped annotation. Also, the @Singleton annotation was accompanied by the @Startup annotation, which informed the EJB container to initialise the bean at startup.

Although in Quarkus the @Singleton annotation also exists, the old annotation in the CSWSYS component source code was replaced by @ApplicationScoped. The difference between these two annotations is also related to eager instance creation. On one hand, with the @ApplicationScoped, a single bean instance is used for the application and shared among all injection points. The instance is created lazily, i.e., once a method is invoked upon the client proxy [73].

On the other hand, @Singleton is just like @ApplicationScoped except that no client proxy is used. The instance is created when an injection point that resolves to a @Singleton bean is injected. Since it has no proxy, an instance is created eagerly when the bean is injected. By contrast, an instance of an @ApplicationScoped bean is created lazily, i.e., when a method is invoked upon an injected instance for the first time [73].

In the end, Quarkus recommends sticking with the @ApplicationScoped, and in order to keep the eager initialization of the Singleton, the previous startUp approach (Figure 4.18) was also used to achieve this.

4.4.4 Private Members

In Quarkus, private members (fields or methods) of a bean class are not directly accessible for injection or interception by the CDI container. By default, the CDI container can only manage and interact with public or package-private members of a bean class. It is generally not recommended to directly access private members of a bean in CDI applications as it violates encapsulation and can create challenges in maintenance and testing. If Quarkus DI needs to access a private member, it has to use reflection, which is why it is encouraged to avoid using private members in beans.

In the initial stages, the chosen approach involved replacing private fields with package-private injection fields (as shown in Figure 4.19), following the recommendation provided by Quarkus. Later on, the approach shifted from field injection to constructor injection. Both construction injection and field injection are methods used for implementing dependency injection in object-oriented programming.

```
2 usages
@Inject
//private RequestManagerResource requestManagerResource;
RequestManagerResource requestManagerResource;
```

Figure 4.19: Field Injection

Construction injection involves passing dependencies to an object through its constructor. In other words, the dependencies are injected into the object when it is created. This approach is typically used when dependencies are required for the object to function correctly. While both construction injection and field injection can be effective, construction injection is generally considered to be the better approach because it ensures that the object has all of its required dependencies from the start, which can make it easier to reason about the object's behavior and dependencies. Additionally, using construction injection can make the code easier to test because it allows dependencies to be easily mocked or stubbed during testing. The figures 4.20 and 4.21 represent the modifications from field injection to constructor injection, where a constructor was created with the variables as arguments. These variables which previously had no modifier (package-private) became private final.

```
//@Inject
private final ServiceRequestValidationService serviceRequestValidationService;
//ServiceRequestValidationService serviceRequestValidationService;

//@Inject
private final RequestService requestService;
//RequestService requestService;

//@Inject
private final SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService;
//SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService;

//@Inject
private final SrvTimeoutConfiguration srvTimeoutConfiguration;
//SrvTimeoutConfiguration srvTimeoutConfiguration;
```

Figure 4.20: Constructor Injection Fields

```
@Inject
public RequestManagerServiceImpl(final ServiceRequestValidationService serviceRequestValidationService,
                                 final RequestService requestService,
                                 final SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService,
                                 final @TimeoutConfigurationDefault SrvTimeoutConfiguration srvTimeoutConfiguration) {
    this.serviceRequestValidationService = serviceRequestValidationService;
    this.requestService = requestService;
    this.specificProcessingSRVLocatorService = specificProcessingSRVLocatorService;
    this.srvTimeoutConfiguration = srvTimeoutConfiguration;
}
```

Figure 4.21: Constructor Injection

After converting the field injection to constructor injection, some errors started to appear, as represented in Figure 4.22. These errors derived from the private member annotations not being present within the constructor in the parameters. This way, Quarkus could not understand what it had to inject.

```
[ERROR]         [error] Build step io.quarkus.arc.deployment.ArcProcessor#validate threw and exception:
javax.enterprise.inject.spi.DeploymentException: javax.enterprise.inject.AmbiguosResolution exception: Ambiguos dependencies for
type com.criticalsoftware.CSWSYS.CSWSYScomponent.common.business.SrvTimeoutConfiguration and qualifiers [@Default]
[ERROR]         - java member:
com.criticalsoftware.CSWSYS.CSWSYScomponent.common.business.RequestManagerServiceImpl():SrvTimeoutConfiguation
[ERROR]         - declared on CLASS bean [types=
[com.criticalsoftware.CSWSYS.CSWSYScomponent.common.business.RequestManagerService,
com.criticalsoftware.CSWSYS.CSWSYScomponent.common.business.RequestManagerServiceImpl, java.lang.Object], qualifiers=[@Default,
@Any], target=com.criticalsoftware.CSWSYS.CSWSYScomponent.common.business.RequestManagerServiceImpl]
```

Figure 4.22: Constructor Injection Error

All the annotations of beans need to go inside the constructor, so Quarkus know what to inject, except of @Inject which stays outside. In addition, when using constructor injection the inject fields turned from package-private to private final, increasing the robustness of the code. This type of visibility, encapsulate the state of an object by restricting direct access from outside the class. Marking a variable as final ensures that its value cannot be changed once assigned. This immutability guarantees that the variable will remain constant throughout the lifetime of the object, reducing the potential for bugs caused by unintended modifications.

Since Quarkus has a simplified bean discovery the content of beans.xml is ignored. The beans.xml file is known as the bean archive descriptor, and was present along the project. CDI has the notion of a bean archive. A bean archive is just a module that has a file named beans.xml in the META-INF directory. The container looks for beans in bean archives. It ignores other modules. Since Quarkus ignore the content of the file [71], the beans.xml was removed. Before removing beans.xml, some conflicts also appeared as Quarkus could not made the discovery of some beans, and the component did not work properly, and after deleting the beans.xml the conflicts disappeared.

4.5 Configuration Files

Maintaining configuration files is a crucial aspect of software maintenance. Quarkus uses its own configuration mechanism, commonly based on the MicroProfile Config specification. As part of the migration process, the Thorntail-specific configuration files and properties were converted to the Quarkus configuration format. After completing the phase of changing dependencies, EJBs, and annotations, the project's configuration scope became a focus area, as expected. It was anticipated that issues related to newly added or existing dependencies would arise during the course of the project.

The initial error encountered during code compilation was caused by the failure to inject certain settings into the variables. This issue was resolved by updating the version of the dependency ***microprofile-config-api*** from 1.2.1 to 2.0. This

API from MicroProfile Config supports configuration injection, allowing configuration properties to be directly injected into application components, such as CDI beans. This promotes the use of dependency injection and simplifies the retrieval of configuration values within the application.

The second error that arose was related to the database connection. Figure 4.23 represents the error log of the database connection.

```
1 [ERROR] Failed to execute goal io.quarkus:quarkus-maven-plugin:2.16.1.Final:build on project CSWSYS-
  component: Failed to build quarkus application: io.quarkus.builder.BuildException: Build Failure: Build
  failed due to errors
2 [ERROR] [error]: Build setp io.quarkus.agroal.deployment.AgroalProcessor#build threw an exception:
  io.quarkus.runtime.configuration.ConfigurationException: Unable to find a JDBC driver corresponding to the
  database kind 'mysql' for the datasource 'cswsys-ds'. Provide a suitable  JDBC driver extension, define
  the driver manually, or disable the JDBC datasource by adding 'quarkus.datasource.CSWSYS.jdb=false' to
  your configuration if you don't need it.
3
```

Figure 4.23: JDBC driver Error

This error can be splitted in two parts. Firstly, the error message "Unable to find a JDBC driver corresponding to the database kind 'mysql' for the data source" indicates that the driver dependency was missing. To resolve this, the dependency ***quarkus-jdbc-mysql*** (Table 4.1) was added.

The second part of the error states "define the driver manually, or disable the JDBC datasource by adding 'quarkus.datasource.CSWSYS.jdbc' to your configuration if you don't need it." The drivers are manually defined within a configuration file. The project already had a configuration file called **server-config.yml**, which allows for the definition of properties and settings related to components and services used in the application, including the database URL, username, password, driver class, connection pool settings, and any other database-specific configuration options. The existing configuration was created using Thorntail flags and needed to be updated for compatibility with Quarkus.

```yaml
thorntail:
  http:
    port: 8070
  management:
    http:
      port: 9970
      disable: true
  datasources:
    jdbc-drivers:
      com.mysql:
        driver-class-name: com.mysql.cj.jdbc.Driver
        xa-datasource-class-name: com.mysql.cj.jdbc.MysqlXADataSource
        driver-module-name: com.mysql
```

Figure 4.24: server-config.yml

Regarding the database, Quarkus supports Hibernate ORM which is a popular object-relational mapping (ORM) framework for Java. It provides a conve-

nient way to map Java objects to relational database tables and perform various database operations. Quarkus sets many Hibernate ORM configuration settings automatically and often uses more modern defaults, but in order to maintain the existing config properties, the Hibernate ORM has various options that need to be added to the new Quarkus application properties files.

The next phase of the work was to create a new configuration file with Quarkus, maintaining the project settings, config properities and adding the Hibernate ORM. To use the HibernateORM, the dependency ***quarkus-hibernate-orm*** which is present in Table 4.1, was added to the root POM. Quarkus allows to configure various properties for Hibernate ORM via the application.yaml file. It possible to define database connection details, Hibernate dialect, transaction management, and other ORM-related settings. Quarkus provides a set of default configurations, so it was only need to specify properties specific of the application. To enable YAML configuration in Quarkus, the dependency *quarkus-config-yaml* was added.

```
quarkus:
  scheduler: true
  http:
    port: 8070
  datasource:
    dco-data:
      db-kind: mysql
      username:
      password:
      jdbc:
        enable-metrics: true
        url: jdbc:mysql://localhost:3306/?useSSL=false&allowPublicKeyRetrieval=true&max_allowed_packet=25M
        validation-query-sql: SELECT 1 FROM _dual WHERE dual_column = 1
        background-validation-interval: 0
  hibernate-orm:
    dco-data:
      datasource: dco-data
      packages: com.criticalsoftware. . data.entities
```

Figure 4.25: Quarkus - Application.yaml

With Thorntail, a persistence.xml file was used, whose goal was to configure persistence units in JPA applications. However, unlike traditional Java EE applications, Quarkus does not rely on a persistence.xml file for configuring JPA. With Quarkus, there is no need to have persistence.xml [64]. Using persistence.xml and HibernateORM throws an exception, so the approach was to remove it, leaving one less file to be maintained.

The Figure 4.25 represents the upgrade of the configuration file from Thorntail (Figure 4.24) to Quarkus. The idea was to maintain all properties that were already defined, such as HTTP port and MySQL database configuration. The remaining config properties of the project have not changed, since it is expected that they maintain their constant value for the proper functioning of the application. With the changes of the configuration file, some code changes needed to be carried out. Since CSWSYS has multiple datasources (Figure 3.2), the migration and the use of Hibernate ORM, required some adaptions.

Figure 4.26 represents the changes along the source code, with the the replace

of annotation from javax with the io.quarkus.hibernate.orm.PersistenceUnit package-level annotation.

```
1 OLD
2 //@PersistenceContext(unitName = "cswsys-data")
3 //private EntityManager entityManager;
4
5 NEW
6 @PersistenceUnit("cswsys-data")
7 EntityManager entityManager;
```

Figure 4.26: HibernateORM code adaptions

Another example of code changes concerning the new configuration file were the paths to get the properties from the configuration. As it is possible to see in Figure 4.27, the variable `CSWSYS_DB_URL` changed from "thorntail.datasources.." to "quarkus.datasource....".

```
1 OLD
2 //public static final String CSWSYS_DB_URL = "thorntail.datasources.data-sources.cswsys-
  ds.connection-url";
3 NEW
4 public static final String CSWSYS_DB_URL = "quarkus.datasource.cswsys-data.jdbc.url";
```

Figure 4.27: Config property path example

This minor change was applied to all paths that contained "thorntail", otherwise would get errors where could not find the properties inside the file. These values are then injected as Config Properties into variables. The @ConfigProperty annotation is part of the MicroProfile Config specification, which is supported by Quarkus. It is used to inject configuration values from the application's configuration sources into Java classes.

```
1 @Inject
2 @ConfigProperty(name = DatabaseHealthConfigKey.CSWSYS_DB_URL)
3 //private String url;
4 String url;
```

Figure 4.28: ConfigProperty Variable

When a field or a method parameter is annotated with @ConfigProperty, Quarkus will search for a matching configuration property and inject its value into the annotated element. In this example, the @ConfigProperty annotation is used to inject the value of the `CSWSYS_DB_URL` into the 'url' field. The name attribute specifies the name of the configuration property to inject. Quarkus will search for a

matching property in the application.yaml and populate the message field with its value. Wrong values inside the configurations files can lead to the bad function of the application, that's why is important to keep maintained.

Quarkus does much of its configuration at build time, reading and using some configurations properties during this phase. These properties are fixed at build time and it is not possible to change them at runtime. It always need to repackage the application in order to reflect changes of such properties. In order to separate both properties, in Quarkus Documentation, the properties fixed at build time are marked with a lock icon in the list of all configuration options while the remaining properties are the additional properties which can be overridable [70].

Two properties files were used:

- One file in the ***dist***/*CSWSYS component/src/main/resources* folder in order to provide all the necessary configs for build time process - Datasources and Hibernate ORM properties;
- One file in the ***deploy***/*local/CSWSYS component/config* folder in order to provide additional configs that can be overridable at runtime - Logging properties, secondary database properties;

During the build phase of the application, if the module does not have a config file inside the resources directory, at build time the application will fail because it needs to have a config file (related to the database properties because of the hibernate orm). If the config file is in the *src/main/resources* directory, the JAR will be created successfully and the config file will be added inside the JAR.

Initially, the same application.yaml file with fixed and overridable properties was used in both the *dist* and *deploy* folders. However, this approach caused issues during the regression system testing phase (section 4.7). The configuration properties file within the JAR package also included overridable properties, resulting in a conflict between the values in the JAR configuration file and the test environment configuration file. As a result, the configuration properties within the JAR file differed from those in the testing environment, leading to errors. In order to solve it, on the JAR package side there was only a configuration file with fixed properties and on the other side only properties that are overridable.

4.6 Unit Tests

After successfully compiling the CSWSYS component code without including the unit tests, it was anticipated that the subsequent step would involve verifying the correctness of the preceding changes by ensuring the passage of all unit tests. Unit testing plays a crucial role in detecting bugs and problems at an early stage of the development process. By testing each code unit in isolation, it becomes possible to identify and address issues before they spread to other areas of the system. This proactive approach aids in minimizing the overall cost and effort involved in resolving bugs at a later stage.

