INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Implementation of ISO 13485:2016 for medical device quality systems

JOÃO MIGUEL CORREIA DA MOTA
Outubro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Implementation of ISO 13485:2016 for medical device quality systems

João Miguel Correia da Mota

Dissertation to obtain the master's degree in Informatics Engineering, Area of Expertise in Graphics and Multimedia Systems

Advisor: António Silva Pereira

Supervisor: Cláudia Babo

Porto, October 2022

Dedicatory

First and foremost, I dedicate this dissertation to my parents and sister.

Secondly, I would like to thank my supervisor Cláudia Babo for incentivizing me to conclude my master's degree and helping me along this work.

Also, I would like to thank my co-workers Marilisa Silva and Dionisia Pereira for helping me review this dissertation with their expertise.

Finally, would like to thank professor António Silva Pereira for his availability and guidance throughout this dissertation.

Resumo

A Norma 13485:2016 especifica os requisitos para um sistema de gestão da qualidade que possa ser usado por uma organização envolvida em uma ou mais etapas de ciclo de vida de um dispositivo médico, incluindo conceção e desenvolvimento, produção, armazenamento e distribuição, instalação, assistência, desativação final e eliminação de dispositivos médicos. Também é usada como referência para conceção e desenvolvimento ou provisão de atividades associadas, como por exemplo, apoio técnico.

Esta norma, baseia-se na Norma 9001:2008 cujo objetivo é documentar processos, procedimentos e responsabilidades envolvidos no cumprimento de políticas e objetivos de qualidade.

O objetivo deste trabalho é realizar uma análise do processo de implementação da Norma 13485:2016 na empresa Nimco Portugal Lda, que produz calçado ortopédico por medida, e pretende certificar os seus produtos como dispositivos médicos.

Esta dissertação representa uma visão geral da Norma e a sua aplicabilidade na área de IT, levantamento de requisitos na estrutura atual da organização, análise de risco à infraestrutura e sistemas, implementação de procedimentos e documentos necessários para atender os requisitos. Além disso, também é desenvolvida uma análise de valor para mencionar os benefícios da aplicação da Norma à organização e à definição da sua proposta de valor.

Durante a realização desta dissertação e após auditoria por parte da Associação Portuguesa de Certificação (APCER), foram enumeradas algumas não conformidades que são analisadas e atribuídas ações para resolução ou mitigação do risco, como por exemplo, relacionada com validação de software. Para solução deste problema, são estudadas diversas metodologias de testes *agile* com o intuito da empresa utilizar no desenvolvimento de software.

A solução para a validação de software é desenhada, descrita e implementada nesta dissertação, utilizando uma das metodologias de testes analisadas.

Por fim, é efetuado um levantamento ao estado da sua implementação e efetuado um inquérito interno para avaliação da solução.

Palavras-chave: dispositivos médicos, sistemas de gestão de qualidade, ISO 13485:2016

Abstract

The International standard 13485:2016 specifies the requirements for a quality management system that can be used by an organization involved in the production of one or more stages of the life cycle of a medical device. These stages can include from design and development to production, storage and distribution, installation, servicing and final decommissioning and disposal of medical devices. It can also be applied to design, development or provision of associated activities, such as technical support.

This standard is based on the ISO 9001:2008, whose objective is to document processes, procedures and responsibilities involved in meeting with quality policies and objectives.

The objective of this work is to carry out an analysis of the implementation process of the ISO 13485:2016 in the company Nimco Portugal Lda, which produces made-to-measure orthopedic footwear, and intends to certify its products as medical devices.

This dissertation represents an overview of the Standard and its applicability in the IT area, requirements survey in the current structure of the organization, risk analysis to the infrastructure and systems, implementation of procedures and documents needed to meet the requirements. In addition, a value analysis is also developed to mention the benefits of applying the Standard to the organization and the definition of its value proposition.

During the realization of this dissertation and after the Associação Portuguesa de Certificação (APCER) audit, some non-conformities were listed that are analyzed and assigned actions for resolution or risk mitigation, such as related to software validation. To solve this problem, several agile testing methodologies are studied with the intent of the company to use them in software development.

The solution for software validation is designed, described and implemented in this dissertation, using one of the analyzed testing methodologies.

Finally, a status of its implementation is assessed, and an internal survey is conducted to evaluate the solution.

Keywords: medical devices, quality management systems, ISO 13485:2016

Table of contents

List of Figures

List of Tables

Acronyms

APCER	Associação Portuguesa de Certificação
MD	Medical Device
MCO	Modular Concept Orthopedics
MDR	Medical Devices Regulation
QMS	Quality Management System
XP	Extreme Programming
TDD	Test driven development
BDD	Behavior driven development
ATDD	Acceptance Test driven development
NCD	New Concept Development
AHP	Analytic Hierarchy Process
JS	JavaScript
CSS	Cascading Style Sheets
RDBMS	Relational Database Management System
IDE	Integrated Development Environment
VCS	Version Control System
FTP	File Transfer Protocol
CI	Continuous Improvement
CD	Continuous Delivery
QA	Quality Assurance
SRP	Single Responsibility Principle
OCP	Open/Closed Principle
LSP	Liskov Substitution Principle
ISP	Interface Segregation Principle
DIP	Dependency Injection Principle

UI	User Interface
C.R.A.P.	Change Risk Anti-Patterns

1 Introduction

1.1 Context

In the highly regulated world of medical device manufacturing, organizations must show that each product meets both customer expectations and applicable regulations (ISO 13485 Certification Services, Training Courses & Resources, 2021).

The global medical devices market size was 432.23 billion US Dollars (380.84 billion Euros) in 2020. The global impact of COVID-19 has been unprecedented and staggering, with medical devices witnessing a negative impact on the adoption rate across all regions amid the pandemic (Market Research Report, 2021).

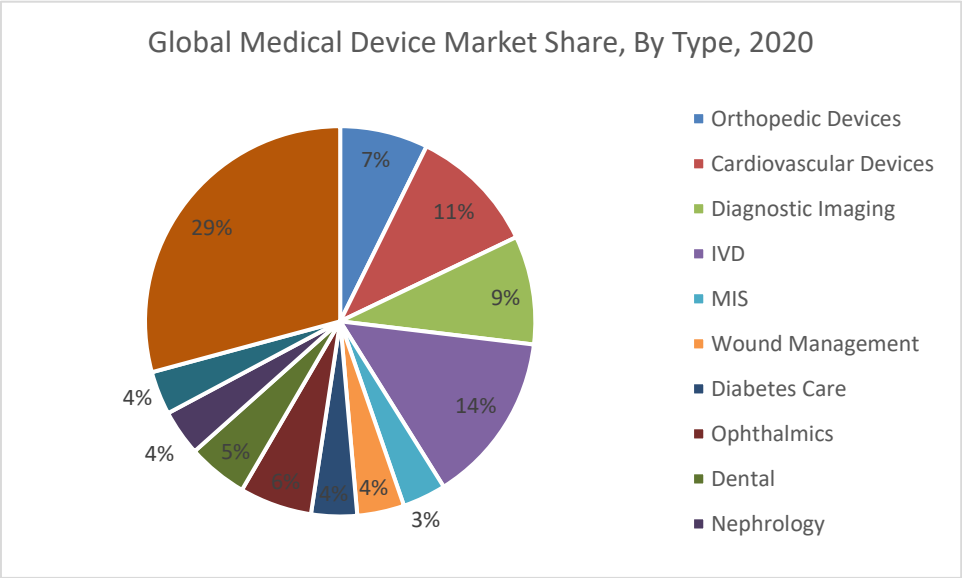

Figure 1 – Global Medical Device Market Share. Source: (Market Research Report, 2021).

This growth is driven by a prevalence of chronic diseases, including diabetes, cancer, and other infectious diseases, owing to the adoption of sedentary lifestyles and other factors. Also, a quick rise in the geriatric population is supplementing the growth of ophthalmic and orthopedic devices due to the increasing incidence of impaired vision and hip fractures in the elderly population (Market Research Report, 2021).

On the 5th of April of 2017, the regulation of the European Union 2017/745 for clinical investigation and sale of medical devices for human use was published (EU 2017/745, 2017). This regulation became mandatory for organizations on the 26th of May of 2021.

It becomes essential that companies implement their Quality Management System (QMS) following the requirements of applicable law. Therefore ISO 13485:2016 came to fruition to provide the companies with a standard set of regulations to ensure security to its users and ensure that devices are safe and risk-free.

ISO 13485:2016 introduces subtle but detailed changes requiring complete workplace documentation, risk management, design control, and regulatory requirements. The revised standard includes the need for a risk-based approach to the Quality Management System, the ISO 14971:2019, a greater focus on regulatory requirements and full management responsibilities, greater control over suppliers and subcontracted activities and an emphasis on risk management throughout the product lifecycle (RAPS, 2021).

ISO 13485:2016 applies to medical device manufacturers and organizations that support medical device manufacturers. It maintains manufacturers' duty to ensure that the devices consistently comply with the customer's requirements and applicable regulations.

This document, within the scope of the thesis curricular unit of the master's degree in Informatics Engineering, specialization in graphics and multimedia systems, was developed on the company Nimco Portugal Lda.

1.2 Problem

Nimco is a specialized footwear manufacturer dedicated to customized footwear production for adults and children, and it has three production lines.

One of the company's goals is to produce medical devices and ensure the safeguard and well-being of its customers. The company must ensure that the productions systems are robust and secure, from ordering to the final product. Another goal is to increase the company’s market share in Europe with the Modular Concept Orthopedics (MCO) and Stock Collection line.

One of the requirements mentioned above is having a certified quality management system based on the ISO 13485:2016. An organization involved in the several stages of the life cycle of a medical device shall use this international standard to verify if its quality management system is compliant. These stages can include design and development, production, storage and

distribution, installation, servicing and final decommissioning and disposal of medical devices, design and development, or associated activities (e.g., technical support).

After Associação Portuguesa de Certificação (APCER) audit some non-conformities were listed regarding IT, especially regarding software validation.

This project will focus on implementing the ISO 13485:2016 on the company by solving or defining solutions to mitigate the non-conformities detected.

1.3 Objectives

The main objective is to evaluate and implement solutions to meet the requirements of the ISO 13485:2016 Medical devices - Quality management systems - Requirements for regulatory purposes.

Nimco, being a company that produces medical devices, wants to comply with the EU regulation that became mandatory in the European Union. As such, it was decided by the company the need for an update to the QMS. This section describes the actions taken to achieve all the objectives defined for this dissertation.

Nimco has already developed and implemented a QMS with all the policies and processes of the company. Due to the continuous improvement of its products and pursuit to better satisfy its customer's needs, in 2004, the company was certified in the ISO 9001 (Quality management systems).

The EU Medical Device Regulation (MDR) is a regulation released by the European Union that normalizes is needed to be done by a company that wants to manufacture or import medical devices. This regulation includes information on how medical devices must be marked and certified. To sell their medical devices in the European market and get a CE marking, the company must meet the requirements mentioned in the regulation (EU) 2017/745 (EU 2017/745, 2017).

The stock collection products have been compliant and certified since 2013. Nimco signed the declaration of conformity on the 19th of May of 2021 at the Chamber of Commerce Nijmegen. In 2022, the company started the certification process and compliance on MCO products with the MDR regulation EU 2017/745. According to this regulation, Nimco products are considered a medical device of class I – low risk.

The following steps shall pursuit this objective:

- Study the ISO and compile a report of the findings for the whole organization focused on the IT requirements.

- Create a risk-based approach for the software and physical architecture of the company, identifying non-compliance with the norm.

- Develop a solution and support with documentation to solve the non-conformities detected by the external audit regarding software validation, based on an agile test-driven development methodology.

- Evaluate if the solution meets the requirements for ISO compliance.

1.4 Research Methodology

This dissertation follows the Design Science Research Methodology (DSRM) to improve the quality of the solution to the problems mentioned in section 1.2.

According to (Peffers et al., 2017), the DSRM is a "commonly accepted framework for design science research and a template for its presentation."

The DSRM, according to (Peffers et al., 2016), is a process model consisting of six activities:

- Problem identification and motivation – consists of defining the problem that is the objective of the research and justifying the solution's value. Section 1.2 describes the problem, and in chapters 2 and 3 are proposed the solution's value.

- Objectives of a solution – consists of defining the objectives based on the problem. These objectives can be quantitative or qualitative, so their outcomes will differ from one another. Section 1.3 is where the objectives are defined.

- Design and development – this activity is expected to create and implement a solution. This design and development include determining desired functionality and architecture and finally developing a solution. Chapter 4 describes the design, and chapter 5 describes the development.

- Demonstration – consists of describing and demonstrating the efficacy of the solution presented. This activity involves experimentation and evidence in how the solution solves the problem. Chapter 6 describes the demonstration.

- Evaluation – consists of observing and measuring how the implementation supports a solution to the problems. It also involves comparing the results in the demonstration with the objectives defined initially. Chapter 6 describes the evaluation.

- Communication – finally, this activity consists of how effective the solution results compared to the objectives. Chapter 7 describes this activity.

1.5 Approach strategy

With the analysis of the ISO 13485:2016 and after the APCER Audit, some non-conformities were detected, and need to be addressed by the company.