During the execution of the unit tests for the CSWSYS component, certain issues were encountered. Firstly, the existing unit tests make use of PowerMock [61], a robust testing framework for Java that extends the capabilities of popular mocking frameworks such as Mockito. However, PowerMock exhibits incompatibilities with Java 11, necessitating its removal from the unit tests. Secondly, some tests specifically target private methods, requiring PowerMock to employ reflection for mocking these private methods. As previously mentioned, Quarkus discourages the use of reflection whenever possible to optimize startup time and minimize memory usage. Figure 4.29 illustrates the warnings observed during the compilation phase, wherein Quarkus needed to use reflection for accessing a private method or variable.

```
WARNING: An illegal reflective access operation was occurred
WARNING: An illegal reflective access by com.criticalsoftware.CSWSYS.core.interceptors.LoggerInterceptorTest to field
java.lang.reflect.Field.modifiers
WARNING: Please consider reporting this to the maintainers of com.critical.software.CSWSYS.core.interceptors.LoggerInterceptorTest
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
```

Figure 4.29: Reflection Warning

The extensive reliance on PowerMock can serve as an indication of poor code design. Ideally, all code should be designed in a way that promotes testability. The practise of mocking private methods or accessing internal state through reflection using PowerMock may suggest a lack of proper encapsulation. By adhering to the principles of encapsulation, code can be structured in a modular, maintainable, and testable manner, eliminating the need for advanced mocking frameworks. To invoke and modify the internal state of private methods and fields, the utility class Whitebox provided by PowerMock is utilised. Whitebox offers a range of static methods that bypass visibility restrictions, enabling access to private members (fields or methods), invocation of private methods, and manipulation of internal state. However, it is important to note that while Whitebox provides a means to interact with the internal behaviour of a class, this approach contradicts the principles of encapsulation and modularity.

The focus of this phase of the work was divided in:

- Replace PowerMock as is not compatible with Java 11;
- Delete Whitebox, in order to delete reflection from code.

4.6.1 Replace PowerMock

The first step taken to delete PowerMock from unit tests, was remove all the PowerMock dependencies from the POM's (Figure 4.30).

The PowerMock was replaced with Mockito [62]. One dependency of Mockito already existed inside project, **mockito-all**. This dependency was replaced by

```
<!--    <dependency>-->
<!--      <groupId>org.powermock</groupId>-->
<!--      <artifactId>powermock-api-mockito</artifactId>-->
<!--      <scope>test</scope>-->
<!--    </dependency>-->
<!--    <dependency>-->
<!--      <groupId>org.powermock</groupId>-->
<!--      <artifactId>powermock-module-testng</artifactId>-->
<!--      <scope>test</scope>-->
<!--    </dependency>-->
```

Figure 4.30: PowerMock dependency

mockito-core 5.1.1 (Figure 4.31) since mockito-core artifact is Mockito's main artifact and mockito-all is an out-dated dependency that bundles Mockito as well as its required dependencies[63].

```
<dependency>
  <groupId>org.mockito</groupId>
  <!--      <artifactId>mockito-all</artifactId>-->
  <artifactId>mockito-core</artifactId>
  <scope>test</scope>
</dependency>
```

Figure 4.31: Mockito dependency replace

The upgrade to mockito-core resulted in the deprecation of certain implementations previously utilized with mockito-all. For instance, mockito.Matchers was replaced by mockito.ArgumentMatchers, and MockitoAnnotations.initMocks was replaced by MockitoAnnotations.openMocks. Furthermore, the mockStatic method provided by PowerMockito was substituted with mockStatic from Mockito. Consequently, with the removal of the PowerMockito dependency, specific code modifications were required. One significant change involved the removal of the PrepareForTest annotation. This annotation served as an indicator for PowerMock, informing it which classes would be manipulated during testing.

```
//@PrepareForTest({ConfigProvider.class, ConfigProviderResolver.class})
//public class FileConfigUtilsTest extends PowerMockTestCase {
brunogandres +2 *
public class FileConfigUtilsTest {

    1 usage
    private static final String CONFIG_YAML_PROPERTIES = "config/config.yaml";
```

Figure 4.32: PowerMock PrepareForTest Annotation

The usage of initMocks was replaced with openMocks (Figure 4.33), which initializes the annotated fields such as @Mock or @InjectMocks. In all tests, the

presence of initMocks is crucial, as it ensures that all mocks are properly initialized. Without this initialization step, the mocks would result in null pointer exceptions.

```
//initMocks(this);
autoCloseable = MockitoAnnotations.openMocks( testClass: this);
```

Figure 4.33: Unit Test Changes - initMocks()

In Figure 4.34, we can observe the replacement of mockStatic from PowerMockito with mockStatic from Mockito. Additionally, to utilize this method for mocking static methods and final types, it was necessary to include the dependency of mockito-inline (as mentioned in Table 4.1). Without this dependency, an error, as shown in Figure 4.35, would occur. By default, the MockMacker feature is disabled and can be activated through an extension file. However, with the mockito-inline dependency, inline mock making is enabled without the need to configure a separate MockMaker extension file. This eliminates the necessity of maintaining an additional file.

```
mocked = mockStatic(ConfigProvider.class);
//PowerMockito.mockStatic(ConfigProvider.class);
```

Figure 4.34: Unit Test Changes - mockStatic

```
java.lang.IllegalStateException: Could not initialize plugin: interface org.mockito.plugins.MockMaker (alternate: null)
<7 internal lines>
```

Figure 4.35: Mockito-inline dependency

4.6.2 Delete Whitebox

The next phase of the unit test modifications involved removing the use of Whitebox. Whitebox facilitated access to private fields and the invocation of private methods through the utilisation of reflection. Throughout the unit tests, certain methods from Whitebox, such as setInternalState and invokeMethod, were employed. The setInternalState method allowed for the modification of the value of private or static fields within a class. It circumvented the encapsulation provided by Java's access modifiers (such as private or protected) and enabled the modification of internal state for testing purposes. On the other hand, the invokeMethod method, as its name suggests, permitted the invocation of private or static methods and the retrieval of their return values. These methods relied on reflection to accomplish their objectives. To eliminate the use of reflection in the unit tests, these methods were removed from the unit-tests.

In order to work around the usage of these methods and maintain the intended purpose of the unit tests, certain modifications were implemented. For

the setInternalState method, two approaches were employed. The first approach involved utilising setters (Figure 4.37), while the second approach involved accessing the variables directly (Figure 4.36). Since Quarkus promotes the avoidance of private members, it is permissible to access variables within the same package and assign values to them.

```
//Whitebox.setInternalState(victim, "batchSizeFirmwareUpdateDevices", 50000);
victim.batchSizeFirmwareUpdateDevices = 50000;
```

Figure 4.36: setInternalState Inject Field

```
//Whitebox.setInternalState(victim, "firmwareStrategyMap", firmwareStrategyMap);
victim.setFirmwareStrategyMap(firmwareStrategyMap);
```

Figure 4.37: Replace of setInternalState with setter

When dealing with the invocation of private methods, the purpose of these unit tests was to assess the behavior of these methods. The objective was to verify if the returned result matched the expected outcome or to determine if the expected exception was raised when incorrect input data was provided. To replace the invokeMethod calls, two approaches were utilized: Verifying the private method's execution through a public method - This approach involved testing a public method that would, in turn, invoke the private method being targeted for testing. This allowed access to the private method and facilitated its evaluation; Changing the visibility of the private method from private to package-private or protected, allowing direct access to the method within the same package.

For the first approach, Figure 4.38 illustrates an example of a test aimed at evaluating the private method *createFirmwareUpdateImageData* (as shown in Figure 4.39) and asserting the expected return value. In the original implementation using invokeMethod, it was only necessary to pass the class, the function name, and the input data as parameters.

```
@Test
public void testCreateFirmwareUpdateImageData() throws Throwable {
    //assertEquals(invokeMethod(victim, "createFirmwareUpdateImageData", image), createFirmwareUpdateImage());
    when(firmwareUpdateIEService.contains(null)).thenReturn( value: false);

    victim.imageParsingAndStoring( imageHash: null, image);

    verify(firmwareUpdateIEService, times( wantedNumberOfInvocations: 1)).write(createFirmwareUpdateImage());
```

Figure 4.38: invokeMethod replace

```
private static FirmwareUpdateIe createFirmwareUpdateImageData(final byte[] image) throws FirmwareUpdateImageException {
```

Figure 4.39: Private Method

The idea behind replacing the invokeMethod calls was to identify if the method being tested was invoked within a public method. In the given scenario, the *createFirmwareUpdateImageData* method is called within the public method *imageParsingAndStoring* (as shown in Figure 4.40), allowing access to the private method for testing purposes. Consequently, it was necessary to adapt the test to align with the structure of the method being tested. In the modified approach, the when() method (Figure 4.38) was utilised to enable method stubbing. This method is employed when there is a need to mock specific return values for particular method calls.

```
@Override
@Transactional
public void imageParsingAndStoring(final String imageHash, final byte[] image) throws FirmwareUpdateImageException {
    LOG.info("operation=imageParsingAndStoring, s1sp={}, imageHash={}", getS1sp(), imageHash);
    if (!firmwareUpdateIEService.contains(imageHash)) {
        final FirmwareUpdateIe firmwareUpdateIE = createFirmwareUpdateImageData(image);
        //firmwareUpdateIE = createFirmwareUpdateImageData(image);
        firmwareUpdateIE.setImageHash(imageHash);
        firmwareUpdateIEService.write(firmwareUpdateIE);
    }
}
```

Figure 4.40: Public Method

As observed in Figure 4.40, in order to access the *createFirmwareUpdateImageData* function, the condition within the if clause needs to evaluate to false. This is where the when() method (as shown in Figure 4.38) becomes relevant. Additionally, the desired return value should be defined before calling the public method. The public method is invoked with the input data, ensuring that the private method being tested also receives the same input as before when using invokeMethod. Subsequently, the verify statement is used to validate specific behaviour, functioning similarly to an assert. It verifies that the write() method within the public method is called at least once with the expected value, similar to the previous assert statement. By employing this approach, it was possible to circumvent the use of invokeMethod, avoiding the need for reflection and achieving the same objective as the original unit test. Furthermore, this approach eliminates the dependency on an additional library such as PowerMock.

The second approach discussed earlier involved adjusting the visibility of certain private methods. In some instances, the methods that needed testing were nested within other private methods, which posed challenges for working around them. To address this issue, the visibility of these private methods was changed to package-private. A member with no explicit access modifier is by default accessible only within classes in the same package. By making the methods package-private, they could be accessed and tested within the unit test code.

```
//private byte[] chooseBlocks(final byte[] secureGbcsManufacturerImage) throws FirmwareUpdateImageException
3 usages  jlbarreiro +2
byte[] chooseBlocks(final byte[] secureGbcsManufacturerImage) throws FirmwareUpdateImageException {
```

Figure 4.41: Package-private Method

As a result, within the unit tests, it became sufficient to directly call class.methodName

(as shown in Figure 4.42). Since the methods were changed to package-private, the need for reflection was eliminated.

```
@Test(dataProvider = "testBlocks")
public void testBlocksSelection(final byte[] secureGbcsManufacturerImage, final byte[] selectedBlocks, final boolean result)
    throws Throwable {
    if (result) {
        //assertEquals(selectedBlocks, invokeMethod(victim, "chooseBlocks", secureGbcsManufacturerImage));
        assertEquals(selectedBlocks, victim.chooseBlocks(secureGbcsManufacturerImage));

    } else {
        //assertNotEquals(selectedBlocks, invokeMethod(victim, "chooseBlocks", secureGbcsManufacturerImage));
        assertNotEquals(selectedBlocks, victim.chooseBlocks(secureGbcsManufacturerImage));
    }
}
```

Figure 4.42: Unit Test with package-private method

Several other improvements were made with regard to Whitebox. Initialization functions that were originally annotated with @PostConstruct and declared as private were modified to be protected or even public. The @PostConstruct annotation is used to mark a method that should be executed after dependency injection is completed, allowing for any necessary initialization steps.

```
@PostConstruct
//private void init() {
protected void init() {
    firmwareStrategies.forEach(firmwareStrategy ->
        firmwareStrategyMap.put(firmwareStrategy.getS1sp(), firmwareStrategy));
}
```

Figure 4.43: PostConstruct visibility

Within certain unit tests, it was necessary to initialise and load specific information before running the tests. However, if the @PostConstruct method is declared as private, the framework is unable to invoke it, resulting in the initialization logic not being executed. To address this issue, the method annotated with @PostConstruct was adjusted to be at least package-private or protected. This ensures that the framework can access and invoke the method, allowing the required initialization to take place before the unit tests are executed.

4.6.3 Unit testing with Constructor Injection

As mentioned in the previous sub-section 4.4.4, it was explained that Field Injection was replaced with Constructor Injection. When using Mockito with @InjectMocks, it attempts to inject mocked dependencies using one of three approaches, following a specified order of precedence:

1. Constructor Injection;

2. Property setter injection;

3. Field injection.

Prior to running each unit test, a setup function annotated with @BeforeMethod is executed to reset and create clean mocks. This setup function ensures that the mocks are initialised and all the required configurations are loaded before each individual unit test. Its purpose is to provide a clean and consistent state for each test, preventing any interference or contamination between test cases.

```
@BeforeMethod
public void setUp(){
    autoCloseable = openMocks( testClass: this);
    //ReflectionUtils.setInternalState(victim, "firmwareUpdateIEService", firmwareUpdateIEService);
}
```

Figure 4.44: setUp function

Upon the completion of a unit test execution, it is necessary to release the allocated resources. To address this, a *close()* function was introduced (as shown in Figure 4.45) to close the mocks and release the associated resources. By utilising the @AfterMethod annotation, the logic within the *close()* function is executed after each individual unit test. This ensures that the state is not preserved between unit tests, preventing any potential interference or inaccurate results when running subsequent tests. Moreover, the use of *openMocks()* in conjunction with AutoCloseable guarantees that the mocks are properly initialised and cleaned up, even in the event of an exception occurring during the test execution. Additionally, a try-with-resources construct was employed for mocking static methods. This approach automatically calls the *close()* method of the AutoCloseable object upon exiting the try-with-resources block, ensuring proper resource management.

```
public void close() throws Exception {
    autoCloseable.close();
    victim = null;
}
```

Figure 4.45: Close function

If the close() method is not invoked, the subsequent unit tests will not be executed properly. This is because the mocks remain registered within the active thread, leading to errors. The problem is depicted in Figure 4.46, highlighting the issue caused by the mocks not being properly released.

```
For com.criticalsoftware.CSWSYS.core.utils.TransactionContext static mocking is already registred in the current thread

To create a new mock, the existing mock registration must be deregistered
        at com.criticalsoftware.CSWSYS.core.webservices.exceptions.ws.rs.NotSupportedExceptionHttpStatusResolverTest.setup
        at java.base/jdk.internal.reflect.NativeMethodAccessorImple.invoke0
```

Figure 4.46: Mock Registred in Thread error

As seen before, the whole project (including CSWSYS component) before the migration, only used Field Injection. After switching to constructor injection a problem derived from TestNG arose. This problem arises when constructor injection and TestNG are used simultaneously.

After switching to Constructor Injection, a bug emerged within the unit tests, specifically related to the creation of test class instances. The state is preserved across multiple test methods, resulting in errors when running the unit tests consecutively. However, the tests execute successfully when run individually.

Mockito recognizes that a instance has already been initialized and attempts to use field injection when the second unit test is executed. This approach works unless the field is declared as final. In the previous example (Figure 4.20), the fields became final when transitioning to constructor injection. Mockito respects constructor injection, meaning it will not modify an object if it has been created using constructor injection. A simple workaround for this "bug" was to assign a null value to the test class instance after executing the unit test (Figure 4.45), using *victim = null;*. This ensures that a new instance of the test class is created for each test, mitigating the issue caused by TestNG's state persistence.

This phase of the work required the most effort as it involved analyzing and modifying all the unit tests accordingly, removing the reflection and maintaning the same objective. After all changes, every unit-test passed successfully. The project's development involving numerous individuals with diverse backgrounds and varying development practices presents challenges in maintaining consistent code quality. Additionally, meeting deadlines sometimes need taking shortcuts to expedite the process, which is where the whitebox functionality becomes useful. In theory, private methods should not be tested directly, but the whitebox's invokeMethod allows for such testing, albeit being considered a suboptimal practice. However, the use of the whitebox can also be viewed from a perspective of "consistency." If the whitebox functionality was already being utilized in the project prior to the current work, it could have influenced subsequent developers to continue using it, maintaining consistency across the codebase.

4.7 Regression System Testing

After implementing all the necessary changes, it was important to validate them through system tests to ensure the proper functioning of the component. The execution of system tests served as regression tests, as the same tests were executed both before and after the changes to ensure that the functionality was maintained.

In the previous section, it was discussed the changes made to the unit tests, which served as a way to validate the implementation of Quarkus. However, unit tests only validate the behaviour of individual code units and do not verify the component as a whole when interacting with other components. On the other hand, system tests are end-to-end tests that verify the entire component from start to finish. These tests are identified by tags representing user stories, indicating that they fulfill the requirements.

As mentioned in the Dependencies Changing sub-section, some components that do not interact with the CSWSYS component were commented out in order to compile the code with Java 11. Consequently, it was not possible to run all the components simultaneously within the IDE. To overcome this limitation, a Docker approach was adopted. A Dockerfile was used to build a Docker container image for the CSWSYS component (in Quarkus), allowing it to be run along with all the other components (in Thorntail) [65].

4.7.1 Test Environment Setup

The first step to run all the system tests was setting up the test environment. Initially, a virtual environment was created within the virtual machine using Python. This new virtual environment had its own pip tool for installing libraries. The following libraries were installed within this virtual environment:

- Python 2.7;
- Robot Framework 4.1.3
- Ruamel.yaml

```
robotframework==4.1.3
ruamel.ordereddict==0.4.15
ruamel.yaml==0.16.13
ruamel.yaml.clib==0.2.2
(venv) brunogandres@brunogandres:~/Desktop/Rep/test$ python --version
Python 2.7.18
```

Figure 4.47: Virtual Environment Libraries

A clone from the git test repository was carried out in order to have all the updated configurations, keys, and scripts in order to be able to execute the system tests of the actual state of the project. These scripts are responsible for getting the latest and most stable version of CSWSYS and later running the containers and populating the testing databases in order to be all set to run the system tests.

Within the project, the approach for testing all the components involved a testing environment where component images were downloaded from a repository and executed within Docker containers. These containers were able to communicate with each other within the same Docker network. The system tests were executed using a script that utilised the Robot Framework. This script executed a

docker-compose file to build the containers with predefined configurations. The CSWSYS component image, which originally used Thorntail, was also present in the repository alongside other component images. Therefore, to execute the CSWSYS component with Quarkus, it was necessary to remove the CSWSYS component field from the docker-compose file and manually execute a docker command to run the locally created image.

From the Quarkus demo [74], a Docker folder containing Dockerfiles was used within the CSWSYS component to define and build the CSWSYS component's docker image. The Dockerfile contains instructions for building the docker image, and the Docker folder was added to the *dist* directory. Additionally, the CSWSYS component utilised the *quarkus-container-image-docker* dependency (listed in Table 4.1), which facilitated the building and generation of the component's docker image.

Given the large number of system tests encompassing all the components, a subset of tests was chosen to be executed. Furthermore, running all the system tests for CSWSYS would require powerful machines, which can be costly. To overcome this limitation, only a subset of tests specific to the CSWSYS component were executed locally. These tests were divided into three modules corresponding to different electricity suppliers, resulting in a total of 310 system tests. The objective of running these system tests was to ensure that they all passed, indicating that the functionality of the CSWSYS component was maintained after migrating to Quarkus. These system tests aimed to assess the connection between the database and the CSWSYS component, as well as the behaviour of receiving valid and invalid service requests with different timeout and expiration values. A detailed report of the results obtained from running the system tests can be found in the appendix B.

4.7.2 Defect Fixing

Several challenges were encountered during the execution of the system tests, including issues with database connections, discrepancies in configurations properties values between the development and testing environments, and variations in expected output logs between Thorntail and Quarkus.

Firstly, regarding the database connection, a different database was used for the testing environment. In the beginning, the same configuration file from the development environment was used, which brought problems. In the development environment, everything was performed on the localhost domain, so the database URL was built with the localhost domain name. However, in the testing environment, the database was running in a Docker container, so the localhost no longer works, as it was necessary to replace it with the Docker network IP. Additionally, the database name in the testing environment differed from that in the development environment, causing connection failures.

The second issue was related to discrepancies in the output logs between Thorntail and Quarkus during the startup of the components. At an early stage of the system tests, the logs were utilized to determine if the containers had started

in order to proceed with the tests. It was expected that there would be some differences between the log outputs of Quarkus and Thorntail. Without the expected log message, the tests would not commence, resulting in the predefined time limits for the components' startup being reached. In the case of Thorntail, as shown in Figure 4.48, a log message from the framework, such as *THORN99999: Thorntail is Ready* was printed after the startup process.

```
The application is now ready on port: 8085
=========================================================================
2023-06-02 10:04:15,770 INFO  [d3c5af2bc777] [o.w.swarm] [main] () THORN99999: Thorntail is Ready
```

Figure 4.48: Thorntail Startup Logs

Within the setup file of the Robot Framework for the CSWSYS component system tests, the expected message log that needed to be waited for was modified. Previously, the expected message log was associated with the startup of the Thorntail application, specifically *THORN99999: Thorntail is Ready*. However, with the transition to Quarkus, the message log to be awaited during the Quarkus application startup was changed (Figure 4.49). Instead of the previous message, the new expected log was *Profile prod activated.* as shown in Figure 4.50.

```
    Start Container  CSWSYS-component
#   Wait Until Container Log Has Message  CSWSYS-component  THORN99999: Thorntail is Ready  seconds_to_timeout=600
    Wait Until Container Log Has Message  CSWSYS-component  Profile prod activated.  seconds_to_timeout=600
```

Figure 4.49: Expected message log change

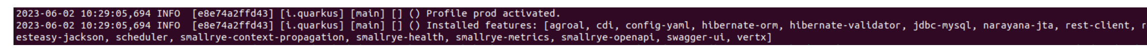


Figure 4.50: Quarkus Message Log example

During the start of the system tests, this verification process continues until either the expected message is found in the container or the timeout limit is reached. The checks are performed at intervals of five seconds, allowing to determine if the container with Quarkus has started or not.

The third error faced was also related to the configuration files of the testing environment. During the execution of the system tests, certain tests were expecting timeout values that were present inside the configuration files in order to test if some service requests took longer than the time defined in the configuration in order to validate or refuse the service request. These timeouts were different from the ones in the testing environment, which resulted in test failures.

The fourth error faced was related with overridable configuration properties within the configuration files thate are used during the build time. One of the system tests aimed to add a service request to the database and wait for the service request to expire according to the defined timeout value present in the configuration file. After passing the expiration date, the service request is removed from the database by another component, and finally, the same service request is added again after the expired service request was removed. The problem was that the

expiration time of the service request had overtaken the specified time to execute the test and remove it from the database by the responsible component. So when checking the database to see if the service request was expired, the wrong timeout value led the clean-up component to never remove the service request from the database because, in that time-box, the service request is still not expired, leading the test to fail since it cannot add a new service request because one is still present in the database (Figure 4.51).

```
REQMNG-SEC - Request Removed After Default Timeout :: This test va.. | FAIL |
Keyword 'Row count Is 0' failed after retrying for 3 minutes 25 seconds. The last error was: Expected zero rows
to be returned from 'SELECT * FROM request' but got rows back. Number of rows returned was 1
```

Figure 4.51: SystemTest Error

Although the configuration file in the test environment was fixed with the correct values, a configuration file in the wrong directory during the component packaging process resulted in a merge between the configuration files, leading to the use of incorrect timeout values. To address this issue, the solution involved modifying the application-side configuration files, as discussed in Section 4.5, separating the build-time configurations into one file (inside *dist* directory) and the overrideable configurations into another (*deploy* directory).

The execution of the system tests also proved to be beneficial as it helped uncover a bug in the code. As mentioned earlier, some system tests were designed to validate the connection between the CSWSYS component and the database. On one hand, the tests aimed to confirm when the connection is UP, and on the other hand, they verified if the health check endpoint returns the correct status when the database check fails. To determine the state of the database connection, a timer was implemented, which periodically executed a query to the database every ten seconds to check for a response. This response indicated whether the database was up or down. In the original implementation, when the Java version was 8, this was achieved using the ScheduleExpression from javax.ejb [67]. However, this approach was commented out during the Dependencies Changing phase, when javax.ejb became deprecated and was not replaced, as shown in Figure 4.52.

```
//    @Resource
//    private TimerService timerService;

//    @PostConstruct
//    private void initSchedule() {
//        final ScheduleExpression scheduleExpression = SchedulerUtils.createScheduleExpression(getCronConfigProperty());
//        timerService.createCalendarTimer(scheduleExpression);
//    }

//    @Timeout
    3 usages   mario +2
    void checkDatabaseConnectionState() {
        getLogger().debug("operation=checkDatabaseConnectionState, message='Starting scheduled job', databaseConnectionAlive={}",
                databaseConnectionAlive);
```

Figure 4.52: Old Check Database Connection function

As a result, the *checkDataBaseConnectionState* method is never invoked, and thus it cannot perform periodic checks. To address this issue, a fix was imple-

mented by leveraging a Quarkus dependency called *quarkus-scheduler* (listed in Table 4.1), which involved annotating the method with @Scheduled. This annotation instructs Quarkus to execute the method every ten seconds. The frequency at which the method needs to run periodically is also defined in the configuration file. Previously, the ScheduleExpression was constructed using a cron expression, which is a string format used to define the schedule for recurring tasks or jobs. Quarkus also supports cron expressions [68], but with a slightly different syntax based on Quartz [69]. Figure 4.53 illustrates the usage of the @Scheduled annotation attached to the *checkDatabaseConnectionState* method, with a cron expression indicating a periodicity of ten seconds. The concurrentExecution flag ensures that the method is never executed concurrently.

```
no usages  brunogandres
@Scheduled(cron = "0/10 * * ? * *",
        concurrentExecution = Scheduled.ConcurrentExecution.SKIP)
void checkDatabaseConnectionStateCopy() {
```

Figure 4.53: Quarkus Scheduled Annotation

The *checkDatabaseConnectionState* method is invoked every ten seconds, performing a database query and updating the connection state based on the response. Figure 4.54 illustrates the behaviour of this periodic task, showing the following steps: 1) The scheduled task is initiated; 2) The query is executed; 3) The scheduled task finishes and updates the database state. Before implementing the fix, if the system test stopped the database container, the connection state would remain "UP" because this scheduled task was never triggered. As a result, the connection state did not reflect the actual state of the database.

```
1 2023-06-02 DEBUG [CSWSYSDatabaseHealthCheckServiceImpl] operation=checkDatabaseConnectionState,
  message='Starting Scheduled Job', databaseConnectionAlive=true
2 2023-06-02 DEBUG [DatabaseHealthCheckTimeoutService] operation=doQuery
3 2023-06-02 DEBUG [CSWSYSDatabaseHealthCheckServiceImpl] operation=checkDatabaseConnectionState,
  message='Finished Scheduled Job', databaseConnectionAlive=true
```

Figure 4.54: Database Connection State Logs

Figure 4.55 illustrates the behaviour when executing the system test and stopping the container with the database.

```
1 The last packet sent successfully to the server was 0 milliseconds ago. The driver
  has not receive any packets from the server.
2 2023-06-02 ERROR [CSWSYSDatabaseHealthCheckServiceImpl] operation=isDatabaseAlive,
  databaseConnectionAlive=false
```

Figure 4.55: Database Connection Down Logs

In this scenario, the connection is lost, no more packets are exchanged, and the state is updated accordingly, aligning with the intended purpose of the test.

These changes resulted in all the tests passing successfully, achieving the objective of preserving functionality after the migration.

4.8 Final considerations

The previous section provided an overview of the changes made and the challenges encountered during the migration of the CSWSYS component from the legacy technology, Thorntail, to Quarkus.

In the initial phase, the focus was on replacing the dependencies from Thorntail with corresponding Quarkus dependencies (Section 4.3). To ensure maintainability and facilitate potential future migrations to different technologies, decisions were made to avoid excessive reliance on the Quarkus framework itself. Instead, more generalized dependencies were used, allowing compatibility with other frameworks.

Once the dependencies were updated, the attention turned to code modifications, specifically addressing EJBs, annotations, and unit tests (Section 4.4). The majority of effort was dedicated to adjusting the unit tests, as some had to be rewritten due to the removal of deprecated practices such as PowerMock and Reflection. Ensuring the successful execution of unit tests without any failures was crucial for validating the functionality of the component. Additionally, configuration properties were migrated to the Quarkus format as part of the configuration adjustments (Section 4.5).

These code changes as well as some code improvements (Field Injection to Constructor Injection, Reflection remove) resulted in an increase in the number of lines in the source code. Table 4.2 represents the static analysis conducted before the migration, while Table 4.3 showcases the static analysis conducted after the migration and code modifications.