The first phase of the project consists of:

- List and analyze the ISO requirements and focusing on the requirements regarding IT.
- For each requirement, create a report of the status of compliance.
- Define the metrics for a risk assessment and apply it in the company regarding the IT requirements.
- Analyze and define solutions to solve the non-conformities detected in the external audit.
- Compare several existing agile methodologies and define an approach to comply with the requirements of the ISO.
- Do a value analysis and value proposition for the company regarding this certification.

The second phase of the project consists of:

- Develop a solution for the software validation requirement that is not compliant with the ISO.
- Implement one of the agile methodologies studied in the state of the art.
- Check if the solutions presented fulfill all objectives defined.

2 State of the Art

2.1 Nimco Portugal, Lda

2.1.1 Company Presentation

Nimco was founded in 1904 in the city of Nijmegen, in the Netherlands, by Johan Veerschuur. In 1968, Adrian Krol bought the company. From this event on, Nimco began to emerge and stand out as one of the world's leading manufacturers of orthopedic footwear.

In 1971, the company established a shoe factory in the city of Nijmegen. Later, in 1999, it established the Portuguese factory, the Nimco Portugal, Lda, for fully custom-made shoes.

In 2006, Nimco created the MCO line to embrace new orthopedic solutions without the need of making individuals lasts. In 2016, the company moved to the current facilities in Portugal in Cesar, Oliveira de Azeméis, to an area of 3600m2, due to the company's growth.

Nimco is a specialized footwear manufacturer dedicated to produce customized footwear for adults and children with three production lines:

1. Orthopedic line, which produces customized orthopedic footwear.
2. MCO line, whose products are built with pre-made master lasts and are changed by adding modules, giving the personalization needed in less severe cases.
3. Stock Collection line, an off-the-shelve collection designed for diabetic patients.

The company manufactures shoes and other orthopedic solutions so that foot health, comfort, and a perfect fit are a mean to improve mobility and, consequently, customers quality of life. The orthopedic footwear adapts to the customer's physical or design needs.

2.1.2 Organizational structure

Nimco has an organizational structure defined into three main structures: administration, operations, and commercial. Figure 2 represents a simplified organization chart.

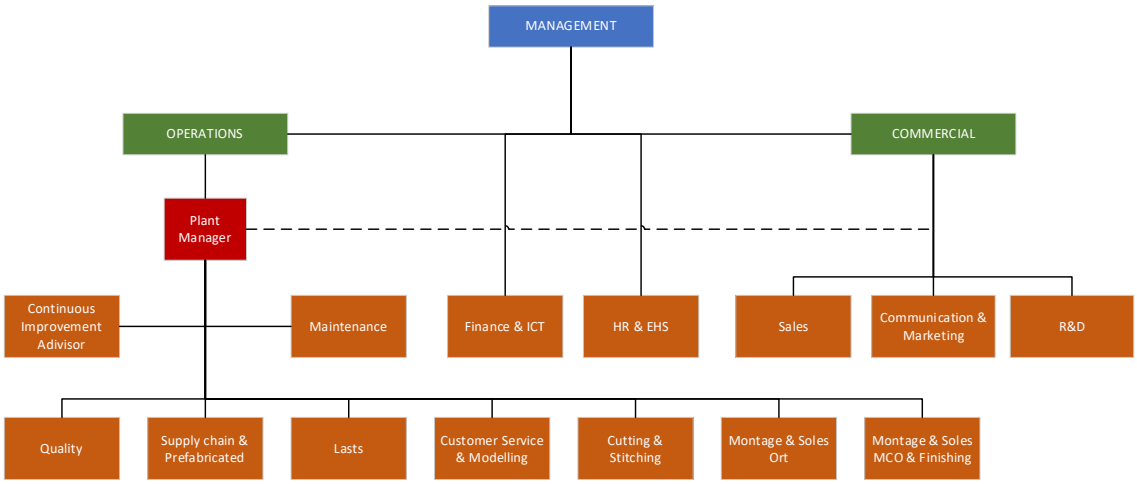

Figure 2 – Organizational structure of Nimco Portugal, Lda.

The IT department, where I currently work in, is responsible for:

- Managing the IT infrastructure - software and hardware.
- Maintain and develop the internal production system.
- Provide help to support other departments regarding IT.
- Manage projects with external partners.
- Acquire new hardware and software.
- Evaluate and ensure the security of the company systems and networks.

The quality department is responsible for:

- Quality Planning.
- Quality Improvement.
- Quality Control.
- Quality Assurance.
- Inspections and oversee the process.
- Quality Audits.
- Metrology measurement.
- Compliance with legal requirements in the quality area.
- Integrated management system management.
- Ensure the updating of client specifications, procedures, and work instructions.

2.2 Products

This subsection describes Nimco's products and collections.

2.2.1 Orthopedic

2.2.1.1 Fully orthopedic

The basic guidelines for NM4Y orthopedic shoes are the following:

- Shoes made with natural and high-quality materials. Use a wide range of materials and choose from several options. Use custom-made shoe last, supplement, and shoe style.
- Use patented stretch leathers for perfect anatomical adaptation of the shoe and maximum comfort.
- Contains minimum seams to avoid friction or points of pressure.
- Great freedom of choice.
- All alterations are possible by the customers, in shape or function, as they are technically possible.

2.2.1.2 Infinity Collection

Infinity is Nimco's latest collection, focused on functionality. This collection contains everlasting classic models, house shoes, and waterline in classic color combinations. That is one of the reasons why Infinity models cannot be changed in their appearance. However, make changes due the function can ensure that the patient has all the needs to walk comfortably.

The customers can change in the model the features that can interfere or benefit the performance of the shoe, as it is an aid for walking:

- Change to the diabetic lining.
- Turn lace into Velcro.
- Ask for special stiffeners in the medium and high boots.
- Rockers.
- Flares.
- Carbon sole reinforcements.
- External raises.

This collection uses Nimco's own MCO system of lasts and fittings or the customers' own produced lasts to produce custom-made orthopedic footwear.

2.2.2 MCO

MCO is a modular concept orthopedic shoe line in which customers can customize features according to some parameters (pre-engineered modules):

- Start by using Nimco's special orthopedic master lasts, a range of hundreds of last options in type, nose type and width.
- Customers can add 14 different standard last adaptations to a master last so it will fit customers' feet to perfection. A representation of these adaptations is in figure 3, and the specification and values are in table 1.
- Further adaptations like extra spaces, rockers, flares, special stiffeners, and others are also available.
- In addition, customers can change the style, colors, leathers, and soles.

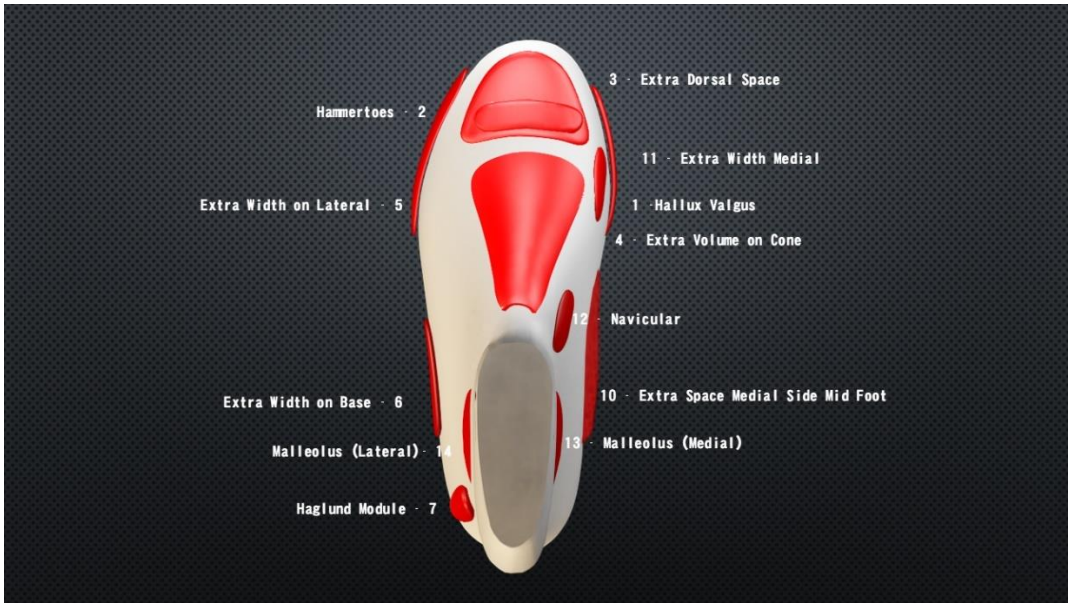

Figure 3 – MCO lasts adaptations.

Table 1 – MCO last adaptations modules and values.

		Values	
Nº	**Module**	**Sizes 16 - 34**	**Sizes 35 - 46**
1	Hallux Valgus	5 or 8 mm	
2	Hammertoes	5 or 8 mm	
3	Extra toe box space dorsal	Up to 5 mm	Up to 10 mm
4	Extra width on cone	Up to 10 mm	Up to 20 mm
5	Extra width lateral	Up to 5 mm	Up to 10 mm
6	Extra width base V	Up to 5 mm	Up to 10 mm
7	Haglund	3 or 7 mm	3, 7 or 10 mm
8	Extra space for foot plantar	3, 5 or 8 mm	
9	Overall extra space	3, 5 or 8 mm	
10	Extra space medial side mid foot	Up to 10 mm	Up to 20 mm
11	Extra width medial	Up to 5 mm	Up to 10 mm
12	Extra space on the navicular area	Up to 15 mm	
13	Extra space on the internal malleolus	Up to 20 mm	
14	Extra space on the external malleolus	Up to 20 mm	

2.2.3 Stock Collection

Untreated diabetes makes feet more vulnerable. This vulnerability is due to impaired metabolism, making the skin on the feet more prone to minor cracks and lesions. Also, it is due to nerve irritation and damage in the long term, which reduces the sensitivity of the nerves. Injuries are less easy to perceive, if at all. Regular, careful foot checks are therefore essential.

Five hundred thirty-seven million adults (20-79 years) live with diabetes, 1 in 10. Diabetes Atlas predicts this number to rise to 643 million by 2030 and 783 million by 2045 (Diabetes Atlas, 2022).

Protective footwear can help prevent pressure sores in the first place, counteracting inflammatory complications and even amputations at an early stage.

Stock Shoes are an "off the shelve" stock collection made for immediate purchase and delivery. These products use materials specially made and thought for diabetic patients. These shoes are developed with unique orthopedic shoes lasts. They have a stitchless construction and extra internal space, making them fit for everyday use for people who love comfort, especially diabetics.

Some of the characteristics of this collection are:

- All models produced using approved ladies' lasts N180 K / N180 M / N180 O, and men's lasts N39 K / M / O.
- These lasts have a nature-shaped front part providing sufficient space for the toes. A narrow heel part provides a perfect fitting that avoids heel slippage.
- It is produced with "Xsensible Inside Stretch Leather," providing greater flexibility to accommodate changes in foot volume, e.g., edema or prominent bunions.
- The linings used in the products are for diabetic patients, made of synthetic material, soft, comfortable, durable, and anti-bacterial.
- Shoes can accommodate two inlays: a 5mm cork inlay and a 5mm cork-leather inlay. Both are removable, and the maximum depth for individual inlays is 10mm.
- Additionally, reinforced inlays can be supplied with the shoes as well.
- All shoes are available in UK ladies' sizes 3 to 9 and men sizes 6 to 12. (Including ½ sizes)
- Outsoles can be sportive Polyurethane outsoles and lightweight Ethylene-vinyl acetate.

2.3 ISO

The International Organization for Standardization (ISO) is an international standards body composed of representatives from various national standards organizations (ISO Members, 2021).

The organization's primary goal is to provide common standards between countries, and more than 20,000 standards have been published (ISO Wiki, 2021).

The use of the standards aids in creating products and services that are safe, reliable, and of good quality. These standards help companies increase productivity while minimizing errors and waste. (ISO Members, 2021).

In Portugal, the national standard organization is Instituto Português da Qualidade (IPQ) (IPQ, 2021).

The following sections will describe the ISO standards used in this project and the focus on the items related to IT.

2.4 ISO 13485:2016

ISO 13485:2016 is designed for organizations to use throughout the life cycle of a medical device, from initial conception to production and post-production, including final decommission and disposal. It also covers storage, distribution, installation, servicing, and the provision of associated services (ISO 13485, 2016).

This ISO helps the organizations design a quality management system that establishes and maintains the effectiveness of its processes. It reflects a solid commitment to continual improvement. It gives customers confidence in its ability to bring safe and effective products to market (ISO 13485, 2016).

While ISO 13485 is a stand-alone standard, it is similar in scope and intent to ISO 9001, Quality management systems. It contains additional requirements specific to organizations involved in the life cycle of medical devices. ISO 13485 removes elements of ISO 9001 because they were not relevant as regulatory requirements (ISO 13485, 2016).

We can assume that while MDR focuses on the product, the ISO 13486:2016 focuses on the product and services.

ISO published a survey where a total of certificates issued by the end of 2020 for the ISO 13486 was 25656, being 11798 on the EU (46%) while in Portugal only 62 (0,24%). In Table 2, we can see all the data from 2004 until 2020 (ISO Survey, 2021).

Table 2 – Surveys from ISO regarding the total of certifications from ISO 13485:2003 and ISO 13485:2016. Source: (ISO Survey, 2021).

Year	Total Worldwide	Total in EU	Total in PT	Growth Rate Worldwide	Growth Rate in EU	Growth Rate in PT
2004	2403	1308	1			
2005	5065	2830	20	111%	116%	1900%
2006	8026	3574	22	58%	26%	10%
2007	12985	7049	28	62%	97%	27%
2008	13234	7463	11	2%	6%	-61%
2009	16425	9008	17	24%	21%	55%
2010	18834	11034	24	15%	22%	41%
2011	19849	10515	28	5%	-5%	17%
2012	22317	12232	38	12%	16%	36%
2013	25655	13203	62	15%	8%	63%
2014	26280	12884	65	2%	-2%	5%
2015	26255	12366	68	0%	-4%	5%
2016	29585	14705	89	13%	19%	31%
2017	31520	16341	60	7%	11%	-33%
2018	19472	10350	66	-38%	-37%	10%
2019	23045	11774	59	18%	14%	-11%
2020	25656	11798	62	11%	0%	5%

By analyzing the data from table 2 and figure 4, the conclusion is that:

- In the last ten years worldwide certifications, the growth rate was around 36%, while in the EU, it was around 7%, and in Portugal around 158%.
- The impact of this certification in Portugal compared with the EU was around 1%.
- The impact of this certification in the EU compared with worldwide was around 46%.

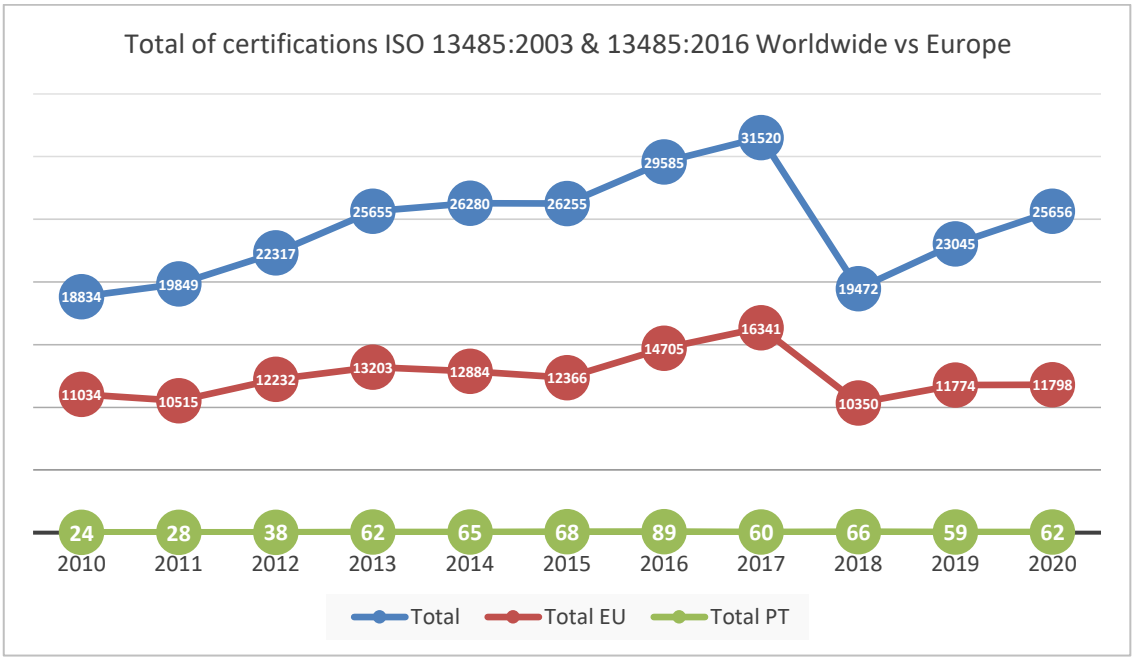

Figure 4 – Total of certifications ISO 13485:2003 & 13485:2016 Worldwide vs Europe.

2.4.1 Requirements

The following subsections describe the IT requirements and generally associated requirements mentioned in the ISO 13485:2016. Table 3 resumes these requirements.

Table 3 – IT requirements and others by association mentioned in the ISO 13485:2016.

Item	Title
4.1.6	General Requirements
4.2.4	Control of documents
4.2.5	Control of records
6.3	Resource management - Infrastructure
7.5.6	Product realization - Production and service provision - Validation of processes for production and service provision
7.6	Control of monitoring and measuring equipment

2.4.1.1 General Requirements

In item 4.1.6 of the norm is mentioned that (ISO 13485, 2016, p17):

- The organization needs to document procedures for validating computer software used in the quality management system.

- These software applications must be validated before initial use and after changes to the software or its functional requirements.

- The approach and activities with the validation and revalidation of the software must be proportional to the risk associated with software usage.

- all records must be maintained and refers to item 4.2.5 of the norm.

2.4.1.2 Control of documents

In item 4.2.4 of the norm, the organization needs to define a period to store the outdated documents. This period defined must assure that the documents are available during, at least, the lifecycle of the medical device (ISO 13485, 2016, p18).

This item refers to documenting, controlling, and how the flow should work, pointing directly to the Quality Management System.

2.4.1.3 Control of records

Item 4.2.5 mentions how to preserve the records, providing evidence of compliance with the norm's requirements.

These records controls should (ISO 13485, 2016, p19):

- Document procedures needed for the identification, storage, security and integrity, retrieval, retention time, and disposition of records.

- Define and implement methods for protecting confidential health information by following applicable regulatory requirements.

- Remain legible, identifiable and retrievable. Also, any changes to a record shall remain trackable.

- Retain the records for at least the lifetime of the medical device, defined by the organization or regulatory requirements but not less than two years of medical device release.

2.4.1.4 Resource management - Infrastructure

Item 6.3 of the norm states that an organization should document the requirements of their infrastructures needed to achieve conformity with the product. In this item, infrastructure includes process equipment, hardware or software, and supporting services like information systems (ISO 13485, 2016, p22).

It also states that the organization should present a document for the requirements regarding maintenance of the infrastructure. This document needs to include the period between them or their absence. These records need to follow the guidelines on item 4.2.5 mentioned earlier.

2.4.1.5 Product realization - Production and service provision - Validation of processes for production and service provision

In item 7.5.6 is stated that the organization needs to validate the application or applications used in production or service provision. These applications need to be validated on their first use and after any change. These procedures and activities associated with software validation and revalidation need to be proportional to a risk associated with the use of the software. These records must be maintained using the guidelines on items 4.2.4, and 4.2.5 mentioned previously (ISO 13485, 2016, p29-30).

2.4.1.6 Control of monitoring and measuring equipment

Item 7.6 states that the organization needs to do the same as item 7.5.6, but in this case, for the software used to monitor and measure if the requirements are met (ISO 13485, 2016, p31-32).

2.4.2 Other ISOs mentioned

On the document of the ISO 13485:2016 is mentioned other ISO and Technical Report as support for compliance with the norm.

These other ISOs are:

- ISO TR 80002-2:2017 – Medical device software – Validation of software for MD quality systems (ISO/TR 80002-2:2017 2017).

- ISO 14971:2019 - specifies terminology, principles, and a process for risk management of medical devices, including software as a medical device and in vitro diagnostic medical devices (ISO 14971:2019, 2019).

2.5 Infrastructure and Software

This section presents the company's physical infrastructure and software used for production.

2.5.1 Physical Infrastructure

The company has a data center inside the factory composed of three racks: one for the networking, one for the development servers and storage, and finally, one for the production environment.

The networking equipment is mostly Cisco, and the firewall is composed of two Sophos in High Availability mode.

The development environment is composed of an HP ProLiant G8 server connected to an HP P2000 storage, with a Hyper-V environment, to create as many Virtual Machines as needed.

The production environment is composed of a fail-over cluster of two nodes (HP ProLiant G9) with Microsoft Windows Server 2022 Datacenter Operating System. This cluster connects to an IBM FlashSystem 5000, where all services and applications reside in a Hyper-V environment. A simplified representation is in figure 5.

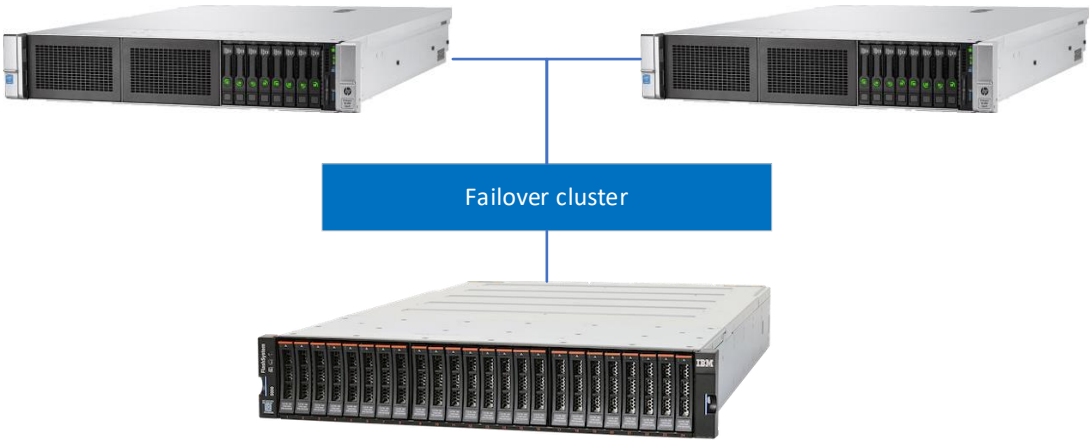

Figure 5 – Nimco production environment.

The backup server also contains a server inside the Hyper-V with the secondary Active Directory node. This server is outside of Nimco's infrastructure domain. It connects to a QNAP NAS, where all the backups reside.

Finally, the website and production system are outside the company data center, on an external server subcontracted to a partner.

2.5.2 Backups

The backups are made internally by using the Veeam backup software. This software connects to an external data center to store these backups as redundancy.

2.5.3 General Software

2.5.3.1 Microsoft Dynamics NAV

Microsoft Dynamics NAV is the company ERP for handling accountancy, HR, purchasing, and inventory.

Currently, the company has two installations of NAV, one with localization of the Netherlands and one with the localization of Portugal.

2.5.3.2 Microsoft Office 365

All the company mailboxes are on Office 365.

2.5.3.3 Microsoft Dynamics 365

Microsoft Dynamics 365 or CRM, inside the office 365 solution, is the software where the sales team does their opportunities analysis and customers management. Also, the company's customer service and quality departments handle the customer's claims.

2.5.3.4 XL-ENZ

XL-ENZ is an ERP developed by Reflecta automation. The company in the Netherlands uses it to manage the stock collection line.

2.5.4 Production software

2.5.4.1 Production System

The software technologies/programming languages are:

- The PHP programming language - Currently on version 8.1.
- The database is in the MySQL engine - Currently on version 8.0.

The software is the base of the company's operations where:

- All the customer's orders are stored.
- All the customer's specifications and protocols are stored.
- The quality department handles customers' claims.
- Products specifications are defined.
- Physical storage of standard lasts is defined.
- The production planning, production stages, and production sheet are defined.

The company website is coupled to the production system software and was initially bought and developed by an external partner and embraced by the IT department.

According to the IT department, the software lacks automated validation or tests, making it harder to ensure everything is properly working without flaws.

2.5.4.2 Lasts Software

Currently, the lasts department uses four software, with a specific purpose:

- For the modeling of lasts uses the Rhinoceros software and a plugin tailor-made for shoemakers: FootMILL.
- To scan the customer's lasts, use the Creaform VXelements software.
- To produce insoles is used the Paromed software and a Paromed Milling Machine.
- To produce lasts in the milling machine, uses Visi software.

2.5.4.3 Modelling and Cutting Software

Currently the modelling department uses a solution produced by the company Mind composed by the following software:

- Mind Cad 2D – to create the modeling plan of the shoes.
- Mind Cad 3D – to create a 3d representation of the shoe.
- Mind Cut – to cut the leathers in the cutting machines.
- Mind PDM – to sort the orders leathers by amount into a batch to get productivity gains regarding leather swap from the cutting table.
- Mind Cad Lasts – to take out measurements from the last and take the 2D plan from the 3D last.

The stitching department uses software to create embroideries and print on leathers using a laser machine with proprietary software.

2.6 Status of the QMS

Currently, the company is certified in the ISO 9001, ISO 14001, and ISO 45001. After a thorough analysis, the QMS is fully implemented and meets the requirements of the ISO 13485:2016.

Currently, the IT department has the internal processes documented and controlled by the QMS. Table 4 lists all these documents.

Table 4 – Procedure's list of the IT department in the QMS of Nimco.

IM.ICT.01-02	New user creation
IM.ICT.02-02	Responsibility terms
IM.ICT.03-02	Disabling a user
IT.ICT.01-02	Disaster recovery plan
IT.ICT.02-02	IT systems usage policies
IT.ICT.03-02	File Storage
IT.ICT.04-02	Password policies
IT.ICT.05-02	Users and computers naming
IT.ICT.06-01	VPN Access

2.7 Risk assessment

Due to the increase of technological complexity of medical devices, the methods for determine and control the risks are increasingly essential to guarantee the safety of its users. The protection of the patient, user, or any individual in contact with the product is essential. Therefore, it is necessary to emphasize the risks associated with the entire product life cycle.

In Annex I of the MDR, the essential requirements are described to reduce and eliminate risks associated with the design, use, transport, and all the type of action that involves handling the device.