Table 4.2: Thorntail CSWSYS component static analysis

	Classes	**Unit Tests**	**Lines**	**Source Code Lines**	**Source Code Lines [%]**	**Comment Lines**	**Blank Lines**
CSWSYS Component	88	103	5720	3782	66%	944	994

Table 4.3: Quarkus CSWSYS component static analysis

	Classes	**Unit Tests**	**Lines**	**Source Code Lines**	**Source Code Lines [%]**	**Comment Lines**	**Blank Lines**
CSWSYS Component	88	103	6054	3952	65%	1020	1082

The increase in the number of lines, while not substantial, can be primarily attributed to the conversion from field injection to constructor injection and the elimination and replacement of the whitebox functionality. The use of constructor injection and the removal of direct method invocations with whitebox resulted in a slight increase in the number of lines. However, no new classes or unit tests were created, and the overall value of the code remained unchanged.

In addition to unit testing, the successful execution of existing system tests was another critical validation criterion for the migration. These end-to-end tests

were designed based on the requirements and served as regression tests, ensuring that the CSWSYS component operated correctly. A subset of relevant system tests was selected for the CSWSYS component, and after resolving the identified issues, all tests were successfully executed.

Table 4.4 represents an overview regarding the file changes that happened. These values were derived by comparing the state of the project before any changes occurred to its state after all changes had been implemented (git commits diff).

Table 4.4: Overall CSWSYS Changes statistic

	Files Changed	**Insertions**	**Deletions**	**Java Files Changed**	**XML Files Changed**	**YAML Files Added**
Overall Changes	191 files	5450	2338	96	46	2

The XML files that were changed are related to the dependency change phase, specifically involving the Thorntail, old Java and Quarkus dependencies. The Java files used by the CSWSYS component were modified. These files were not exclusively associated with the CSWSYS component, they also belong to the modules shared by other components. As mentioned in the section regarding configuration files, the required changes involved converting the files from Thorntail format to Quarkus format. Two additional files were added to the project, regarding this changes. No additional methods or classes were added, as the complexity of the CSWSYS component remained unchanged.

Throughout the completion of the planned tasks, valuable lessons were learned. It became apparent that a different approach could have been employed. Firstly, deleting all unnecessary POM files would have improved the organization and efficiency of the dependency change process. Secondly, focusing on identifying and commenting only the specific sub-modules used by the CSWSYS component would have eliminated the need to modify unnecessary modules initially. In addition, the utilization of libraries like *lombok* [79] could have been beneficial, as they enable the use of annotations that help reduce the number of lines for constructors, getters, and setters level. This not only makes the code cleaner but also enhances readability.

Another important lesson learned was the importance of thorough documentation throughout the maintenance work. It is crucial to document bug fixes, code enhancements, and any findings made during the maintenance phase. Comprehensive documentation facilitates tracking of implemented changes and serves as a valuable reference for future maintenance activities. Inaccurate or incomplete documentation caused delays during the internship, highlighting the critical role of accurate documentation.

Chapter 5

Conclusion

The objective of this internship was to migrate a component (the CSWSYS component) of a larger system (CSWSYS) from an end-of-life technology, Thorntail, to a more recent one, Quarkus. Using legacy technologies such as Thorntail may lead to vulnerabilities, security weaknesses, and incompatibilities. Maintaining a software system prevents future problems and allows the software to stay updated and functional over time.

In order to address this maintenance task, we adhered to a defined approach based on the software maintenance process (sub-section 2.2.1). Initially, an analysis of the entire CSWSYS project's structure and organisation, with a focus on the CSWSYS component's source code, was conducted. The second phase concentrated on implementing modifications and improvements to enhance the code's maintainability. The final phase involved reviewing and validating the maintenance improvements via system tests, which are end-to-end tests performed on the CSWSYS component in a testing environment, to ensure its proper functioning. Successful execution of both unit tests and regression system tests was used as validation for the migration.

It is important to note that this work served as a proof-of-concept for a specific use case and was conducted under specific time constraints, which allowed only a subset of the CSWSYS component's regression system tests to be run. Due to these constraints, we were unable to execute performance and load tests to compare its behaviour against Thorntail. This limitation, which was identified as a risk at the onset of the project, led to the activation of a mitigation plan, making these tests out of scope. Therefore, it was not possible to validate the performance metrics discussed in sub-section 2.5.5 within the project timeframe, making it a task for future consideration. Performance and load tests are crucial for ensuring software component reliability, stability, and efficiency. Nonetheless, the primary objective of the internship was fulfilled: the CSWSYS component successfully runs on Quarkus, within the specified plan.

In conclusion, this document provides a detailed account of the changes, strategies, practices, and challenges encountered during this maintenance project, offering a realistic view of the task. The internship provided valuable insight into the complexities of maintaining a legacy system and the overall maintenance pro-

cess.

The findings could serve as a valuable resource if the client decides to convert the remaining components to Quarkus. As future work, some tasks and assessments remain to be carried out in order to assess the effort required to migrate all the project based on the necessary changes that were made to the CSWSYS component. The unit tests is the phase where more effort will be required as it is necessary to remove all the reflection. The effort needs to be weighted according to the skill and experience of the people who will carry out the remaining components migration and according the size of each component in order to get an accurate estimate. The documentation of the changes made reduces the effort required to search for alternatives to Thorntail, as many changes will be similar to those made to the CSWSYS component. Additionally, the migration of modules shared among other components eliminates the need for additional efforts associated with those modules. As previously mentioned, performance and load tests, as well as security assessments, need to be executed in order to present guarantees of migration.

References

[1] *What Makes The Software Maintenance So Important?*, https://apibest.com/blog/what-makes-software-maintenance-so-important. Accessed: 2022-12.

[2] *Who we are?*, https://www.smartdcc.co.uk/about-dcc/who-we-are/. Accessed: 2022-12.

[3] *Smart Metering Implementation Programme [SMIP]*, https://smartenergycodecompany.co.uk/glossary/smart-metering-implementation-programme/. Accessed: 2022-09.

[4] *What is SMETS1 and SMETS2 smart meter?*, https://www.ugp.co.uk/support/faqs/smart-meters/what-is-smets1-and-smets2-smart-meter/. Accessed: 2022-09.

[5] *Bourque and R.E. Fairley, eds., Guide to the Software Engineering Body of Knowledge, Version 3.0, IEEE Computer Society, 2014; www.swebok.org.*

[6] *ISO/IEC/IEEE 14764:2006, Software Engineering — Software Life Cycle Processes — Maintenance*

[7] *ISO/IEC/IEEE 12207:2017, Systems and software engineering — Software life cycle processes*

[8] *A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," in Computer, vol. 28, no. 8, pp. 44-55, Aug. 1995, doi: 10.1109/2.402076.*

[9] *How to Plan for Software Maintenance*, https://medium.com/swlh/types-of-software-maintenance-2b0503848b43. Accessed: 2022-11.

[10] *Kontogiannis K, Techniques for Software Maintenance (2011)*

[11] *Muller H, Reps T, Snelting G, Program Comprehension and Software Reengineering*

[12] *Software Engineering | Reverse Engineering*, https://www.geeksforgeeks.org/software-engineering-reverse-engineering/. Accessed: 2023-11.

[13] *ISO/IEC FDIS 9126-1:2000, Information technology — Software product quality - Quality model*

[14] *P. Grubb and A.A. Takang, Software Maintenance: Concepts and Practice, 2nd ed., World Scientific Publishing, 2003.*

[15] *R.D. Banker, S.M.Datar and D. Zweig, SOFTWARE COMPLEXITY AND MAINTAINABILITY, 1989*

[16] *Understandability: The Most Important Metric You're Not Tracking*, https://www.pagerduty.com/eng/what-is-software-understandability/. Accessed: 2023-01.

[17] *Understandability: The Most Important Metric You're Not Tracking*, https://www.infoq.com/articles/understandability-metric-not-tracking/. Accessed: 2023-01.

[18] *Tarwani, Sandhya and Anuradha Chug. "Agile Methodologies in Software Maintenance: A Systematic Review." Informatica (Slovenia) 40 (2016)*

[19] *Heeager, L.T., Rose, J. Optimising agile development practices for the maintenance operation: nine heuristics. Empir Software Eng 20, 1762–1784 (2015).*

[20] *F. u. Rehman, B. Maqbool, M. Q. Riaz, U. Qamar and M. Abbas, "Scrum Software Maintenance Model: Efficient Software Maintenance in Agile Methodology," 2018 21st Saudi Computer Society National Computer Conference (NCC), 2018, pp. 1-5, doi: 10.1109/NCG.2018.8593152.*

[21] *Non-functional Requirements: Examples, Types, How to Approach*, https://www.altexsoft.com/blog/non-functional-requirements/. Accessed: 2022-10.

[22] *CS 410/510 - Software Engineering - System Dependability*, https://cs.ccsu.edu/~stan/classes/CS410/Notes16/10-SystemDependability.html. Accessed: 2022-10.

[23] *What is Scrum?*, https://www.scrum.org/resources/what-is-scrum. Accessed: 2022-12.

[24] *Scrum Project Management: Advantages and Disadvantages*, https://www.simplilearn.com/scrum-project-management-article. Accessed: 2022-12.

[25] *Kanban vs Scrum vs Scrumban: What Are The Differences?*, https://ora.pm/blog/scrum-vs-kanban-vs-scrumban/. Accessed: 2022-12.

[26] *What is kanban?*, https://www.atlassian.com/agile/kanban. Accessed: 2022-12.

[27] *What is a kanban board?*, https://www.atlassian.com/agile/kanban/boards. Accessed: 2022-12.

[28] *What are WIP limits?*, https://www.atlassian.com/agile/kanban/wip-limits. Accessed: 2022-12.

[29] *Kanban vs. Scrum: A simple breakdown of each complex methodology*, https://www.teamwork.com/blog/kanban-vs-scrum/. Accessed: 2022-12.

[30] *KANBAN VS. SCRUM: WHAT ARE THE DIFFERENCES?*, https://www.planview.com/resources/guide/introduction-to-kanban/kanban-vs-scrum/. Accessed: 2022-12.

[31] *Understanding the Kanban Guide for Scrum Teams*, https://www.scrum.org/resources/blog/understanding-kanban-guide-scrum-teams. Accessed: 2022-12.

[32] *What is Scrumban? The Best Parts of Scrum and Kanban*, https://www.businessprocessincubator.com/content/what-is-scrumban-the-best-parts-of-scrum-and-kanban/. Accessed: 2022-12.

[33] *M. O. Ahmad, P. Kuvaja, M. Oivo and J. Markkula, "Transition of Software Maintenance Teams from Scrum to Kanban," 2016 49th Hawaii International Conference on System Sciences (HICSS), 2016, pp. 5427-5436, doi: 10.1109/HICSS.2016.670.*

[34] *What is a tech stack? Technology stack in a nutshell*, https://dac.digital/what-is-a-tech-stack-technology-stack-in-a-nutshell/. Accessed: 2022-12.

[35] *What is a Tech Stack? Choosing What Goes In Yours*, https://www.heap.io/topics/what-is-a-tech-stack. Accessed: 2022-12.

[36] *Red Hat build of Thorntail*, https://access.redhat.com/products/thorntail. Accessed: 2022-11.

[37] *The End of an Era*, https://thorntail.io/posts/the-end-of-an-era/. Accessed: 2022-11.

[38] *What is Quarkus?*, https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus. Accessed: 2022-11.

[39] *What is Quarkus?*, https://www.ionos.com/digitalguide/server/configuration/what-is-quarkus/. Accessed: 2022-11.

[40] *Container First*, https://quarkus.io/container-first/. Accessed: 2022-11.

[41] *McCluskey G. (1998, January) - Using Java Reflection*, https://www.oracle.com/technical-resources/articles/java/javareflection.html. Accessed: 2022-11.

[42] *Quarkus vs Spring Boot: Which Framework is Right for You.*, https://rollbar.com/blog/quarkus-vs-spring-boot/. Accessed: 2022-11.

[43] *Quarkus vs Spring boot Pros and cons.*, https://techbriel.com/quarkus-vs-spring-boot-pros-and-cons/. Accessed: 2022-11.

[44] *Pros and Cons of Using Spring Boot*, https://bambooagile.eu/insights/pros-and-cons-of-using-spring-boot/. Accessed: 2022-11.

[45] *What is Java Spring Boot?*, https://www.ibm.com/topics/java-spring-boot. Accessed: 2022-11.

[46] *Advantages of Spring Boot*, https://www.adservio.fr/post/advantages-of-spring-boot. Accessed: 2022-11.

[47] *Spring Boot vs Quarkus*, https://www.baeldung.com/spring-boot-vs-quarkus. Accessed: 2022-11.

[48] *Difference Between Java EE and Spring: Which Framework is the Best Choice?*, https://anywhere.epam.com/business/spring-vs-java-ee. Accessed: 2022-11.

[49] *Tomcat vs JBoss – What's the Difference? (Pros and Cons)*, https://cloudinfrastructureservices.co.uk/tomcat-vs-jboss-whats-the-difference/. Accessed: 2022-11.

[50] *Thoughts on Quarkus*, https://blog.sebastian-daschner.com/entries/thoughts-on-quarkus. Accessed: 2022-11.

[51] *SOA VS MICROSERVICES: WHAT'S THE DIFFERENCE?*, https://www.crowdstrike.com/cybersecurity-101/cloud-security/soa-vs-microservices/. Accessed: 2022-10.

[52] *Risk Matrix*, https://www.wrike.com/blog/what-is-risk-matrix/#Risk-matrix-example.

[53] *Bitbucket*, https://en.wikipedia.org/wiki/Bitbucket. Accessed: 2023-05.

[54] *Introduction to PowerMock*, https://www.baeldung.com/intro-to-powermock. Accessed: 2023-05.

[55] *Smallrye Metrics*, https://quarkus.io/guides/smallrye-metrics. Accessed: 2023-05.

[56] *Smallrye Health*, https://quarkus.io/guides/smallrye-health. Accessed: 2023-05.

[57] *MicroProfile Health*, https://download.eclipse.org/microprofile/microprofile-health-2.1/microprofile-health-spec.html. Accessed: 2023-05.

[58] *Startup*, https://docs.oracle.com/javaee/6/api/javax/ejb/Startup.html. Accessed: 2023-05.

[59] *Startup Event*, https://quarkus.io/guides/cdi-reference#startup_event. Accessed: 2023-05.

[60] *Listening for startup and shutdown events*, https://quarkus.io/guides/lifecycle#listening-for-startup-and-shutdown-events. Accessed: 2023-05.

[61] *PowerMock - GitHub*, https://github.com/powermock/powermock. Accessed: 2023-05.

[62] *Mockito*, https://site.mockito.org/. Accessed: 2023-05.

[63] *The Difference Between mockito-core and mockito-all*, https://www.baeldung.com/mockito-core-vs-mockito-all. Accessed: 2023-05.

[64] *Setting up and configuring Hibernate ORM*, https://quarkus.io/guides/hibernate-orm#setting-up-and-configuring-hibernate-orm. Accessed: 2023-05.

[65] *Containerize an application*, https://docs.docker.com/get-started/02_our_app/. Accessed: 2023-05.

[66] *Building*, https://quarkus.io/guides/container-image#building. Accessed: 2023-05.

[67] *ScheduleExpression*, https://docs.oracle.com/javaee/6/api/javax/ejb/ScheduleExpression.html. Accessed: 2023-05.

[68] *Updating the application configuration file*, https://quarkus.io/guides/scheduler#updating-the-application-configuration-file. Accessed: 2023-05.

[69] *CronTrigger Tutorial*, https://www.quartz-scheduler.org/documentation/quartz-2.3.0/tutorials/crontrigger.html. Accessed: 2023-05.

[70] *All configuration options*, https://quarkus.io/guides/all-config. Accessed: 2023-05.

[71] *Bean Discovery*, https://quarkus.io/guides/cdi-reference#bean_discovery. Accessed: 2023-05.

[72] *Singleton Session Bean*, https://www.baeldung.com/java-ee-singleton-session-bean. Accessed: 2023-05.

[73] *@ApplicationScoped and @Singleton look very similar. Which one should I choose for my Quarkus application?*, https://quarkus.io/guides/cdi#applicationscoped-and-singleton-look-very-similar-which-one-should-i-choose-fo
Accessed: 2023-05.

[74] *CREATING YOUR FIRST APPLICATION*, https://quarkus.io/guides/getting-started. Accessed: 2023-05.

[75] *Where Is the Maven Local Repository?*, https://www.baeldung.com/maven-local-repository. Accessed: 2023-05.

[76] *JDK 11 Release Notes*, https://www.oracle.com/java/technologies/javase/11-relnote-issues.html#JDK-8190378. Accessed: 2023-05.

[77] *Just In Time Compiler*, https://www.geeksforgeeks.org/just-in-time-compiler/. Accessed: 2023-05.

[78] *SonarQube Documentation*, https://docs.sonarqube.org/latest/. Accessed: 2023-05.

[79] *Project Lombok*, https://projectlombok.org/. Accessed: 2023-05.

[80] *Managing Code Complexity*, https://devguide.trimble.com/development-practices/managing-code-complexity/. Accessed: 2023-05.

Appendices

Appendix A

Show and Tell

In this section, the Shows and Tells presentations that were carried out during the internship are available.

Critical software
1st Show and Tell
CSWSYS component–
Quarkus
Bruno Gandres
Date: 10/07/2023
CRITICALSOFTWARE.COM
© Copyright Critical Software. All rights reserved.
Information Classification: Confidential
1

Agenda

- Overview
- Developed Work

© Copyright Critical Software. All rights reserved.

Overview

- Request Manager Migration: From Thorntail to Quarkus;
- Java 11;
- Quarkus 2.16.1 Final;
- The work carried out so far has focused on:
  - Dependencies;
  - CDI;
  - Some adaptations in unit tests.

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

- Since the other components were in java 8 and the request manager was in java 11, there were problems compiling the components all together;
- Components that were not required were commented on in the root pom.

```
<modules>
    <module>core</module>
<!--    <module>crypto</module>-->
    <module>request-manager</module>
<!--    <module>command-manager</module>-->
<!--    <module>configuration-manager</module>-->
    <module>smki-manager</module>
    <module>smki-web-client</module>
    <module>detect</module>
    <module>dist</module>
    <module>data</module>
<!--    <module>migration</module>-->
<!--    <module>migration-sftp-client</module>-->
    <module>smki-web-server</module>
<!--    <module>clean-up-scheduler</module>-->
</modules>
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Dependencies

- Javaee and Throntail's dependencies were commented;
- Quarkus can use SmallRye Health which is an implementation of the MicroProfile Health specification. SmallRye Metrics allows applications to gather metrics and statistics;
- @Health was changed to @Readiness.

```
    <!-- Health check -->
        <!-- Old -->
<!--    <dependency>-->
<!--        <groupId>io.thorntail</groupId>-->
<!--        <artifactId>microprofile-health</artifactId>-->
<!--    </dependency>-->

    <!-- New -->
    <dependency>
        <groupId>org.eclipse.microprofile.health</groupId>
        <artifactId>microprofile-health-api</artifactId>
    </dependency>
```

```
    <!-- Thorntail Microprofile Metrics -->

    <!-- Old -->
<!--    <dependency>-->
<!--        <groupId>io.thorntail</groupId>-->
<!--        <artifactId>microprofile-metrics</artifactId>-->
<!--    </dependency>-->

    <!-- New -->
    <dependency>
        <groupId>org.eclipse.microprofile.metrics</groupId>
        <artifactId>microprofile-metrics-api</artifactId>
    </dependency>
```

```
<!-- Java EE 7 dependency -->
<!-- Old -->
<!--        <dependency>-->
<!--            <groupId>javax</groupId>-->
<!--            <artifactId>javaee-api</artifactId>-->
<!--        </dependency>-->
```

© Copyright Critical Software. All rights reserved.

Work Developed

CDI

- Quarkus needs a Jandex index to detect annotated classes;
- Included jandex-maven-plugin to enable the DCI bean discovery.

```
<build>
  <plugins>
    <plugin>
      <groupId>io.smallrye</groupId>
      <artifactId>jandex-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

```
<plugin>
  <groupId>io.smallrye</groupId>
  <artifactId>jandex-maven-plugin</artifactId>
  <version>3.0.5</version>
  <executions>
    <execution>
      <id>make-index</id>
      <goals>
        <goal>jandex</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Annotations

- @Startup
  - Startup annotation was removed;
  - There was an option to declare an observer of the StartupEvent or annotate the bean with @io.quarkus.runtime.Startup (synthetic observer of StartupEvent is generated).

```
no usages   brunogandres
protected void startUp(@Observes @Initialized(ApplicationScoped.class) final Object toStart) {
    LOG.debug("operation=startUp, start={}", toStart);
}
```

© Copyright Critical Software. All rights reserved.

Work Developed

Annotations

- @Stateless

```
//@Stateless
Tiago Manso +8
@ApplicationScoped
@Transactional
```

- @TransactionAttribute

```
//@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
@Transactional(Transactional.TxType.NOT_SUPPORTED)
```

•To avoid the usage reflection, Quarkus recommends not using private members in your beans.

```
@Inject
//private RequestDAO requestDAO;
RequestDAO requestDAO;
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Database

- Add JDBC driver dependency;
- With Hibernate-Orm, the file persistence.xml is not necessary.

```xml
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-hibernate-orm</artifactId>
</dependency>
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-jdbc-mysql</artifactId>
</dependency>
```

```
[ERROR] Failed to execute goal io.quarkus:quarkus-maven-plugin:2.16.1.Final:build (default) on project dco-request-manager-shared: Failed to build quarkus application: io.quarkus.builder.BuildException: Build failure: Build failed due to errors
[ERROR]         [error]: Build step io.quarkus.agroal.deployment.AgroalProcessor#build threw an exception: io.quarkus.runtime.configuration.ConfigurationException: Unable to find a JDBC driver corresponding to the database kind 'mysql' for the datasource 'dco-ds'. Eith
er provide a suitable JDBC driver extension, define the driver manually, or disable the JDBC datasource by adding 'quarkus.datasource.dco-ds.jdbc=false' to your configuration if you don't need it.
[ERROR]         at io.quarkus.agroal.deployment.AgroalProcessor.resolveDriver(AgroalProcessor.java:353)
[ERROR]         at io.quarkus.agroal.deployment.AgroalProcessor.getAggregatedConfigBuildItems(AgroalProcessor.java:321)
[ERROR]         at io.quarkus.agroal.deployment.AgroalProcessor.build(AgroalProcessor.java:87)
[ERROR]         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
[ERROR]         at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
[ERROR]         at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
[ERROR]         at java.base/java.lang.reflect.Method.invoke(Method.java:566)
[ERROR]         at io.quarkus.deployment.ExtensionLoader$3.execute(ExtensionLoader.java:909)
[ERROR]         at io.quarkus.builder.BuildContext.run(BuildContext.java:281)
[ERROR]         at org.jboss.threads.ContextHandler$1.runWith(ContextHandler.java:18)
[ERROR]         at org.jboss.threads.EnhancedQueueExecutor$Task.run(EnhancedQueueExecutor.java:2449)
[ERROR]         at org.jboss.threads.EnhancedQueueExecutor$ThreadBody.run(EnhancedQueueExecutor.java:1478)
[ERROR]         at java.base/java.lang.Thread.run(Thread.java:829)
[ERROR]         at org.jboss.threads.JBossThread.run(JBossThread.java:501)
[ERROR] -> [Help 1]
[ERROR]
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Unit Tests

- Powermock has some issues with Java 11;
- Attempted to replace powermock-api-mockito with powermock-api-mockito2, but errors started appearing;
- It is not good practice to use powermock, so all powermock dependencies were commented/removed;
- Powermockito was replaced with Mockito;
- Whitebox was removed.

```
<!--    <dependency>-->
<!--      <groupId>org.powermock</groupId>-->
<!--      <artifactId>powermock-api-mockito</artifactId>-->
<!--      <scope>test</scope>-->
<!--    </dependency>-->
<!--    <dependency>-->
<!--      <groupId>org.powermock</groupId>-->
<!--      <artifactId>powermock-module-testng</artifactId>-->
<!--      <scope>test</scope>-->
<!--    </dependency>-->
```

```
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-core</artifactId>
  <version>5.1.1</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.mockito</groupId>
  <artifactId>mockito-inline</artifactId>
  <version>5.1.1</version>
  <scope>test</scope>
</dependency>
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Unit Tests

- With the new version of mockito some adaptations have been made:
  - org.mockito.Matchers → org.Mockito. ArgumentMatchers
  - PowerMockito.mockStatic → Mockito.mockStatic
  - MockitoAnnotations.initMocks → MockitoAnnotations.openMocks

```
//import static org.mockito.Matchers.any;
//import static org.mockito.Matchers.eq;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.eq;
```

```
import static org.mockito.MockitoAnnotations.openMocks;
//import static org.mockito.MockitoAnnotations.initMocks;
```

```
import static org.mockito.Mockito.mockStatic;
//import static org.powermock.api.mockito.PowerMockito.mockStatic;
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Unit Tests

- openMocks returns a closable to close when completing any test.
- The mockStatic creates a thread-local mock controller for all static methods. Should be closed or the mock will remain active on the current thread, showing an error.

```
@BeforeMethod
public void setUp() throws NoSuchFieldException, IllegalAccessException {

    autoCloseable = openMocks( testClass: this);
    //mockStatic(ConfigProviderResolver.class);
    //when(ConfigProviderResolver.instance()).thenReturn(null);

    configProviderResolverMockedStatic = mockStatic(ConfigProviderResolver.class);
    configProviderResolverMockedStatic.when(ConfigProviderResolver::instance).thenReturn( value: null);
```

```
@AfterMethod
public void close() throws Exception {
    autoCloseable.close();
    configProviderResolverMockedStatic.close();
```

© Copyright Critical Software. All rights reserved.

Some problems

- Missing dependencies;
- Missing implementations derived from recent/old versions;
- Difficulties with Mockito;

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

THANK YOU!

CRITICALSOFTWARE.COM

© Copyright Critical Software. All rights reserved.

Figure A.1: 1st Show and Tell

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Agenda

- Overview
- Developed Work

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Overview

- 1st Show and Tell (23/03/2023):
  - Thorntail's dependencies deleted;
  - Quarkus dependencies added;
  - Powermock to Mockito.

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Overview

- Reflection removal;
- Constructor Injection;

- **Milestones:**
    - 28/03/2023 - Request Manager runs on Quarkus;
    - 30/03/2023 - Request Manager passes SOAP UI tests;
    - 13/04/2023 - Reflection removal;
    - 18/04/2023 - Replacing Field Injection with Constructor Injection.

© Copyright Critical Software. All rights reserved.
Information Classification: Confidential

Work Developed

Configurations

- The confluence page for Quarkus configurations, indicated to put the config path in the VM options, which led to IDE running the command with config first and then with jar, failing to run quarkus-jar.

Confluence example:

```
Configuration   Code Coverage   EnvFile   Logs
Path to JAR:            /home/jpcavaleiro/bitbucket/dco/dist/command-manager/ie/target/dco-command-manager-ie-runner.jar
VM options:             -Ddco.config.path=/home/jpcavaleiro/bitbucket/dco/deploy/local/dco-command-manager-ie/config/application.yaml
Program arguments:
Working directory:      /home/jpcavaleiro/bitbucket/dco/dist/command-manager/ie/target
Environment variables:
☐ Redirect input from:
JRE:                    Default (<no JRE> - module not specified)
Search sources using module's classpath:  <whole project>
```