There is currently an ISO addressed to medical devices manufacturers, explaining the procedures adopted for this process, its implementation, and compliance with the regulations. Its quality can be guaranteed by the CE Marking and the ISO in focus on this work, the ISO 13485:2016 about quality management systems for medical devices.

2.7.1 ISO 14971:2019 - Application of risk management

Risk management is a complex subject because each stakeholder can place a different value on the acceptability of risks concerning the anticipated benefits. Medical devices' risk management concepts are essential because of various stakeholders, including medical practitioners, organizations providing health care, governments, industry, patients, and public members (ISO 14971:2019, 2019).

The process described in the ISO 14971:2019 intends to assist manufacturers of medical devices in identifying the hazards associated with the medical device. This process also involves estimating and evaluating the associated risks, controlling these risks, and monitoring the effectiveness of the controls in production and post-production.

These requirements apply to all phases of the life cycle of a medical device and apply to several risks associated with a medical device. However, the focus of this dissertation are data and systems security.

The risk management process shall include the following elements (ISO 14971:2019, 2019):

- Risk analysis.
- Risk evaluation.
- Risk control.
- Production and post-production information.

A schematic representation of the risk management process is in figure 6.

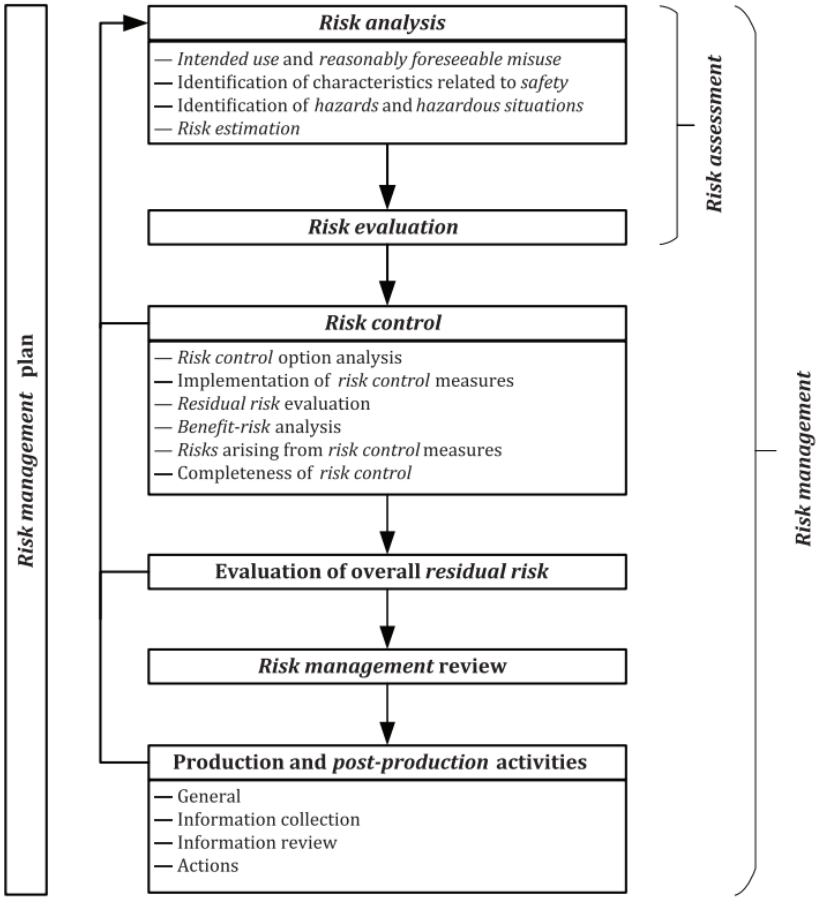

Figure 6 – A schematic representation of the risk management process. Source: (ISO 14971, 2019).

The following subsections describe each element and the company's criteria used for this process.

2.7.1.1 Risk analysis

ISO 14971:2009 does not specify what method to use for risk estimation, but it requires an estimation. Risk estimation for each danger/problem identified and definition is through two components (ISO 14971:2019, 2019):

- The probability of occurrence.
- The consequences.

For this risk analysis estimation, it was decided with Nimco to use a qualitative analysis by using a matrix, where M is the probability, and N is the consequence associated with harm.

For the probability of occurrence of harm were defined four levels of probability, presented in table 5, from the most frequent to the least, and for the consequences associated, were defined also four levels, presented in table 6, from the most severe to the least.

Table 5 – Probability of occurrence of harm levels.

Weight	**Probability**	**Description**
4	Very High	Probability of occurring frequently.
3	High	Probability of occurring a lot of times.
2	Medium	Probability of occurring sometimes.
1	Low	Probability of occurring is so low that may never happen in the lifetime of an item.

Table 6 – Consequences associated to harm levels.

Weight	Consequence	Description
4	Unacceptable	Serious and probably irreversible injury or death to the patient/user. Serious damage to the environment. Critical to Medical Device performance or customer's trust.
3	Serious	Severe but reversible injury to the patient/user. Significant damage to the environment. Negative consequence on the performance of the Medical Device or customer's trust.
2	Medium	Moderate harm to the patient/user. Moderate harmful damage to the environment. Declining performance of the Medical Device or lack of trust by the customers for the product.
1	Low	Low/no harm for patient/user. Low consequence on the environment. Low consequence on the performance of the Medical Device or customer's trust.

2.7.1.2 Risk evaluation

During the risk evaluation, the company needs to evaluate the estimated risk and determine if the risk is acceptable for each situation identified as hazardous. The company determines the risk acceptability by using the criteria defined in the risk management plan. If the risk is considered acceptable, it is unnecessary to apply any of the requirements mentioned on this norm and treat it as a residual risk (ISO 14971:2019, 2019).

Nimco used a Risk Matrix by multiplying the risks estimated in table 5 and in table 6 to determine if it is necessary to take any actions to control and reduce its risk. Table 7 presents this Risk Matrix.

Table 7 – Risk evaluation.

		Consequence			
		Low	**Medium**	**Serious**	**Unacceptable**
Probability		**1**	**2**	**3**	**4**
Very High	**4**	4	8	12	16
High	**3**	3	6	9	12
Medium	**2**	2	4	6	8
Low	**1**	1	2	3	4

The estimated values in table 7 are divided, according to type of risk, into four colored delimited areas:

1. Green determines an acceptable risk.

2. Yellow determines a manageable risk, but the company should create actions to reduce the risk probability, as needed.

3. Orange determines serious risks and the company needs to implement actions to reduce and control the risk as soon as possible.

4. Red is an unacceptable risk, and the company must address it immediately.

Table 8 presents the classification given to the risk, divided into four categories.

Table 8 – Risk levels.

Rank	**Risk Level**
Low	Risk < 4
Medium	4 ≤ Risk ≤ 8
High	8 ≤ Risk ≤ 12
Very High	Risk > 12

2.7.1.3 Risk control

Often, there will be more than one way to reduce risk. This norm lists three mechanisms, and its order priority list is essential (ISO 14971:2019, 2019).

The three mechanisms listed are (ISO 14971:2019, 2019):

1. The first and most important option in risk control analysis is safe design and manufacture because design solutions inherit the characteristics of the medical device.

2. When the first is not duable, protective measures such as barriers or alarms are appropriate.

3. Finally, the third option is to provide information for safety like warnings or contra-indications.

After all the above mentioned measures have been verified and applied, it is also necessary to evaluate if the residual risk is acceptable. This evaluation ensures that the risk analysis still stands.

2.7.1.4 Production and post-production information

Risk management does not stop when a medical device goes into production. It usually begins with an idea, and then companies collect information from several sources. The risk estimation

is refined throughout the design process and made more accurate when the prototype is built (ISO 14971:2019, 2019).

Therefore, the company needs to collect and review production and post-production information to evaluate its importance to safety. With effective production and post-production activities, the risk management process truly becomes an iterative closed-loop process to ensure the safety of the medical device (ISO 14971:2019, 2019).

2.7.2 Company risk assessment

The top management is responsible for the risk management and delegates it to a nominated management team, the MD responsible technician, and the management team. This nominated team ensures the risk assessment regarding the products and production process.

This subsection presents the risk assessment made with the IT department to the IT infrastructure and software.

2.7.2.1 IT risk assessment

During the IT risk assessment, the IT department listed twenty-two issues that could be a risk to the company with the following risk evaluation:

- Two issues with High Risk level.
- Three issues with Medium Risk level.
- Seventeen with Low Risk level.

The IT department gave the weights to these identified issues and defined strategies to reduce the risk or solve the issue. Table 9 lists the high risk and medium risk level issues and weights.

Table 9 – IT risk assessment.

Issue Number	Activity	Danger / Dangerous Situation	Risk	Probability Weight	Consequence Weight	Risk Evaluation
1	Material traceability	Bulk traceability	Company has no way of guaranteeing traceability of materials per batch	4	3	12
2	Production system validation after system changes	Failure/lack of validation of the production system after system changes	No computer bug detection	2	3	6
3	Create customers in the various ERP's	Manual process carried out by the finance department at NL and PT	Inconsistent information in customer creation	2	3	6
4	Customer technical specifications	System is not prepared for an order to make visible the alerts that are related to the order placed by the customer, showing all customer alerts in production sheet	Customer alert failure	2	3	6
5			Failure to comply with the customer's order	2	3	6

For each issue in table 1, the IT and Quality departments determined the consequence, the control measure, decision, and status.

Table 10 – IT risk assessment – Issue #1.

Consequence	In claims, it is not possible to identify which batch was used in the manufacture of the product in question
Control Measure	Implementation of a batch-to-order traceability system
Decision	Implementation of a batch-to-order traceability system
Status	Planned

Table 11 – IT risk assessment – Issue #2.

Consequence	Computer error
Control Measure	Make validation records
Decision	Implementation of unit test system after developments
Status	Ongoing

Table 12 – IT risk assessment – Issue #3.

Consequence	Incorrect shipments, incorrect contacts
Control Measure	Validation of information by the finance department in Portugal
Decision	Implementation of a single customer creation system
Status	Ongoing

Table 13 – IT risk assessment – Issue #4 and #5.

Consequence	Non-compliant product
Control Measure	Control on the production stages
Decision	Implementation of conditional alerts system on demand
Status	Ongoing

The IT and Quality department handles the first, third, fourth and fifth issues. The second issue is in the scope of this dissertation, and chapter four presents the analysis and design for a solution regarding this issue. Chapter five presents the implementation of the solution presented in chapter four.

2.8 Requirement's analysis

According to the applicable norm (ISO 13485:2016), in the table 14 is listed all the information regarding the assessment of Nimco's infrastructure and it's QMS. It was added a column to indicate what is the status of compliance with the items in the norm, "compliant" when it's fully compliant and "non-compliant" when is not totally compliant.

Table 14 – ISO 13485:2016 requirements assessment.

4.1.6	General Requirements	Compliant
4.2.4	Control of documents	Compliant
4.2.5	Control of records	Compliant
6.3	Resource management - Infrastructure	Compliant
7.5.6	Product realization - Production and service provision - Validation of processes for production and service provision	Non-compliant
7.6	Control of monitoring and measuring	Non-compliant

By analyzing the previous table, table 14, we can understand that there are currently two areas that need to be addressed by the company: items 7.5.6 and item 7.6. This dissertation addresses these items but may not be a critical factor for the company getting the certification.

The IT department lacks documentation and a testable product of the production system, making it harder to ensure the changes do not affect the application quality. An agile test methodology could solve the problem making it easier to comply with items 7.5.6 and 7.6.

Experiment results to evaluate the efficacy of TDD are statistically inconsequential. The measurement of this efficacy of TDD relates to (Khanam et al., 2017):

- McCabe's Cyclomatic complexity.
- The number of lines of code written.
- The number of acceptance test cases passed.
- Branch coverage.
- The number of user stories implemented per person-hours.

Even though test dependencies lead to complications, writing tests during the development process remained consistent (Khanam et al., 2017).

Experiment results to evaluate the efficacy of TDD in suggests that the statistics result related to McCabe's Cyclomatic complexity, number of lines of code per, number of acceptance test cases passed, branch coverage, person hours, number of user stories implemented per person hours are statistically inconsequential (Khanam et al., 2017). Even though test dependencies lead to complications, the process of writing tests during the development process remained consistent (Khanam et al., 2017).

2.9 Agile

Agile is a wide umbrella of software development beliefs. It is a conceptual framework for software engineering that begins with a starting planning phase and follows the road toward the deployment phase with iterative and incremental interactions throughout the life cycle of the project. The initial goal for the agile methods is to reduce the overhead in the software development process with the ability to adopt the changes without risking the process or without excessive rework (Al-Saqqa et al., 2020).

2.9.1 Continuous Integration and Continuous Quality

Continuous Integration is a software development practice where team members integrate their work frequently. Usually, each person integrates at least daily, leading to multiple integrations per day. An automated build verifies each integration (including test) to detect integration errors quickly (Fowler, 2006).

Continuous integration at its foundations has the process of an automated deployment pipeline. It is, in essence, the principle taken to its logical conclusion (Fowler and Foemmel, 2006).

The aim of this deployment pipeline divides into three goals (Fowler and Foemmel, 2006):

- Every part of the building, deploying, testing, and releasing software becomes visible to everybody involved, aiding collaboration.
- Feedback is improved to identify and resolve the problems as early as possible.
- Enables the team to deploy and release any version of the software to any environment at will through a fully automated process

2.9.2 Agile methodologies

Below are the agile methodologies studied for this dissertation.