```
Caused by: java.io.FileNotFoundException Create breakpoint : /home/brunogandres/Desktop/Rep/dco/deploy/local/dco-request-manager/config/application.yaml/keystores/smki-manager/trust-keystore.jks
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
    at java.base/sun.net.www.protocol.file.FileURLConnection.connect(FileURLConnection.java:86)
    at java.base/sun.net.www.protocol.file.FileURLConnection.getInputStream(FileURLConnection.java:184)
    at java.base/java.net.URL.openStream(URL.java:1165)
    at com.criticalsoftware.dco.smkimanager.business.KeyStoreServiceImpl.init(KeyStoreServiceImpl.java:64)
```

© Copyright Critical Software. All rights reserved.

Work Developed

Configurations

- The confluence page for Quarkus configurations, indicated to put the config path in the VM options, which led to IDE running the command with config first and then with jar, failing to run quarkus-jar.

Confluence example:

Configuration | Code Coverage | EnvFile | Logs

Path to JAR:	/home/jpcavaleiro/bitbucket/dco/dist/command-manager/ie/target/dco-command-manager-ie-runner.jar
VM options:	-Ddco.config.path=/home/jpcavaleiro/bitbucket/dco/deploy/local/dco-command-manager-ie/config/application.yaml
Program arguments:	
Working directory:	/home/jpcavaleiro/bitbucket/dco/dist/command-manager/ie/target
Environment variables:	
☐ Redirect input from:	
JRE:	Default (<no JRE> - module not specified)
Search sources using module's classpath:	<whole project>

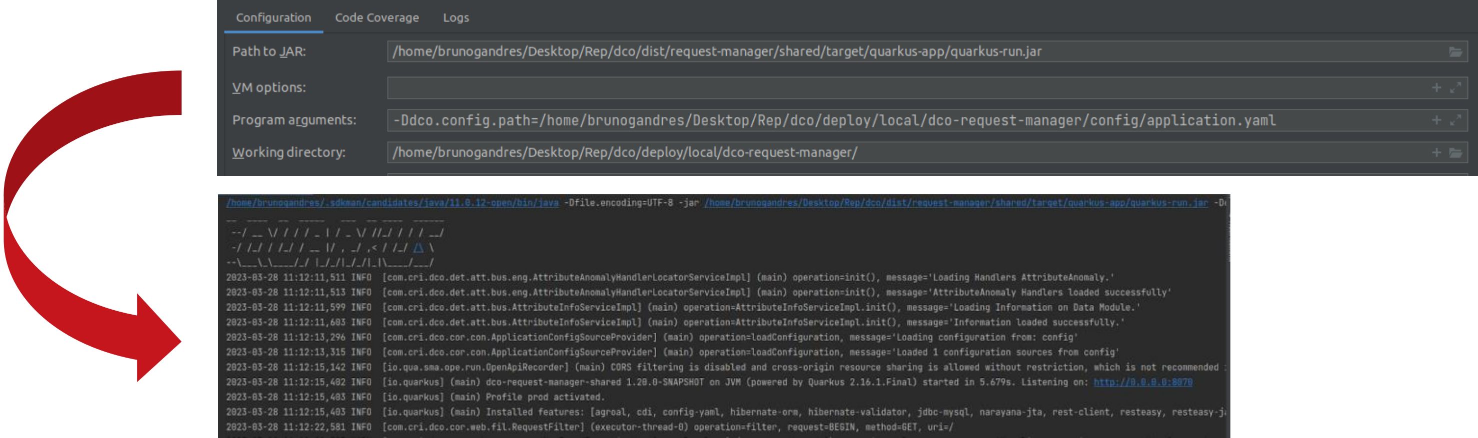
```
Caused by: java.io.FileNotFoundException Create breakpoint : /home/brunogandres/Desktop/Rep/dco/deploy/local/dco-request-manager/config/application.yaml/keystores/smki-manager/trust-keystore.jks
    at java.base/java.io.FileInputStream.open0(Native Method)
    at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
    at java.base/java.io.FileInputStream.<init>(FileInputStream.java:112)
    at java.base/sun.net.www.protocol.file.FileURLConnection.connect(FileURLConnection.java:86)
    at java.base/sun.net.www.protocol.file.FileURLConnection.getInputStream(FileURLConnection.java:184)
    at java.base/java.net.URL.openStream(URL.java:1165)
    at com.criticalsoftware.dco.smkimanager.business.KeyStoreServiceImpl.init(KeyStoreServiceImpl.java:64)
```

© Copyright Critical Software. All rights reserved.

Work Developed

Configurations

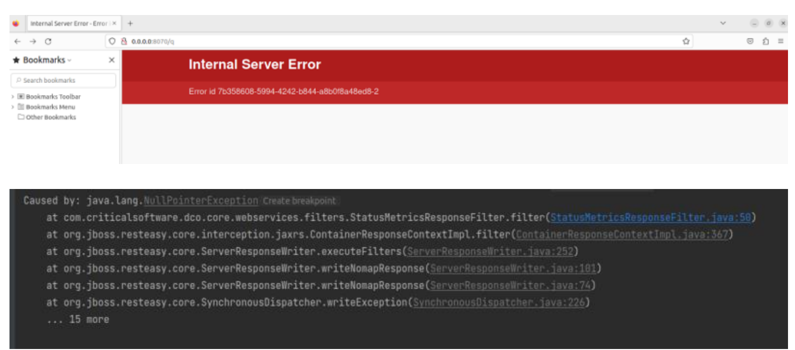
- The config path must be passed as a **Program Argument;**
- In the execution command, the jar path appears first and then the configuration path.

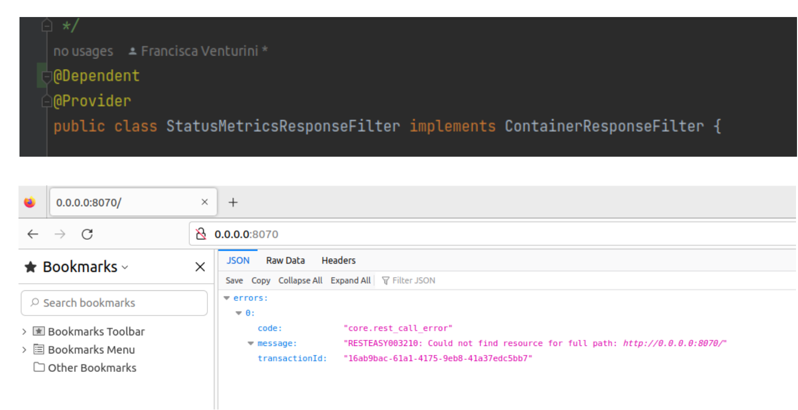


© Copyright Critical Software. All rights reserved.

Work Developed

- Some classes had no annotation, which did not allow Jandex to do the discovery.





© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Constructor Injection

- Field Injection was replaced with Constructor Injection;
- It was a suggestion from the first show and tell;
- Injected fields can be declared as final, which helps with robustness.

```
@Inject
ServiceRequestValidationService serviceRequestValidationService;
2 usages
@Inject
RequestService requestService;
2 usages
@Inject
SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService;
2 usages
@Inject
@TimeoutConfigurationDefault
SrvTimeoutConfiguration srvTimeoutConfiguration;
```

→

```
@Inject
public RequestManagerServiceImpl(final ServiceRequestValidationService serviceRequestValidationService,
                                 final RequestService requestService,
                                 final SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService,
                                 final @TimeoutConfigurationDefault SrvTimeoutConfiguration srvTimeoutConfiguration) {
    this.serviceRequestValidationService = serviceRequestValidationService;
    this.requestService = requestService;
    this.specificProcessingSRVLocatorService = specificProcessingSRVLocatorService;
    this.srvTimeoutConfiguration = srvTimeoutConfiguration;
}
```

© Copyright Critical Software. All rights reserved.
Information Classification: Confidential

Work Developed

Reflection Removal

- Removal of **invokeMethods:**
  - Work around with public methods;
  - Change private methods to package-private.

```java
@Test
public void testCreateFirmwareUpdateImageData() throws Throwable {
    //assertEquals(invokeMethod(victim, "createFirmwareUpdateImageData", image), createFirmwareUpdateImage());
    when(firmwareUpdateIEService.contains(null)).thenReturn( value: false);

    victim.imageParsingAndStoring( imageHash: null, image);

    verify(firmwareUpdateIEService, times( wantedNumberOfInvocations: 1)).write(createFirmwareUpdateImage());
}
```

```java
public void imageParsingAndStoring(final String imageHash, final byte[] image) throws FirmwareUpdateImageException {
    LOG.info("operation=imageParsingAndStoring, s1sp={}, imageHash={}", getS1sp(), imageHash);
    if (!firmwareUpdateIEService.contains(imageHash)) {
        final FirmwareUpdateIe firmwareUpdateIE = createFirmwareUpdateImageData(image);
        firmwareUpdateIE.setImageHash(imageHash);
        firmwareUpdateIEService.write(firmwareUpdateIE);
    }
}
```

```java
public void testBlocksSelection(final byte[] secureGbcsManufacturerImage, final byte[] selectedBlocks, final boolean result)
    throws Throwable {
    if (result) {
        //assertEquals(selectedBlocks, invokeMethod(victim, "chooseBlocks", secureGbcsManufacturerImage));
        assertEquals(selectedBlocks, victim.chooseBlocks(secureGbcsManufacturerImage));
```

```java
//private byte[] chooseBlocks(final byte[] secureGbcsManufacturerImage) throws FirmwareUpdateImageException {
3 usages  jlbarreiro +2
byte[] chooseBlocks(final byte[] secureGbcsManufacturerImage) throws FirmwareUpdateImageException {
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Reflection Removal

- Removal of **invokeMethods:**
    - Replacing Private @PostConstructs to package-private/protected.

```
@PostConstruct
//private void init() {
protected void init() {
    LOG.info("operation=AttributeInfoServiceImpl.init(), message='Loading Information on Data Module.'");
```

```
//Whitebox.invokeMethod(dataModule, "init");
dataModule.init();
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Reflection Removal

- Removal of **setInternalState:**
  - Mockito tries to inject mocked dependencies using one of the three approaches:
    - Constructor Injection
    - Setter Injection
    - Field Injection
  - With the constructor injection, the arguments are resolved with mocks declared in the test only.

```
//victim = new FirmwareUpdateImageSecureStrategyImpl();
//ReflectionUtils.setInternalState(victim, "firmwareUpdateSecureService", firmwareUpdateSecureService);
```

```
//Whitebox.setInternalState(victim, "batchSizeFirmwareUpdateDevices", 50000);
victim.batchSizeFirmwareUpdateDevices = 50000;
```

© Copyright Critical Software. All rights reserved.
Information Classification: Confidential

Work Developed

Constructor Injection - Findings

- The annotations need to be inserted in the constructor.

```java
@Inject
public RequestManagerServiceImpl(final ServiceRequestValidationService serviceRequestValidationService,
                                 final RequestService requestService,
                                 final SpecificProcessingSRVLocatorService specificProcessingSRVLocatorService,
                                 final @TimeoutConfigurationDefault SrvTimeoutConfiguration srvTimeoutConfiguration) {
    this.serviceRequestValidationService = serviceRequestValidationService;
    this.requestService = requestService;
    this.specificProcessingSRVLocatorService = specificProcessingSRVLocatorService;
    this.srvTimeoutConfiguration = srvTimeoutConfiguration;
}
```

```
[ERROR]         [error]: Build step io.quarkus.arc.deployment.ArcProcessor#validate threw an exception: javax.enterprise.inject.spi.DeploymentException: javax.enterprise.inject.AmbiguousResolutionE
ception: Ambiguous dependencies for type com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfiguration and qualifiers [@Default]
[ERROR]         - java member: com.criticalsoftware.dco.requestmanager.common.business.RequestManagerServiceImpl():srvTimeoutConfiguration
[ERROR]         - declared on CLASS bean [types=[com.criticalsoftware.dco.requestmanager.common.business.RequestManagerService, com.criticalsoftware.dco.requestmanager.common.business.RequestManage
ServiceImpl, java.lang.Object], qualifiers=[@Default, @Any], target=com.criticalsoftware.dco.requestmanager.common.business.RequestManagerServiceImpl]
[ERROR]         - available beans:
[ERROR]                 - CLASS bean [types=[com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfiguration, com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfi
urationSpecificCohortImpl, java.lang.Object], qualifiers=[@Default, @Any], target=com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfigurationSpecificCohortImpl]
[ERROR]                 - CLASS bean [types=[com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfiguration, com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfi
urationImpl, java.lang.Object], qualifiers=[@Default, @Any], target=com.criticalsoftware.dco.requestmanager.common.business.SrvTimeoutConfigurationImpl]
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

Constructor Injection - Findings

- For **constructor injection**, TestNg with @InjectMocks does not create a new instance of the test class; it keeps the state between test methods. When openMocks is called for the second time, Mockito will find the victim field already initialized and will try to use field injection. It's a known issue of InjectMock + TestNg;
- A solution is to put victim = null in a @AfterMethod.

```
@AfterMethod
public void close() throws Exception {
    autoCloseable.close();
    victim = null;
```

© Copyright Critical Software. All rights reserved.

Work Developed

SOAP UI

- Changes to the Health Check Tests:
  - Spaces.

© Copyright Critical Software. All rights reserved.

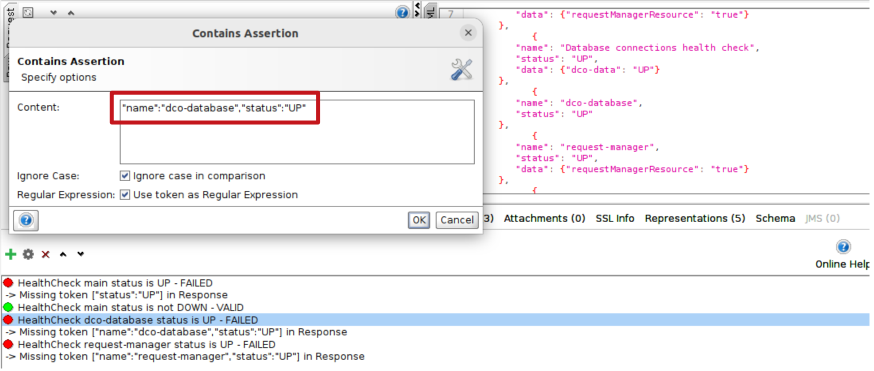
Information Classification: Confidential

Work Developed

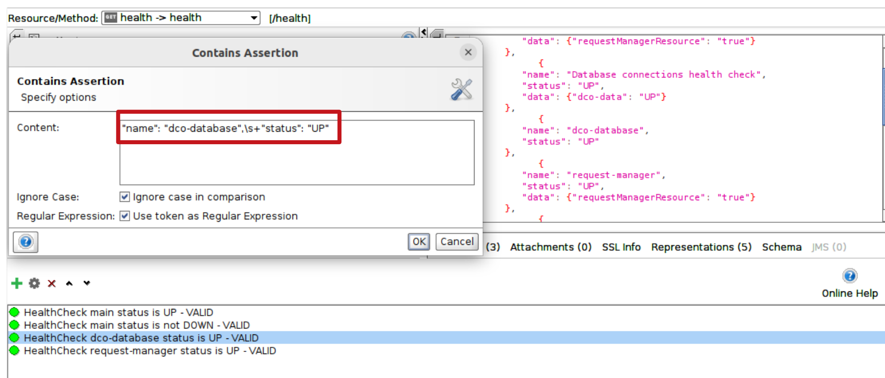
SOAP UI

- Changes to the Health Check Tests:
  - Spaces.

Failure:



Sucess:



© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

THANK YOU!

CRITICALSOFTWARE.COM

© Copyright Critical Software. All rights reserved.

Figure A.2: 2nd Show and Tell

Critical
software
3rd Show and Tell
CSWSYS component –
Quarkus
Bruno Gandres
Date: 07/07/2023
CRITICALSOFTWARE.COM
© Copyright Critical Software. All rights reserved.
Information Classification: Confidential
1

Agenda

- Overview
- System Tests

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Overview

- Execute system tests regarding Request Manager
- Defect fixes

- **Milestones:**
    - 28/03/2023 - Request Manager runs on Quarkus;
    - 30/03/2023 - Request Manager passes SOAP UI tests;
    - 13/04/2023 - Reflection removal;
    - 18/04/2023 - Replacing Field Injection with Constructor Injection;
    - 31/05/2023 – All system tests regarding Request Manager Passed;

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests

- Request Manager was commented in start-test-env.sh docker compose from testing environment

```
services_core=("portainer"
               "dco-clean-up-scheduler"
               "dco-configuration-manager"
               "dco-smki-web-server"
#               "dco-request-manager"
               )
```

```
131 #  dco-request-manager:
132 #    image: ${DOCKER_REPO}/dco/dco-request-manager:${DCO_BUILD_TAG}
133 #    container_name: dco-request-manager
134 #    deploy:
135 #      resources:
136 #        limits:
137 #          memory: 4G
138 #    restart: ${RESTART_STRATEGY}
139 #    environment:
140 #      EXTRA_JVM_PROPS: ${EXTRA_JVM_PROPS}
141 #    depends_on:
142 #      - dco-database-mysqld
143 #    volumes:
144 #      - "${DCO_CONFIG_VOL_DIR}/dco-request-manager/config:/opt/dco/config"
145 #      - "${DCO_CONFIG_VOL_DIR}/dco-request-manager/logs:/opt/dco/logs"
146 #    ports:
147 #      - "8070:8070"
148 #    networks:
149 #      dconet:
150 #        ipv4_address: ${DCO_REQMAN_IE_IP}
151 #    healthcheck:
152 #      test: ["CMD-SHELL", "if [[ $$(curl -s -f http://localhost:8070/health | awk -F\\\" '{print $$4}') = *UP* ]]; then exit 0; else exit 1; fi"]
153 #      interval: 1m
154 #      timeout: 5s
155 #      retries: 3
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests

- Dockerfile were added to Request Manager’s dist folder;
- Dependency *quarkus-container-image-docker* was added to perform Docker builds;
- The remaining component’s containers were run automatically, and the Request Manager container was run manually.

```
docker run --name CSWSYS-component --memory 4G --restart no --volume $(pwd)/deploy/local/ CSWSYS-component
/config:/deployments/config -v $(pwd)/deploy/local/ CSWSYS-component /logs/:/deployments/logs --env
EXTRA_JVM_PROPS=-DCSWSYS.smki-web-client.server.url=https://smki-web-server:7086/smki -p 8070:8070 --
network test_cswsysnet --ip 172.200.0.60 --health-cmd "if [[ $$(curl -s -f http://localhost:8070/health | awk -F\\\" '{print
$$4}') = *UP* ]]; then exit 0; else exit 1; fi" --health-interval 1m --health-retries 3 --health-timeout 5s brunogandres/cswsys-
component-shared:1.20.0-SNAPSHOT
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests - Problems

- Database connection refuse
  - Wrong database URL inside deploy config file – Was localhost as the development environment
  - Wrong database name - CSWSYSvv

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests - Problems

- Not finding the right log in order to understand that the container is running
    - The expected log message in order to know if the container was running with Thorntail was “THORN99999: Thorntail is Ready”
    - Was changed by “Profile prod activated”

```
#TODO: Replace lines when we have the request manager splitted again
#    Start Container  dco-request-manager-${s1sp}
#    Wait Until Container Log Has Message  dco-request-manager-${s1sp}  THORN99999: Thorntail is Ready  seconds_to_timeout=600
    Start Container  dco-request-manager
    Wait Until Container Log Has Message  dco-request-manager  Profile prod activated.  seconds_to_timeout=600
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests - Problems

- Problem with database connection checks.
  - The javax.ejb.scheduleExpression replacement had been forgotten, so the periodic task of checking the connection every 10 seconds never happened, keeping the database state UP when the test sent the database down.

```
==============================================================================
Health-Checks :: This test suite tests the output returned by the health ch...
==============================================================================
* Uploading /system-testing/scenarios/resources/ada-files/NO_LIMITS_ADA.csv ADA config to DCO
* Done uploading /system-testing/scenarios/resources/ada-files/NO_LIMITS_ADA.csv ADA config file
* No detect configs to upload for the current path
REQMNG-IE - Health Check - All UP :: The normal status of the Requ...
Script is waiting [local time 2023-05-11T13:17:00.93Z] + 10 seconds
| PASS |
------------------------------------------------------------------------------
REQMNG-IE - Health Check - Database DOWN :: This test validates th...
Script is waiting [local time 2023-05-11T13:17:16.54Z] + 10 seconds
| FAIL |
UP != DOWN
------------------------------------------------------------------------------
Health-Checks :: This test suite tests the output returned by the ... | FAIL |
2 critical tests, 1 passed, 1 failed
2 tests total, 1 passed, 1 failed
==============================================================================
```

```
//@PostConstruct
//private void initSchedule() {
    //final ScheduleExpression scheduleExpression = SchedulerUtils.createScheduleExpression(cronExpression);
    //timerService.createCalendarTimer(scheduleExpression);
//}
```

```
@Scheduled(cron = "0/10 * * ? * *",
        concurrentExecution = Scheduled.ConcurrentExecution.SKIP)
void checkDatabaseConnectionState() {
```

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests - Problems

- Wrong config values:
  - The same configuration files from development were used in the testing environment with the wrong configuration properties.
  - Inside the *dist* folder, there was an application.yaml with configuration properties fixed at build time and also configuration properties that are overridable at runtime, which led to an override of the timeout values of the tests.

dist/CSWSYS-component/src/main/resources

```yaml
quarkus:
  swagger-ui:
    always-include: true
  http:
    port: 8070
    non-application-root-path: ${quarkus.http.root-path}
  datasource:
    dco-data:
      db-kind: mysql
      username: dco
      password: dco
      jdbc:
        url: jdbc:mysql://localhost:3306/dccdco?useSSL=false&allowPublicKeyRetrieval=true&max_allowed_packet=25M
  hibernate-orm:
    dco-data:
      datasource: dco-data
      packages: com.criticalsoftware.dco.dcodata.entities
```

deploy/local/ CSWSYS-component /config