2.9.2.1 Scrum

Scrum methodology is an immaterial group of patterns used broadly inside the business equally portrayed near the Scrum Alliance (Saleh et al., 2019).

The three sections of a Scrum assemble are (Saleh et al., 2019):

- Scrum Master (SM).
- Product Owner (PO).
- Development Team.

In Scrum, from each one gathering deals their assignments by the strategy for four collectibles: a Product Backlog, a Sprint Backlog, a Product Increase, and Definition of Done (Saleh et al., 2019).

Scrum is, in this manner, seen as a prescriptive procedure. The Scrum gather is required to execute five actions to attain its destinations. These actions are Backlog Elaboration, Sprint Planning, Daily Scrum meetings, Sprint Reviews, and Sprint retrospectives (Saleh et al., 2019).

2.9.2.2 Kanban

Kanban is contrary to Scrum and is a method that can manage the challenges and winds up being even more intense for bunches in passing on business regard additive. Also, it is highly subject to self-dealt with gatherings. Moreover, its system requires the rare condition of the expert relationship taking the shape of various practices, which encourage the Agile use in the affiliation (Saleh et al., 2019).

2.9.2.3 Extreme Programming (XP)

XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements (Bell, 2001). Also is considered as a discipline of software development based on values of simplicity, communication, feedback, and courage, by bringing the whole team together in the presence of simple practices, with enough feedback

to enable the team to see where they are and to tune the practices to their unique situation (Lindstrom & Jeffries, 2003).

The XP methodology is composed of four values that guide the programmers' actions thorough the projects (Lindstrom & Jeffries, 2003):

- Communication.
- Simplicity.
- Feedback.
- Courage.

This methodology comprises 12 practices, shown in figure 7, that guide new teams that use XP to focus on using and developing skills with these practices (Lindstrom & Jeffries, 2003).

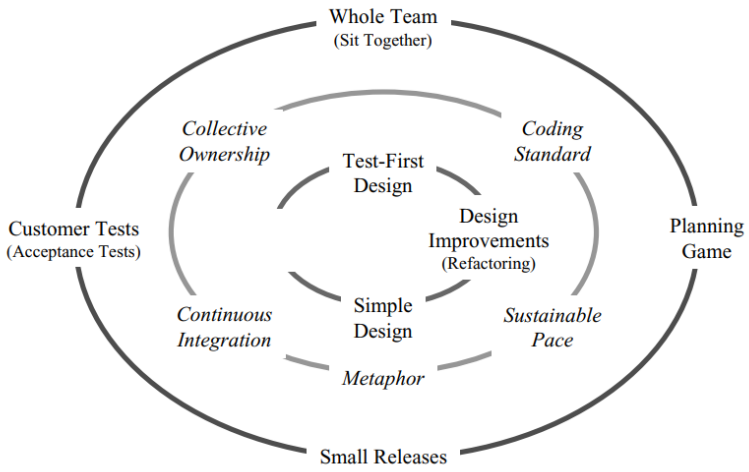

Figure 7 – XP Practices and the Circle of Life. Source: (Lindstrom & Jeffries, 2003).

2.9.3 Agile testing methodologies

TDD, BDD, and ATDD are agile software development frameworks and are unit testing approaches. TDD, BDD, and ATDD are software development techniques that use unit tests to incrementally deliver small pieces of functionality (Moe, 2019).

In the subsections below are detailed each of these methodologies.

2.9.3.1 Test Driven Development

Test-Driven Development (TDD) consist of (Al-Saqqa et al., 2020):

- Building a small, iteratively automated testing program.

- Write the code that can pass that test.
- Leave the enhancement of that code to be done later.

Other traditional software development methods prefer to do the test after code completion, while TDD is the opposite (Al-Saqqa et al., 2020).

TDD has two main rules (Anwer et al., 2017):

- If the test fails, then write a code to solve it.
- Do not make duplications in the code.

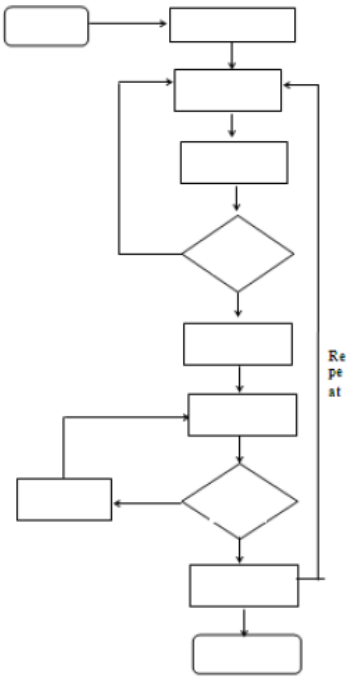

Figure 8 – Acceptance test flow in TDD. Source: (Moe, 2019).

TDD is an agile model gaining popularity among developers due to its benefits. Here are some of the advantages of TDD (Anwer et al., 2017):

- Test first approach of TDD helps find defects earlier and near their origin, reducing defect rate and cost of debugging considerably. In traditional development methods, programmers perform the testing activity late. In a later stage of development, defect debugging, and maintenance becomes more dull that may cause more defects injection in software. Due to the incremental approach and quick feedback from testing, the external quality of code becomes significantly enhanced.

- TDD improves the efficiency as it works in small iterations. In each iteration, tests provide quick feedback whenever programmers add new code, helping find and correct errors earlier and stopping defect propagation.

- Automated tests written during TDD iterations are beneficial in enhancing the code quality.

- TDD works in small iterations that divide the overall development into smaller, more manageable parts providing an opportunity to concentrate and develop more effectively.

- Code refactoring improves software design-related issues. TDD helps in writing code having loose coupling and high cohesion.

- Programmers write simple classes and modules to implement a small functionality that reduces code's size and complexity.

TDD is a good development approach, but there are some limitations also. On the following list are some of those limitations (Anwer et al., 2017):

- TDD is a disciplined approach that requires some unique skills (like writing test cases, which is usually the tester's duty) that programmers feel challenging to practice.

- On software projects that require synchronization, the developer must be able to check whether TDD can be applied or not.

- Managing test suits is another problem with TDD. Proper maintenance is required to use these test suits.

- TDD lacks documentation, making it more challenging in the maintenance process.

- Sometimes, TDD becomes more time-consuming because of repeated test failures.

- TDD does not provide any guidance about management aspects of software projects. It only focuses on engineering-related activities.

2.9.3.2 Behavior Driven Development

The Behavior-Driven Development (BDD), introduced by Dan North and inspired by Kent Beck's Test-Driven Development (TDD), is an outside-in agile software development process frequently used in Extreme Programming and microservices development. BDD uses a business readable and Domain Specific Language (DSL) called Gherkin to describe software behaviors while hiding its implementation details. The acceptance test scenarios follow a simple syntax (Rhaman & Gao, 2015):

- Given – some initial context

- When – an event occurs

- Then – ensure some outcomes.

In BDD, the user initially gathers software requirements or behaviors using scenarios with the above format where the lines with "Given," "When," and "Then" keywords are known as "Step." These steps are parsed and executed by a BDD test framework to verify the software expectation, also known as Automated Acceptance Tests. These scenarios act as a powerful means of communication among software developers, testers, and analysts to improve the quality of the software through better understanding (Rhaman & Gao, 2015).

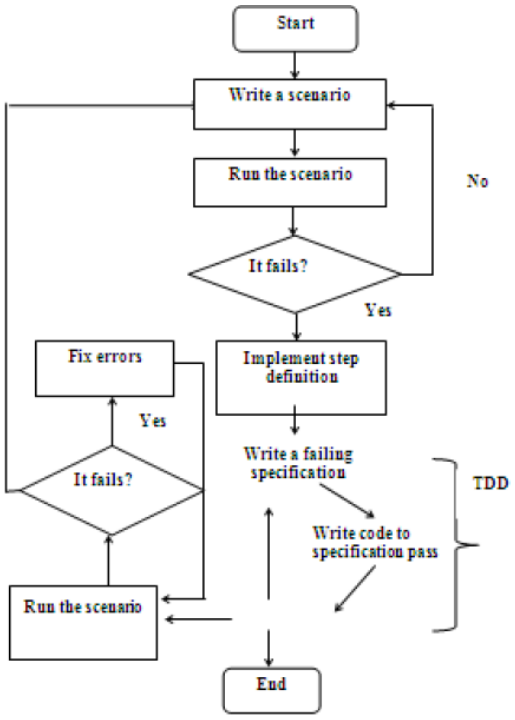

Figure 9 – Acceptance test flow in BDD. Source: (Moe, 2019).

A few points will benefit the software team if it plans to implement BDD. These are some advantages (Moe, 2019):

- The software team is no longer defining 'test' but is defining 'behavior.'
- Better communication between developers, testers, and product owners.
- The learning curve is much shorter because BDD uses a simple language.
- Being non-technical, it can reach a wider audience.
- The behavioral approach defines acceptance criteria prior to development.

Some of disadvantages of BDD (Moe, 2019):

- Recommended prior experience with TDD
- BDD is incompatible with the waterfall approach.

- Testers using BDD need to have appropriate technical skills. If the requirements are not correctly specified, BDD may not be effective.

2.9.3.3 Acceptance Test-Driven Development

Acceptance test-driven development (ATDD) is a development methodology based on communication between the business customers, the developers, and the testers. The ATDD process follows these steps (Moe, 2019):

1. Select user story
2. Write acceptance test
3. Implement user story
4. Run acceptance test
5. Make slight change/refactor

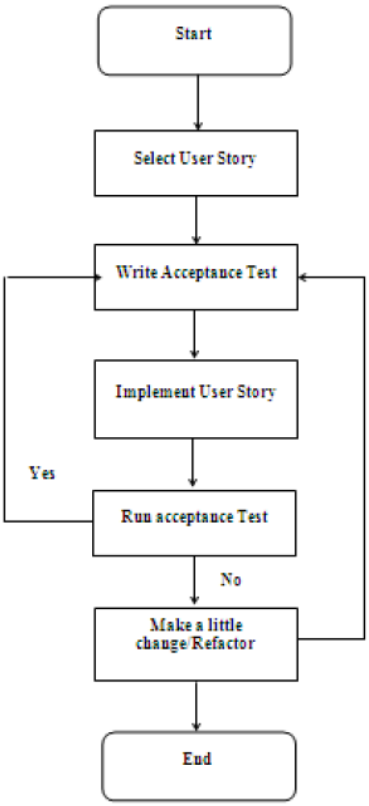

Figure 10 – Acceptance test flow in ATDD. Source: (Moe, 2019).

Some of the advantages of using the approach can be these (Moe, 2019):

- Improve communication and collaboration between project stakeholders.

- A shared understanding of what a successful implementation means.
- Better coverage of business expectations.
- Faster feedback.

Some of disadvantages of ATDD (Chen, 2022):

- ATDD is less flexible because it becomes harder to change requirements after acceptance tests are written and coding has begun.
- ATDD is also process-heavy and may result in multiple tests with only minor differences.

ATDD is a new methodology that requires rigor and discipline. Also, to use this approach, the user needs to find the right balance between people, processes, and tools (Moe, 2019).

2.10 Approach Strategy

In conclusion, about state of the art and to solve the problem, was decided:

- Maintain and use Scrum methodology, which the IT department already uses
- Implement a Test-Driven Development methodology; in this case, the IT department decided to use BDD.
- Use a careful approach and apply BDD to the production system.
- By using a test methodology for the production system, the company can ensure that the codebase is thoroughly tested and passed the standard regarding software validation.

3 Value Analysis and Proposition

Value analysis is a process of analysis and evaluation, intending to obtain a set of necessary functions in a project at the lowest possible cost (Rich and Holweg, 2000). By this definition, value analysis concerns the function of a product to meet the demands or applications needed by a customer. The review process must include an understanding of the product's purpose. Also, this analysis must result in design improvements to lower the production costs while maintaining the value (Rich and Holweg, 2000).

As shown in Figure 11, value analysis consists of five steps: orientation, identification and functional analysis, creative alternatives, analysis and evaluation, and implementation.

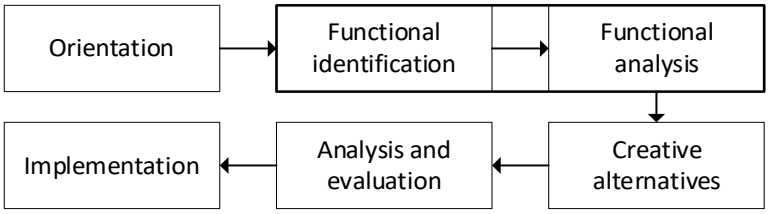

Figure 11 – Value analysis process.

For each step mentioned, we can apply different techniques. The following sections describe:

- The Fuzzy Front End.
- The value analysis for the company.
- The Analytic Hierarchy Process to assist in choosing the best methodology of testing.
- The value proposition by using the Business Model Canvas.

3.1 Fuzzy Front End

The innovation process consists of the fuzzy front end (FFE), the new product development (NPD) process, and commercialization (Koen et al.,2002). The first part regards one of the most significant opportunities to improve the overall innovation process.