```yaml
quarkus:
  http:
    port: 8070
  log:
    level: INFO
    min-level: DEBUG
    category:
      "com.criticalsoftware.dco":
        level: DEBUG
        handlers: FILE,INTERNAL,HTTP_DUMPER
    console:
      format: "%K{level}%z{GMT}%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%h] [%c{1.}] [%t] [%X{moduleIdKey}] (%X{transactionId}) %s%e%n"
    file:
      enable: true
      async: true
      format: "%K{level}%z{GMT}%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%h] [%c{1.}] [%t] [%X{moduleIdKey}] (%X{transactionId}) %s%e%n"
      path: logs/dco-request.log
      file-suffix: .yyyy-MM-dd
    handler:
      file:
        "HTTP_DUMPER":
          enable: true
          format: "%K{level}%z{GMT}%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%h] [%c{1.}] [%t] [%X{moduleIdKey}] (%X{transactionId}) %s%e%n"
          path: logs/dco-request-http-dumper.log
        "INTERNAL":
          enable: true
          format: "%K{level}%z{GMT}%d{yyyy-MM-dd HH:mm:ss,SSS} %-5p [%h] [%c{1.}] [%t] [%X{moduleIdKey}] (%X{transactionId}) %s%e%n"
          path: logs/dco-request-internal.log
```

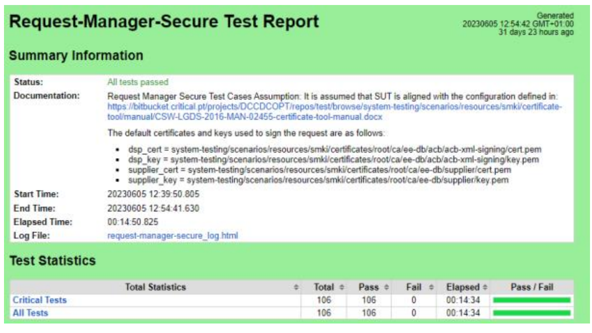
© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

Work Developed

System Tests

- Trilliant | Secure | Ie system tests were executed successfully



Request-Manager-Secure Test Report

Generated
20230605 12:54:42 GMT+01:00
31 days 23 hours ago

Summary Information

Status:	All tests passed
Documentation:	Request Manager Secure Test Cases Assumption: It is assumed that SUT is aligned with the configuration defined in: https://bitbucket.critical.pt/projects/DCCDCOPT/repos/test/browse/system-testing/scenarios/resources/smki/certificate-tool/manual/CSW-LGDS-2016-MAN-02455-certificate-tool-manual.docx The default certificates and keys used to sign the request are as follows: • dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/acb-xml-signing/cert.pem • dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/acb-xml-signing/key.pem • supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem • supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem
Start Time:	20230605 12:39:50.805
End Time:	20230605 12:54:41.630
Elapsed Time:	00:14:50.825
Log File:	request-manager-secure_log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	106	106	0	00:14:34	
All Tests	106	106	0	00:14:34	

Request-Manager-Ie Test Report

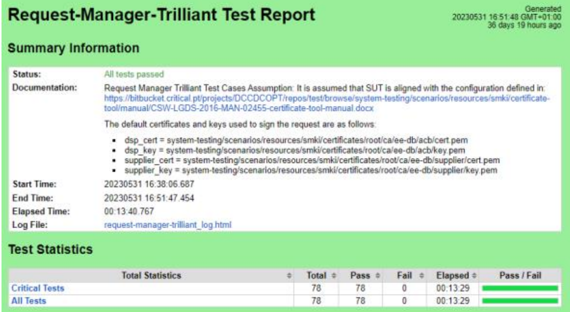
Generated
20230605 12:12:02 GMT+01:00
32 days 0 hours ago

Summary Information

Status:	All tests passed
Documentation:	Request Manager IE Test Cases Assumption: It is assumed that SUT is aligned with the configuration defined in: https://bitbucket.critical.pt/projects/DCCDCOPT/repos/test/browse/system-testing/scenarios/resources/smki/certificate-tool/manual/CSW-LGDS-2016-MAN-02455-certificate-tool-manual.docx The default certificates and keys used to sign the request are as follows: • dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/cert.pem • dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/key.pem • supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem • supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem
Start Time:	20230605 11:56:52.694
End Time:	20230605 12:12:01.518
Elapsed Time:	00:15:08.824
Log File:	request-manager-ie_log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	126	126	0	00:14:50	
All Tests	126	126	0	00:14:50	



Request-Manager-Trilliant Test Report

Generated
20230531 16:51:48 GMT+01:00
36 days 19 hours ago

Summary Information

Status:	All tests passed
Documentation:	Request Manager Trilliant Test Cases Assumption: It is assumed that SUT is aligned with the configuration defined in: https://bitbucket.critical.pt/projects/DCCDCOPT/repos/test/browse/system-testing/scenarios/resources/smki/certificate-tool/manual/CSW-LGDS-2016-MAN-02455-certificate-tool-manual.docx The default certificates and keys used to sign the request are as follows: • dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/cert.pem • dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/key.pem • supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem • supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem
Start Time:	20230531 16:38:06.687
End Time:	20230531 16:51:47.454
Elapsed Time:	00:13:40.767
Log File:	request-manager-trilliant_log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	78	78	0	00:13:29	
All Tests	78	78	0	00:13:29	

© Copyright Critical Software. All rights reserved.

Information Classification: Confidential

THANK YOU!

CRITICALSOFTWARE.COM

© Copyright Critical Software. All rights reserved.

Figure A.3: 3rd Show and Tell

Appendix B

Regression System Testing execution reports

In this section, the execution report of the Regression System Tests are available.

Generated
20230531 16:51:48 GMT+01:00

Summary Information

Status:	All tests passed
Documentation:	The default certificates and keys used to sign the request are as follows: ▪ dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/cert.pem ▪ dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/key.pem ▪ supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem ▪ supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem
Start Time:	20230531 16:38:06.687
End Time:	20230531 16:51:47.454
Elapsed Time:	00:13:40.767
Log File:	t_log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	78	78	0	00:13:29	
All Tests	78	78	0	00:13:29	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
14321	2	2	0	00:01:43	
14322	4	4	0	00:08:41	
14416	1	1	0	00:00:01	
14417	1	1	0	00:00:00	
14418	1	1	0	00:00:01	
14419	1	1	0	00:00:00	
14420	1	1	0	00:00:03	
14421	1	1	0	00:03:01	
14422	1	1	0	00:00:00	
14423	1	1	0	00:00:00	
14424	1	1	0	00:01:43	
14425	1	1	0	00:00:01	
14426	1	1	0	00:01:01	
14427	1	1	0	00:04:01	
14429	1	1	0	00:00:00	
14430	1	1	0	00:00:01	
14431	1	1	0	00:00:00	
14432	1	1	0	00:02:56	
14433	1	1	0	00:00:00	
14434	1	1	0	00:00:00	
14486	1	1	0	00:00:00	
14699	2	2	0	00:04:43	
15159	2	2	0	00:04:43	
15275	36	36	0	00:00:21	
15595	1	1	0	00:00:01	
15596	1	1	0	00:00:01	
15597	1	1	0	00:00:01	
15598	1	1	0	00:00:01	
15599	1	1	0	00:00:01	
15601	1	1	0	00:00:01	
15602	1	1	0	00:00:01	
15603	1	1	0	00:00:01	
15604	1	1	0	00:00:01	
15605	1	1	0	00:00:01	
15606	1	1	0	00:00:01	
15609	1	1	0	00:00:01	
15612	1	1	0	00:00:01	
15613	1	1	0	00:00:01	
15618	1	1	0	00:00:01	
15621	1	1	0	00:00:01	
15625	1	1	0	00:00:01	
15626	1	1	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
-15627	1	1	0	00:00:01	
-15628	1	1	0	00:00:01	
-15632	1	1	0	00:00:01	
-15633	1	1	0	00:00:01	
-15634	1	1	0	00:00:01	
-15637	1	1	0	00:00:01	
-15638	1	1	0	00:00:01	
-15639	1	1	0	00:00:01	
-15640	1	1	0	00:00:01	
-15641	1	1	0	00:00:01	
-15643	1	1	0	00:00:01	
-15645	1	1	0	00:00:01	
-15646	1	1	0	00:00:01	
-15648	1	1	0	00:00:01	
-15649	1	1	0	00:00:01	
-15652	1	1	0	00:00:01	
-15654	1	1	0	00:00:01	
-15656	1	1	0	00:00:01	
-157	1	1	0	00:02:56	
-15771	9	9	0	00:00:03	
-15805	1	1	0	00:00:00	
-15806	1	1	0	00:00:00	
-15807	1	1	0	00:00:00	
-15808	1	1	0	00:00:00	
-15809	1	1	0	00:00:00	
-15810	1	1	0	00:00:00	
-15811	1	1	0	00:00:00	
-15812	1	1	0	00:00:00	
-15813	1	1	0	00:00:00	
-15873	2	2	0	00:00:01	
-15880	1	1	0	00:00:01	
-15881	1	1	0	00:00:01	
-15970	1	1	0	00:00:00	
-16233	9	9	0	00:08:08	
-16357	1	1	0	00:00:00	
-16854	1	1	0	00:00:01	
-16855	1	1	0	00:00:04	
-16856	1	1	0	00:00:01	
-16857	1	1	0	00:00:01	
-16858	1	1	0	00:00:01	
-16981	2	2	0	00:00:01	
-17307	1	1	0	00:00:01	
-17308	1	1	0	00:00:00	
-175	2	2	0	00:04:02	
-18686	5	5	0	00:00:05	
-18713	1	1	0	00:00:01	
-18714	1	1	0	00:00:01	
-18715	1	1	0	00:00:01	
-18716	1	1	0	00:00:01	
-2152	1	1	0	00:00:01	
-41	5	5	0	00:00:04	
-46	4	4	0	00:00:01	
-6726	1	1	0	00:04:01	
-8920	1	1	0	00:01:01	
_NEGATIVE_TEST	41	41	0	00:00:22	
_POSITIVE_TEST	37	37	0	00:13:07	
_SRV11.3	1	1	0	00:04:01	
_TIMEOUT	5	5	0	00:12:42	
TEAM_VV	78	78	0	00:13:29	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
F	78	78	0	00:13:41	

Test Details

Totals | Tags | Suites | Search

Type:

Figure B.1: Test Report

Generated
20230605 12:54:42 GMT+01:00

Summary Information

Status: All tests passed

Documentation:

The default certificates and keys used to sign the request are as follows:

- dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/acb-xml-signing/cert.pem
- dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/acb-xml-signing/key.pem
- supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem
- supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem

Start Time: 20230605 12:39:50.805

End Time: 20230605 12:54:41.630

Elapsed Time: 00:14:50.825

Log File: _log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	106	106	0	00:14:34	
All Tests	106	106	0	00:14:34	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
14321	2	2	0	00:01:45	
14322	4	4	0	00:08:53	
14396	1	1	0	00:00:00	
14397	1	1	0	00:00:00	
14398	1	1	0	00:00:01	
14400	1	1	0	00:00:00	
14401	1	1	0	00:00:01	
14402	1	1	0	00:00:00	
14403	1	1	0	00:00:01	
14404	1	1	0	00:03:01	
14405	1	1	0	00:00:00	
14406	1	1	0	00:01:12	
14407	1	1	0	00:00:00	
14408	1	1	0	00:02:56	
14409	1	1	0	00:00:00	
14410	1	1	0	00:00:01	
14411	1	1	0	00:00:00	
14412	1	1	0	00:00:03	
14413	1	1	0	00:01:44	
14414	1	1	0	00:04:02	
14415	1	1	0	00:00:01	
14699	2	2	0	00:04:44	
15159	2	2	0	00:04:44	
15275	64	64	0	00:00:44	
15530	1	1	0	00:00:01	
15531	1	1	0	00:00:01	
15532	1	1	0	00:00:01	
15533	1	1	0	00:00:01	
15534	1	1	0	00:00:01	
15535	1	1	0	00:00:01	
15536	1	1	0	00:00:01	
15537	1	1	0	00:00:01	
15538	1	1	0	00:00:01	
15539	1	1	0	00:00:01	
15540	1	1	0	00:00:01	
15541	1	1	0	00:00:01	
15542	1	1	0	00:00:01	
15543	1	1	0	00:00:01	
15544	1	1	0	00:00:01	
15545	1	1	0	00:00:01	
15546	1	1	0	00:00:01	
15547	1	1	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
-15548	1	1	0	00:00:01	
-15549	1	1	0	00:00:01	
-15550	1	1	0	00:00:01	
-15551	1	1	0	00:00:01	
-15552	1	1	0	00:00:01	
-15553	1	1	0	00:00:01	
-15554	1	1	0	00:00:01	
-15555	1	1	0	00:00:01	
-15556	1	1	0	00:00:01	
-15557	1	1	0	00:00:01	
-15558	1	1	0	00:00:01	
-15559	1	1	0	00:00:01	
-15560	1	1	0	00:00:01	
-15561	1	1	0	00:00:01	
-15562	1	1	0	00:00:01	
-15563	1	1	0	00:00:01	
-15564	1	1	0	00:00:01	
-15565	1	1	0	00:00:01	
-15566	1	1	0	00:00:01	
-15567	1	1	0	00:00:01	
-15568	1	1	0	00:00:01	
-15569	1	1	0	00:00:01	
-15570	1	1	0	00:00:01	
-15571	1	1	0	00:00:01	
-15572	1	1	0	00:00:01	
-15573	1	1	0	00:00:01	
-15574	1	1	0	00:00:01	
-15575	1	1	0	00:00:01	
-15576	1	1	0	00:00:01	
-15577	1	1	0	00:00:01	
-15578	1	1	0	00:00:01	
-15579	1	1	0	00:00:01	
-15580	1	1	0	00:00:01	
-15581	1	1	0	00:00:01	
-15582	1	1	0	00:00:01	
-15583	1	1	0	00:00:01	
-15584	1	1	0	00:00:01	
-15585	1	1	0	00:00:01	
-15586	1	1	0	00:00:01	
-15587	1	1	0	00:00:01	
-15588	1	1	0	00:00:01	
-15589	1	1	0	00:00:01	
-15590	1	1	0	00:00:01	
-15591	1	1	0	00:00:01	
-15592	1	1	0	00:00:01	
-15593	1	1	0	00:00:01	
-157	1	1	0	00:02:56	
-15771	9	9	0	00:00:03	
-15823	1	1	0	00:00:00	
-15824	1	1	0	00:00:00	
-15825	1	1	0	00:00:00	
-15826	1	1	0	00:00:00	
-15827	1	1	0	00:00:00	
-15828	1	1	0	00:00:00	
-15829	1	1	0	00:00:00	
-15830	1	1	0	00:00:00	
-15831	1	1	0	00:00:00	
-15873	2	2	0	00:00:02	
-15878	1	1	0	00:00:01	
-15879	1	1	0	00:00:01	
-15970	1	1	0	00:00:00	
-16233	9	9	0	00:08:21	
-16356	1	1	0	00:00:00	
-16849	1	1	0	00:00:01	
-16850	1	1	0	00:00:01	
-16851	1	1	0	00:00:05	
-16852	1	1	0	00:00:02	
-16853	1	1	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
DCO-16981	2	2	0	00:00:27	
17299	1	1	0	00:00:01	
17301	1	1	0	00:00:26	
175	2	2	0	00:04:13	
18686	5	5	0	00:00:07	
18709	1	1	0	00:00:01	
18710	1	1	0	00:00:01	
18711	1	1	0	00:00:01	
18712	1	1	0	00:00:01	
2152	1	1	0	00:00:01	
41	5	5	0	00:00:05	
46	4	4	0	00:00:02	
6726	1	1	0	00:04:02	
8920	1	1	0	00:01:12	
_NEGATIVE_TEST	55	55	0	00:00:36	
_POSITIVE_TEST	51	51	0	00:13:57	
_SRV11.3	1	1	0	00:04:02	
_TIMEOUT	5	5	0	00:12:54	
TEAM_VV	106	106	0	00:14:34	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
	106	106	0	00:14:51	

Test Details

Totals | Tags | Suites | Search

Type:

Figure B.2: Test Report

Generated
20230605 12:12:02 GMT+01:00

Summary Information

Status:	All tests passed
Documentation:	The default certificates and keys used to sign the request are as follows: - dsp_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/cert.pem - dsp_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/acb/key.pem - supplier_cert = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/cert.pem - supplier_key = system-testing/scenarios/resources/smki/certificates/root/ca/ee-db/supplier/key.pem
Start Time:	20230605 11:56:52.694
End Time:	20230605 12:12:01.518
Elapsed Time:	00:15:08.824
Log File:	e_log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	126	126	0	00:14:50	
All Tests	126	126	0	00:14:50	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
10117	1	1	0	00:00:01	
12256	1	1	0	00:00:05	
12726	1	1	0	00:00:01	
13023	1	1	0	00:00:01	
13068	1	1	0	00:00:01	
14321	2	2	0	00:01:45	
14322	4	4	0	00:08:48	
14699	2	2	0	00:02:51	
15159	2	2	0	00:02:51	
15173	1	1	0	00:00:20	
15275	64	64	0	00:00:47	
15466	1	1	0	00:00:01	
15467	1	1	0	00:00:01	
15468	1	1	0	00:00:01	
15469	1	1	0	00:00:01	
15470	1	1	0	00:00:01	
15471	1	1	0	00:00:01	
15472	1	1	0	00:00:01	
15473	1	1	0	00:00:01	
15474	1	1	0	00:00:01	
15475	1	1	0	00:00:01	
15476	1	1	0	00:00:01	
15477	1	1	0	00:00:01	
15478	1	1	0	00:00:01	
15479	1	1	0	00:00:01	
15480	1	1	0	00:00:01	
15481	1	1	0	00:00:01	
15482	1	1	0	00:00:01	
15483	1	1	0	00:00:01	
15484	1	1	0	00:00:01	
15485	1	1	0	00:00:01	
15486	1	1	0	00:00:01	
15487	1	1	0	00:00:01	
15488	1	1	0	00:00:01	
15489	1	1	0	00:00:01	
15490	1	1	0	00:00:01	
15491	1	1	0	00:00:01	
15492	1	1	0	00:00:01	
15493	1	1	0	00:00:01	
15494	1	1	0	00:00:01	
15495	1	1	0	00:00:01	
15496	1	1	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
-15497	1	1	0	00:00:01	
-15498	1	1	0	00:00:01	
-15499	1	1	0	00:00:01	
-15500	1	1	0	00:00:01	
-15501	1	1	0	00:00:01	
-15502	1	1	0	00:00:01	
-15503	1	1	0	00:00:01	
-15504	1	1	0	00:00:01	
-15505	1	1	0	00:00:01	
-15506	1	1	0	00:00:01	
-15507	1	1	0	00:00:01	
-15508	1	1	0	00:00:01	
-15509	1	1	0	00:00:01	
-15510	1	1	0	00:00:01	
-15511	1	1	0	00:00:01	
-15512	1	1	0	00:00:01	
-15513	1	1	0	00:00:01	
-15514	1	1	0	00:00:01	
-15515	1	1	0	00:00:01	
-15516	1	1	0	00:00:01	
-15517	1	1	0	00:00:01	
-15518	1	1	0	00:00:01	
-15519	1	1	0	00:00:01	
-15520	1	1	0	00:00:01	
-15521	1	1	0	00:00:01	
-15522	1	1	0	00:00:01	
-15523	1	1	0	00:00:01	
-15524	1	1	0	00:00:01	
-15525	1	1	0	00:00:01	
-15526	1	1	0	00:00:01	
-15527	1	1	0	00:00:01	
-15528	1	1	0	00:00:01	
-15529	1	1	0	00:00:01	
-157	1	1	0	00:02:57	
-15771	9	9	0	00:00:04	
-15814	1	1	0	00:00:00	
-15815	1	1	0	00:00:01	
-15816	1	1	0	00:00:00	
-15817	1	1	0	00:00:00	
-15818	1	1	0	00:00:00	
-15819	1	1	0	00:00:00	
-15820	1	1	0	00:00:00	
-15821	1	1	0	00:00:00	
-15822	1	1	0	00:00:00	
-15873	2	2	0	00:00:02	
-15876	1	1	0	00:00:01	
-15877	1	1	0	00:00:01	
-15970	1	1	0	00:00:00	
-15982	1	1	0	00:00:20	
-16233	9	9	0	00:08:16	
-16355	1	1	0	00:00:00	
-16395	3	3	0	00:01:09	
-16844	1	1	0	00:00:05	
-16845	1	1	0	00:00:01	
-16846	1	1	0	00:00:01	
-16847	1	1	0	00:00:02	
-16848	1	1	0	00:00:01	
-16981	2	2	0	00:00:01	
-17311	1	1	0	00:00:01	
-17312	1	1	0	00:00:00	
-175	2	2	0	00:04:08	
-186	11	11	0	00:00:10	
-18686	8	8	0	00:00:09	
-18707	1	1	0	00:00:01	
-18708	1	1	0	00:00:01	
-2152	1	1	0	00:00:01	
-2247	1	1	0	00:00:01	
-2248	1	1	0	00:00:01	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
DCO-2250	1	1	0	00:03:00	
2251	1	1	0	00:00:01	
2252	1	1	0	00:00:00	
2253	1	1	0	00:00:01	
2254	1	1	0	00:00:00	
2255	1	1	0	00:00:00	
2256	1	1	0	00:00:01	
2257	1	1	0	00:00:03	
2258	1	1	0	00:00:01	
2259	1	1	0	00:01:44	
2261	1	1	0	00:00:00	
2262	1	1	0	00:02:57	
2263	1	1	0	00:00:00	
2264	1	1	0	00:00:01	
2265	1	1	0	00:00:01	
2266	1	1	0	00:00:01	
2267	1	1	0	00:00:00	
2268	1	1	0	00:00:01	
2269	1	1	0	00:00:01	
2270	1	1	0	00:00:01	
2271	1	1	0	00:00:01	
2272	1	1	0	00:00:00	
3999	6	6	0	00:00:06	
4072	1	1	0	00:00:01	
4073	1	1	0	00:00:01	
4074	1	1	0	00:00:01	
4075	1	1	0	00:00:01	
4076	1	1	0	00:00:01	
4077	1	1	0	00:00:01	
41	5	5	0	00:00:05	
46	4	4	0	00:00:02	
546	6	6	0	00:00:06	
6168	1	1	0	00:03:01	
6726	2	2	0	00:04:02	
6946	2	2	0	00:00:02	
7311	1	1	0	00:00:00	
7312	1	1	0	00:04:02	
7357	1	1	0	00:00:01	
7548	1	1	0	00:00:01	
8920	1	1	0	00:01:07	
9508	1	1	0	00:01:07	
_NEGATIVE_TEST	72	72	0	00:00:58	
_POSITIVE_TEST	54	54	0	00:13:53	
_SRV11.3	1	1	0	00:04:02	
_TIMEOUT	6	6	0	00:12:50	
TEAM_VV	126	126	0	00:14:50	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
	126	126	0	00:15:09	

Test Details

Totals | Tags | Suites | Search

Type:

Figure B.3: Test Report