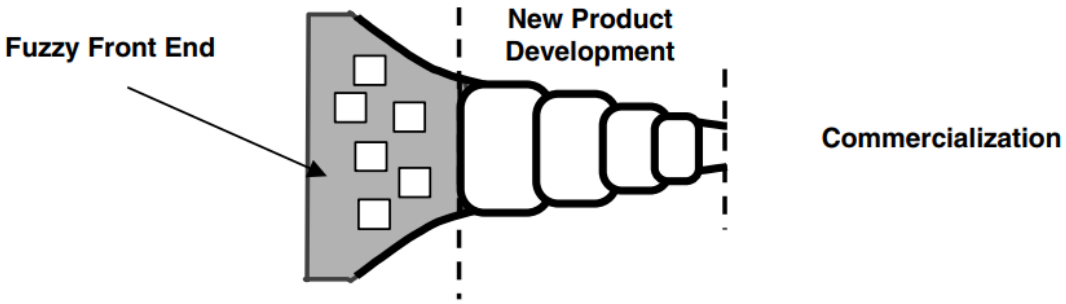

Figure 12 – Diagram of the innovation process. Source: (Koen et al., 2002).

According to (Koen et al., 2002), while the fuzzy front end may be unreliable and unpredictable, the new product development phase is more specific.

Koen provided a common language and terminology to optimize the fuzzy front end and turn it into a "front end of innovation." This common language is the New Concept Development Model (NCD), with the clear objective of cultivating ideas and concepts (Koen et al., 2002).

The New Concept Development model consists of three areas, as illustrated in figure 13: the engine, the five key activities, and finally, the influencing factors (Koen et al., 2002):

- The engine is the organization's leadership, culture, and business strategy that drives the five key activities.

- The Inner circle is the five activities of the Fuzzy Front End (opportunity identification, opportunity analysis, idea generation and enrichment, idea selection, and concept definition). These activities are mentioned below in detail.

- The exterior part includes organizational capabilities, external factors, and enabling sciences.

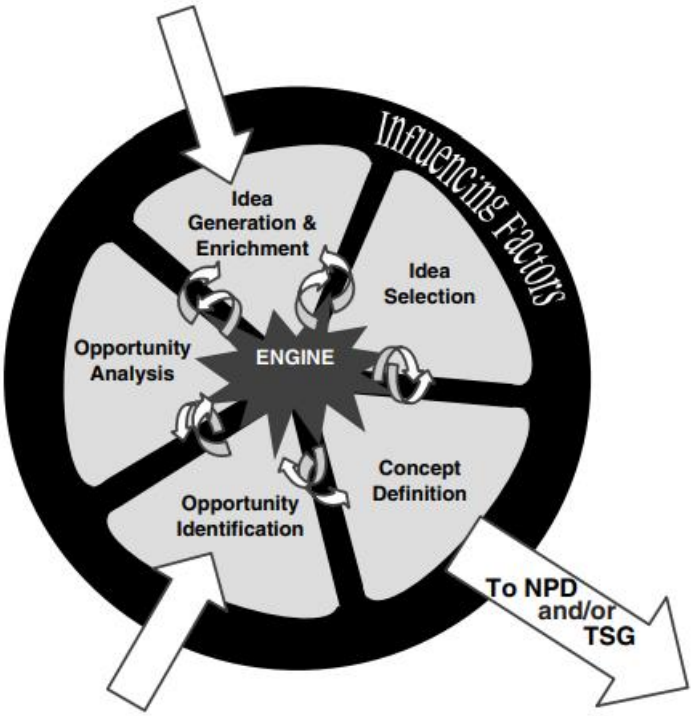

Figure 13 – NCD Model. Source: (Koen et al., 2014).

3.1.1 Opportunity identification

In this element, the focus is on identifying opportunities that the company or an individual is interested in getting into. Typically, companies consider business and technological opportunities and allocate resources to new areas of market growth, operating effectiveness, and efficiency, driven by company goals (Koen et al., 2002, p15).

One of the identified opportunities in this dissertation is the company goal of increasing market growth across the European Union.

Figure 14 presents the total health spending by the percentage of the Gross Domestic Product. This data measures the final consumption of health care goods and services, including personal and collective services, excluding spending on investments (OECD, 2021).

The markets presented in figure 14 are where Nimco currently has their biggest customers and where they want to increase their market share.

By analyzing this report by OECD is concluded that (OECD, 2021):

- The top country with the most health spending is the US, with 16,8% of the Gross Domestic Product.

- The UK is the second with 12,8%.

- Germany is third with 12,5%.
- France is fourth with 12,4%.
- The Netherlands is 9th, and Portugal is 12th, with 11,2% and 10,1% respectively.

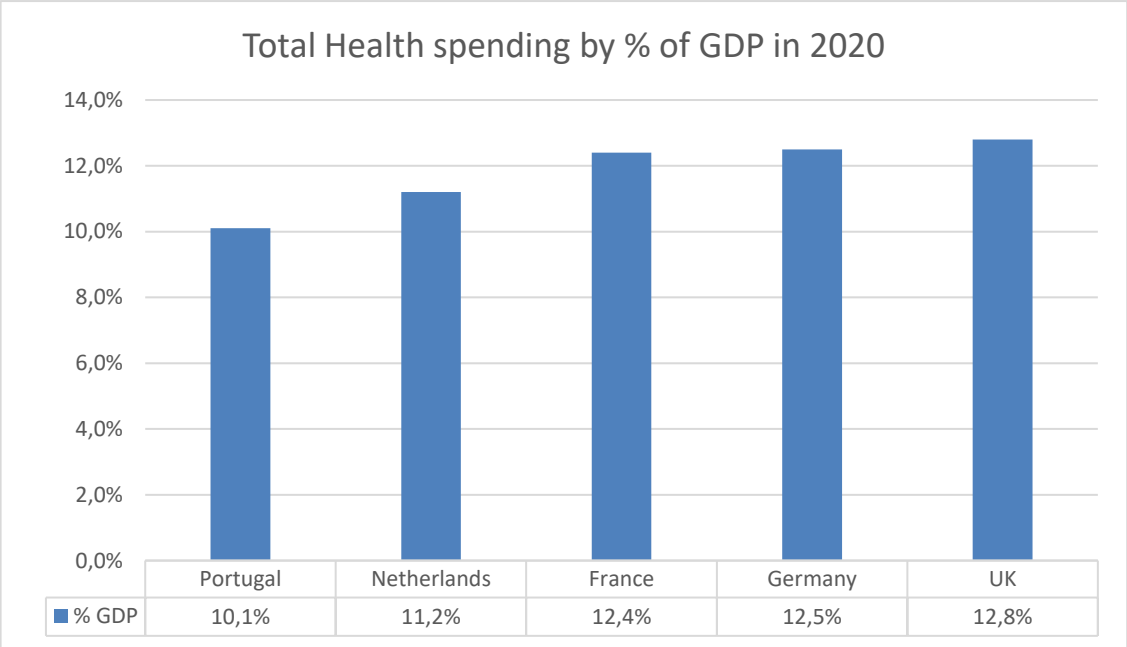

Figure 14 – Total Health spending by % of GPD in 2020. Source: (OECD, 2021).

3.1.2 Opportunity analysis

After the opportunity identification, an assessment is necessary to confirm that it is worth pursuing and is viable. To translate opportunity identification into specific business and technology opportunities, the person doing this analysis must (Koen et al., 2002, p18):

- Gather additional information, typically involving making technology and market assessments.
- Commit an extensive effort to focus groups, market studies, and scientific experiments.

A typical analysis for an opportunity would include (Koen et al., 2002, p18):

- A strategic framing - define if this opportunity fits within the company's market.
- A market segment assessment - description of a market by showing why it represents a fantastic opportunity, depending on several factors.
- Competitor analysis – determine the competitors in the market segments analyzed.
- Customer assessment – Determine the customer needs not met by current products.

For this dissertation, the opportunity analysis throughout internal meetings led to:

- Getting the ISO certification would benefit the company because it is mandatory to produce for the European Union markets.
- It would bring increased value to the company's reputation.
- It would get recognition from peers in the same field of expertise for producing safe medical devices.

3.1.3 Idea generation and enrichment

This idea generation and enrichment element are related to the development and maturation of a specific idea. This Idea generation is an evolutionary process where an idea may go through many iterations, changes are examined, studied, discussed, and developed. Normally direct contact with customers and users and collaboration with other companies and institutions often enhance this process (Koen et al., 2002, p19).

The output of the previous phase led to this phase, where the departments assess the company's compliance with the norm and identify what is compliant or not. With the help of risk analysis to the company, this assessment led to issues for posterior analysis and study in the Idea Selection phase.

3.1.4 Idea selection

After defining the potential ideas, it is necessary to select an idea to follow. The problem for most businesses is selecting which ideas to pursue to achieve the most business value, and making a good decision is critical to the future of the business (Koen et al., 2002, p22).

This phase consisted of defining a risk analysis method, analyzing and identifying the IT infrastructure and software issues, and selecting one for the next phase.

3.1.5 Concept definition

Finally, the last element is the one that can represent the exit of the frontend phase for the development of a new product. All pertinent information is gathered around the chosen idea and the impacted markets to show a compelling concept for investment in the business or technology proposition (Koen et al., 2002, p26).

In this phase, the design and architectural approach were established and ready for the development of the issue selected.

3.2 Customer Value

This section explains the meaning of Value, Perceived Value, and Value for the customer and how it relates with this dissertation to understand the concepts and advance to the value proposition.

- Value definition is needs, desire, interest, beliefs, attitudes, and preferences (Nicola et al., 2012).

- Perceived value is how a customer sees the product's utility based on perceptions of what is received and given (Zeithaml, 1988).

- Value for the Customer can be an ambiguous concept since it is defined based on how a customer interacts with a given product and its experience fulfilling its needs (Woodroof, 1997).

Four parts divide the value for the customer (Nicola et al., 2012):

- Pre-purchase – predict how people perceive the value.

- Point of trade – the quality of the product and associated costs.

- Post-Purchase – how the customer sees the delivered value from his point of view and experience.

- After-Use – Reflection about the point of sale.

Table 15 reflects how the four parts are applied in the context of this project and understand how the customer perceives the value proposition.

Table 15 – Longitudinal perspective of value.

	Benefits	**Disadvantages**
Pre-purchase	Increase production efficiency. Quality of value delivery. Costs reduction with raw materials.	Cost and time implementing the ISO 13485:2016.
Point of trade	Better inter-department teamwork. Quality of the value delivered. Identification of issues in the company.	Costs of mitigation or resolution of issues found during the assessments.
Post-Purchase	Better product market strategies. Increase company market share. Costs reduction.	
After-Use	Customers satisfaction on received products. Increased quality for the products.	

3.3 Value Proposition

The Value Proposition is how a company presents to its client segments the advantages of a given item of value and why the clients choose the company's product as it stands out from the competition (Osterwalder & Pigneur, 2003).

The value proposition in this dissertation is an implementation of the ISO 13485:2016 in the company.

This norm is regarding Medical Devices and concretely for medical devices' quality systems. This implementation on the company allows identifying potential flaws in the manufacturing processes, company systems and infrastructure, and others. This implementation will allow the company to produce and sell in the European Union because it complies with the mandatory regulation, 2017/745. Also, it will enable the obtainment of the CE marking for the products in the scope of the certification.

3.4 Canvas Business Model

The Business Model is a conceptual tool that contains a set of elements and their relationships and allows expressing the business logic of a specific company (Osterwalder & Pigneur, 2003).

Nine sections divide the Canvas Business Model that are crucial for creating the business model.

These sections are:

1. Key Partners are the essential partnerships of the business.
2. Key Activities are the company's main activities to make the value possible.
3. Key Resources are the essential resources to make the value possible.
4. Value Propositions are what will deliver the value to the end customer, a product, or service that will them solve a specific problem.
5. Customer relationships are the relationship the company expects to maintain with its customers.
6. Channels are how the company reaches the customers.
7. Customer Segments are for whom the company will deliver value.
8. Revenue Streams are how will the value delivered generate revenue.
9. Cost Structure is the related costs.

Figure 15 illustrates this business model.

<table>
<tr><td>Key Partners
Customers, The company departments involved in this implementation, The COO, Plant Manager
External Consultants
Partners</td><td>Key Activities
Assess the company to see if it is compliant with the norm.
For the issues found make risk analysis, propose and define solutions.

Key Resources
Human Resources: Quality department, Production team, ICT department, Plant Manager
Financial resources: Costs to implement resolutions to the issues found.
Physical resources: regarding production machines, IT infrastructure, hardware and software.</td><td>Value Propositions
Better product marketing strategies.
Increase company market share.
Costs reduction.
Customers satisfaction on delivered products.
Increased quality for the products.
Increased confidence and brand goodwill</td><td>Customer Relationships
Communication is done through email and Teams or in person at the customer's facilities or at Nimco's facilities

Channels
Email, Meetings, Chats</td><td>Customer Segments
Orthopedic companies that want to sell Nimco products</td></tr>
<tr><td>Cost Structure
Costs depend on the assessment and how many issues were found.</td><td></td><td>Revenue Streams
New markets
Products delivered to customers</td><td></td><td></td></tr>
</table>

Figure 15 – Canvas Business Model Diagram.

3.5 Analytic Hierarchy Process

Analytic Hierarchy Process (AHP) is used to aid in the multicriteria decision and compare the three agile test methodologies mentioned in section 2.9.

The Analytic Hierarchy Process is a mathematical method for quantitative multiple criteria evaluations that relies on numeric techniques to compare a set of alternatives with defined evaluation criteria. This calculation method is through pairwise comparisons with a scale of absolute judgments that assigns each criterion to a higher or lower level of importance (Saaty, 1990).

This method relies on the decision maker's judgment to assign the priority scale and has seven phases (Nicola, 2019):

1. Defining the problem with a Hierarchical Decision Tree.
2. Comparison of alternatives and criteria by using a comparison matrix.
3. Obtain the Relative Priority Vector for each criterion.
4. Evaluate the consistency of the decisions.
5. Pairwise comparison matrix for each criterion according to each alternative.
6. Obtain the Composite Priority Vector for the alternatives.
7. Selection of the alternative using the Composite Priority Vector.

The alternatives are:

- Test-Driven Development.
- Behavior Driven Development.
- Acceptance Test-Driven Development.

The criteria for the decision are:

- Definition.
- Participants.
- Focus.
- Language.

Figure 16 defines the problem as a hierarchical decision tree for easy understanding.

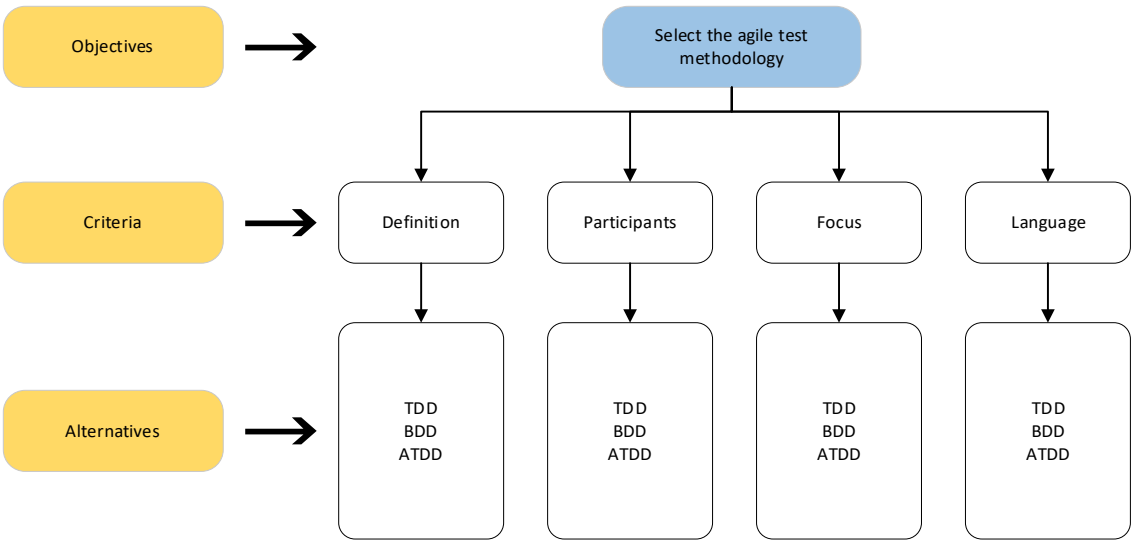

Figure 16 – Hierarchical Decision Tree - Selection of the agile test methodology.

The next step is to define the priorities of the criteria based on importance by using pairwise comparisons to differentiate the criteria against each other. Table 16 shows the prioritization made with the help of the fundamental scale and its values (Saaty, 1990).

Table 16 – Fundamental Scale. Source: (Saaty, 1990).

Intensity of importance on an absolute scale	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favor one activity over another
5	Essential or strong importance	Experience and judgement strongly favor one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is needed
Reciprocals	If activity i has one of the above numbers assigned to it when compared with activity j, then j has the reciprocal value when compared with i	
Rationals	Ratios arising from the scale	If consistency were to be forced by obtaining n numerical values to span the matrix

Table 17 shows the comparison in the context of this problem.

Table 17 – Comparison Matrix between the defined criteria.

	Learning Curve	**Participants**	**Focus**	**Language**
Learning Curve	1	1	1/8	1/7
Participants	1	1	1/8	1/7
Focus	8	8	1	2
Language	7	7	1/2	1

After the comparison, the results were normalized to match the values assigned to each criterion to the same unit.

The relative priority vector is then calculated to define the importance of each criterion over the others. Table 18 shows the obtained results.

Table 18 – Normalized Comparison Matrix and Relative Priority Vector.

	Learning Curve	**Participants**	**Focus**	**Language**	**Priority Vector**
Learning Curve	0,059	0,059	0,071	0,043	0,058
Participants	0,059	0,059	0,071	0,043	0,058
Focus	0,471	0,471	0,571	0,609	0,530
Language	0,412	0,412	0,286	0,304	0,353

According to the results, the focus of the agile testing methodology is the most critical criteria, followed by the language. Finally, the learning curve and the participants are the least critical characteristic (BrowserStack, 2022).

In the next step, to measure the consistency of the judgments, the consistency ratio (RC) is calculated. The judgments are considered reliable if the value of RC is below 0,1. (Nicola, 2019).

First, λ-max is calculated using the formula:

$$Ax = \lambda_{max} x$$

Where A is the criteria comparison matrix and x the priority vector. The result of λ-max is approximately 4,04.

With this value, it is possible to calculate the Consistency Index (IC) using the following formula:

$$IC = \frac{\lambda_{max} x - n}{n - 1}$$

Where n is the number of criteria. The result of the Consistency Index is approximately 0,013.

Then, the Consistency ratio can be calculated using the formula:

$$RC = \frac{IC}{IR}$$

IC is the previously calculated value and IR is the random consistency value for our matrix. The IR value is a constant value defined in Table 19.

Table 19 – Random Consistency Values. Source: (Nicola, 2019).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

This matrix is a 4x4 matrix, so the Random Consistency value (IR) to order four matrixes is 0,90, according to Table 19. The value of RC is approximately 0,014. Since this value is below 0,1, the judgments are considered reliable. Table 20 shows a summary of the values obtained.

Table 20 – λ-max, Consistency Index and Consistency Ratio results.

λ-max	4,04
Consistency Index	0,013
Consistency Ratio	0,014

After calculating the global priority vector, the next step is to create a comparison matrix for each criterion, considering both alternatives, to find the best approach based on the criteria. The process is done by repeating the creation of the comparison matrices, between each criterion and alternative, the respective normalization matrix, and calculating the priority vector. Table 21 shows the initial comparison using the fundamental scale presented in Table 16.

Table 21 – Comparison Matrices between Alternatives and the Criteria.

Learning Curve			
	TDD	**BDD**	**ATDD**
TDD	1	1/3	1
BDD	3	1	3
ATDD	1	1/3	1
Participants			
	TDD	**BDD**	**ATDD**
TDD	1	1/5	1/8
BDD	1	1	1/8
ATDD	7	7	1/2
Focus			
	TDD	**BDD**	**ATDD**
TDD	1	1	1/8
BDD	8	8	1
ATDD	7	7	1/2
Language			
	TDD	**BDD**	**ATDD**
TDD	1	1	1/8
BDD	8	8	1
ATDD	7	7	1/2

Then, the matrices are normalized, and the respective local priority vector is calculated. Table 22 shows the result, along with the local priority vector.

Table 22 – Normalized Matrices for Alternative and the Criteria Comparisons and Priority.

Learning Curve				
	TDD	**BDD**	**ATDD**	**Priority Vector**
TDD	0,20	0,20	0,20	0,20
BDD	0,60	0,60	0,60	0,60
ATDD	0,20	0,20	0,20	0,20
Participants				
	TDD	**BDD**	**ATDD**	**Priority Vector**
TDD	0,13	0,13	0,11	0,12
BDD	0,63	0,65	0,67	0,65
ATDD	0,25	0,22	0,22	0,23
Focus				
	TDD	**BDD**	**ATDD**	**Priority Vector**
TDD	0,17	0,11	0,20	0,16
BDD	0,33	0,22	0,20	0,25
ATDD	0,50	0,67	0,60	0,59
Language				
	TDD	**BDD**	**ATDD**	**Priority Vector**
TDD	0,59	0,56	0,60	0,58
BDD	0,12	0,11	0,10	0,11
ATDD	0,29	0,33	0,30	0,31

The results of Table 22 are then compiled into a new matrix along with the relative priority vector from Table 18. With these values, it is possible to calculate the Composite Priority Vector by multiplying the result of each alternative by the priority vector. This vector will represent the importance of the alternative based on the criteria. The results are shown in Table 23.

Table 23 – Criteria/Alternatives Classification Matrix and Composite Priority.

	Learning	**Participants**	**Focus**	**Language**	**Global**
TDD	0,20	0,16	0,12	0,58	0,29
BDD	0,60	0,25	0,65	0,11	0,43
ATDD	0,20	0,59	0,23	0,31	0,28

According to the AHP method and the respective results of the global priority vector from Table 23, behavior-driven development is the approach of choice, followed by Test-Driven Development and Acceptance Test-Driven Development.

4 Solution Design

In this chapter, the solution design for implementing software validation into the codebase of the production system is described by considering the functional and non-functional requirements for the solution for existing functionalities that will change or adapt to be fully testable.

The following steps define the technologies and tools used to accomplish the objectives and design the solution's architecture.

4.1 Technologies

The objective is to implement behavior test-driven development studied in the previous sections into the existing production system. This subsection introduces the status of the production system, other supporting tools used, the preferable architecture of the system by introducing four design patterns used in the development, the SOLID Design Principles, and finally, the case study to achieve the objectives.

4.1.1 Production system

The production system is a monolithic application developed in PHP.

A monolithic application is one with a single layer comprising everything that the application does, with each different concern squished together. These applications are not modular and thus do not provide reusable code and are often hard to maintain (Wilson, 2015).

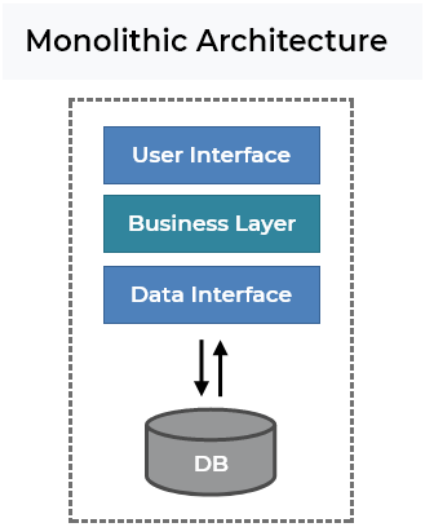

Figure 17 – Monolithic Architecture. Source: (Domareski, 2021).

4.1.1.1 Programming language

Currently, the application is running in PHP 8.1, released in 2021 (PHP Supported versions, n.d.).

PHP is a general-purpose scripting language geared toward web development. Danish-Canadian programmer Rasmus Lerdorf created it in 1994 (Wiki PHP, 2022).

4.1.1.2 Front-end/Back-end

The application's User Interface (UI) comprises Bootstrap Framework, JavaScript, jQuery, and CSS.

- Bootstrap is a free and open-source CSS framework directed at responsive, mobile-first front-end web development. It contains HTML, CSS and JavaScript-based design templates for typography, forms, buttons, navigation, and other interface components (Wiki Bootstrap, 2022).

- JavaScript (JS) is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions (JavaScript, n.d.).

- jQuery is a fast, small, and feature-rich JavaScript library. It simplifies things like HTML document traversal and manipulation, event handling, animation, and Ajax with an easy-to-use API that works across many browsers (jQuery, n.d.).

- Cascading Style Sheets (CSS) is a stylesheet language used to describe the presentation of a document written in HTML or XML. CSS describes how elements should render on screen, paper, speech, or other media (CSS: Cascading style sheets, n.d.).

4.1.1.3 Database

The database of the application is in MySQL and is in version 8.0.

MySQL is an open-source relational database management system (RDBMS). SQL is a language programmers use to create, modify and extract data from the relational database and control user access to the database. A relational database organizes data into one or more tables in which data may relate to one another, and these relations help structure the data (Wiki MySQL, 2022).

4.1.1.4 Composer

Composer is used to managing all external dependencies of the production system.

Composer is a tool for dependency management in PHP. It allows you to declare the libraries your project depends on, and it will manage (install/update) them for you (Composer, n.d.).

4.1.2 Git

The application uses Git as the version control system (VCS), introduced to the project in 2017.

Git is a free and open-source distributed version control system designed to handle everything from small to massive projects with speed and efficiency (Git, n.d.).

The git project structure comprises two main branches, "master" for production and "qa" for quality assurance (QA). All other in-development features have their branch.

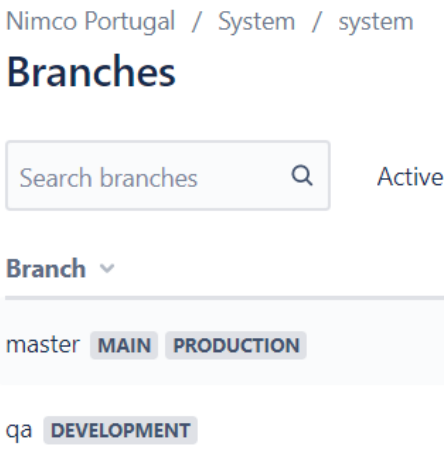

Figure 18 – Git structure.

Nimco has the VCS hosted on Bitbucket. Bitbucket is a Git-based source code repository hosting service owned by Atlassian (Wiki Bitbucket, 2022).

4.1.3 Integrated Development Environment

The IT department currently uses Jetbrains PHPStorm, an Integrated Development Environment (IDE) tailored for the PHP language and uses it to build the solution.

PhpStorm is a fast and brilliant IDE tailored for PHP developers. PhpStorm utterly understands the code, provides intelligent coding assistance and refactoring capabilities, highlights and fixes errors on the fly, and allows easy debugging and testing of code (PhpStorm, n.d.).

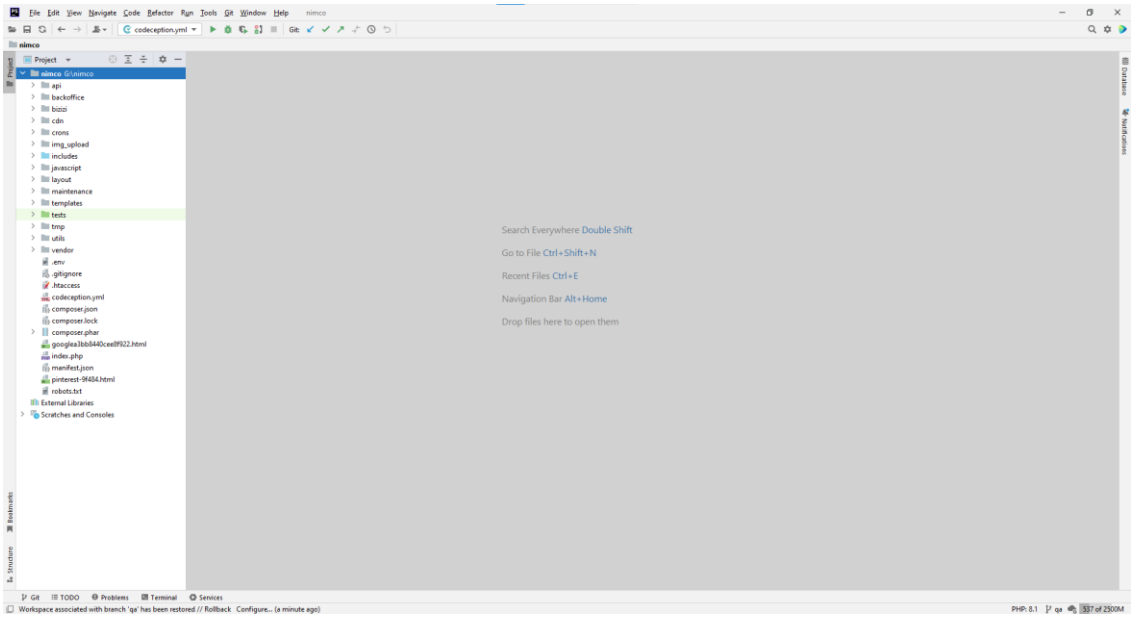

Figure 19 – PhpStorm Interface and project structure.

4.1.4 Testing framework

The testing framework used to apply the BDD is Codeception. It can manage unit, functional, and acceptance tests for PHP. Codeception was chosen as the testing tool because supports all three testing types, for writing unit, functional, and acceptance tests in a unified framework (Codeception, n.d.).

The next chapter presents the Codeception configuration and integration into the project.

	Unit Tests	Functional Tests	Acceptance Tests
Scope of the test	Single PHP class	PHP Framework (Routing, Database, etc.)	Page in browser (Chrome, Firefox, or PhpBrowser)
Testing computer needs access to project's PHP files	Yes	Yes	No
Webserver required	No	No	Yes
JavaScript	No	No	Yes
Additional software required	None	None	Drivers for Firefox/Chrome
Test execution speed	Very fast	Fast	Slow

Figure 20 – Codeception requirements for each individual test type. Source: (Codeception, n.d.).

4.1.5 Continuous Improvement / Continuous Delivery

The IT Department already had installed a Continuous Improvement (CI) / Continuous Delivery (CD) application. However, it is not in use because it is part of the project of modernizing the production system software. The CI/CD application is Jetbrains TeamCity. This application is where all development components will combine:

- Source Code.
- Version Control Branches (Git).
- Tests.
- Metrics.

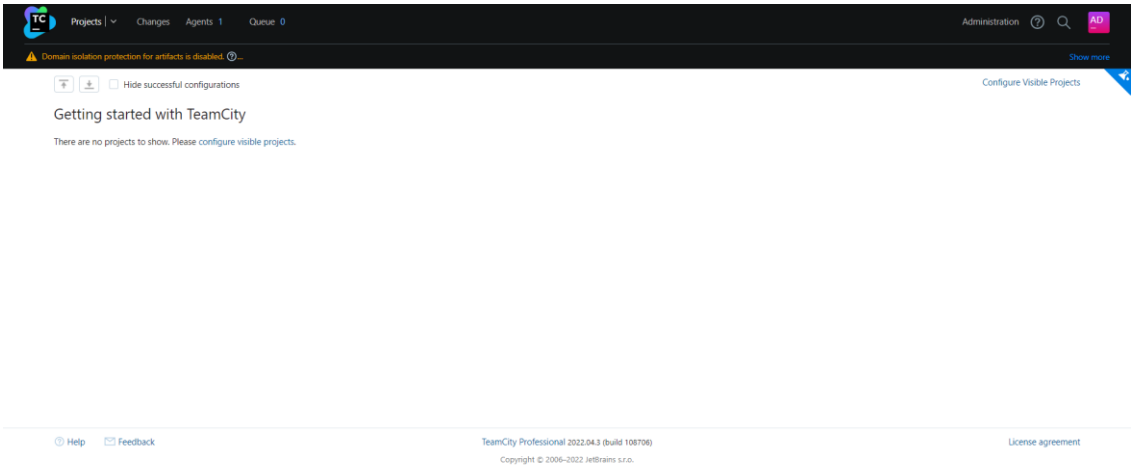

Figure 21 – Jetbrains TeamCity GUI.

4.2 Architecture

Software architecture is the structure that defines the flow of information through a software system. It is a set of decisions about how software is organized and operates to meet that software's goals (Wilson, 2015).

This chapter describes four design patterns used to develop the solution and the design principles the code should follow. Due to the project being a legacy application, where its scripts, classes, and functions combine the concerns of model, view, and controller into the same scope, this chapter references how the application's architecture should be, but it is not the primary objective.

4.2.1 Design Patterns

A design pattern is a specific solution to a commonly occurring problem in software development. These design patterns constitute a common language among software developers to discuss problems and describe solutions (Wilson, 2015).

4.2.1.1 Factory pattern

The Factory is a creational design pattern that provides an interface for creating objects in a superclass and allows subclasses to alter the type of objects created (Refactoring.Guru - Factory, n.d.).

There are three benefits to using factories (Wilson, 2015):

- Reusable code - any place that must create an object applies the same logic.
- Testable code - factories make creational logic easy to test. If this code were directly within some other class, it would not be possible to test separately.

- Easy to change - If the logic ever changes, the change only needs to occur in one place.

4.2.1.2 Repository pattern

A Repository is an object that allows the retrieval and persisting of data in a data store (Wilson, 2015).

A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting similarly to a set of domain objects in memory. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way closer to the persistence layer. Client objects declaratively build queries and sends them to the repositories for answers. Repositories also support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping (Fowler, 2015).

4.2.1.3 Adapter pattern

The adapter pattern allows for encapsulating the functionality of one object and making it conform to the functionality of another object (Wilson, 2015).

This pattern allows us to take one object and adapt it to fit the interface of another object, making them compatible in whatever context was trying to use them (Wilson, 2015).

4.2.1.4 Strategy pattern

The Strategy pattern allows an application's runtime to determine an algorithm's behavior. Strategies are usually a family of classes that share a familiar interface encapsulating individual behavior that can be interchangeable (Wilson, 2015).

4.2.2 SOLID Design Principles

SOLID Design Principles are a set of five basic design principles for object-oriented programming described by Robert C. Martin (Martin, 2000).

The SOLID Principles are:

1. Single Responsibility Principle (SRP).
2. Open/Closed Principle (OCP).
3. Liskov Substitution Principle (LSP).
4. Interface Segregation Principle (ISP).
5. Dependency Injection Principle (DIP).

4.2.2.1 Single Responsibility Principle

The Single Responsibility Principle states that objects should have one purpose, a principle that is often violated, especially by new programmers. A class becomes a jack of all trades and often performs several tasks, sometimes within several thousand lines of code, depending on the method (Wilson, 2015).

4.2.2.2 Open/Closed Principle

The Open/Closed Principle states that classes should be open to extension but closed to modification. This principle means that developers working on the system should not be allowed or encouraged to modify the source of existing classes but instead find ways to extend the existing classes to provide new functionality (Martin, 2000).

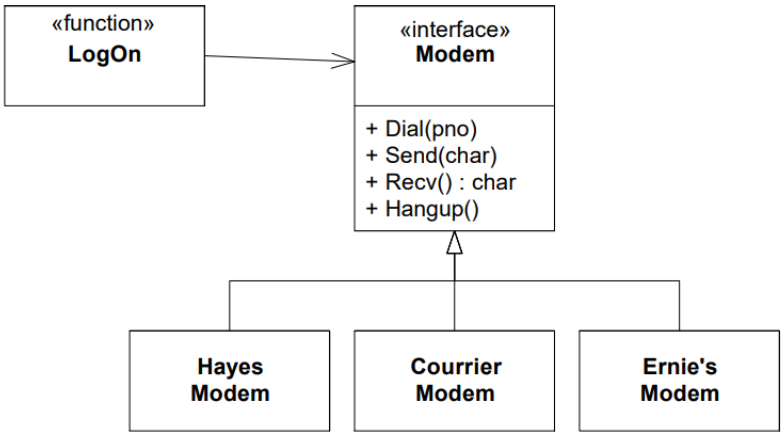

Figure 22 – Example of an Open/Closed Principle. Source: (Martin, 2000).

In Figure 22, as an example of the OCP, the LogOn function depends only upon the Modem interface. Additional modems will not cause the LogOn function to change. Thus, we have created a module that can be extended, with new modems, without requiring modification (Martin, 2000).

4.2.2.3 Liskov Substitution Principle

The Liskov Substitution Principle states that objects of the same interface should be interchangeable without affecting the behavior of the client program (Wilson, 2015).

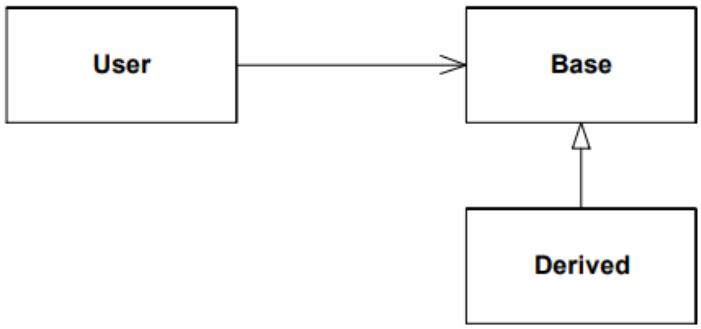

Figure 23 – Example of a Liskov Substitution Principle. Source: (Martin, 2000).

As illustrated in Figure 23, derived classes should be substitutable for their base classes. A user of a base class should continue to function correctly if a derivative of that base class is passed to it.

4.2.2.4 Interface Segregation Principle

The interface Segregation Principle dictates that client code should not be forced to depend on methods it does not use. The principle intends to fix the problem of extensive interfaces, which define many method signatures. It also relates to the SRP that interfaces should have only one responsibility (Wilson, 2015).

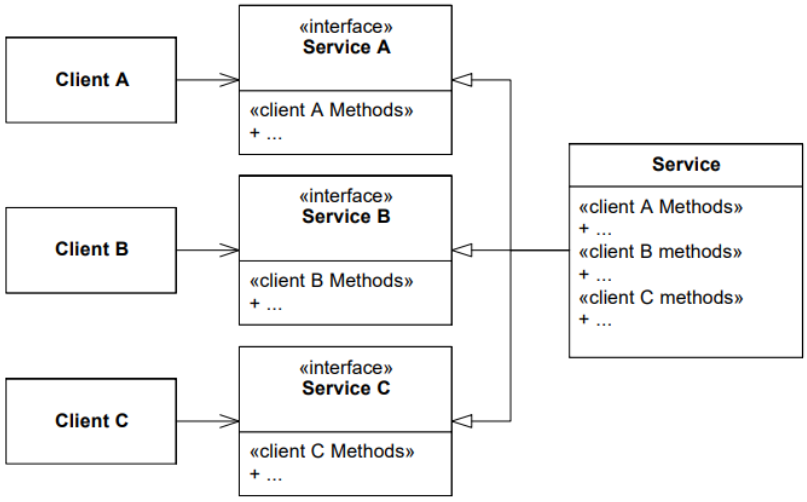

Figure 24 – Example of an Interface Segregation Principle. Source: (Martin, 2000).

For example, as shown in Figure 24, the methods needed by each client are placed in unique interfaces that are specific to that client. Those interfaces are multiply inherited by the Service class and implemented there. If the interface for client A needs to change, client B and client C will remain unaffected (Martin, 2000).

4.2.2.5 Dependency Injection Principle

The Dependency Inversion Principle states that (Wilson, 2015):

- High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Dependency Inversion is the strategy of depending upon interfaces or abstract functions and classes rather than upon concrete functions and classes (Martin, 2000).

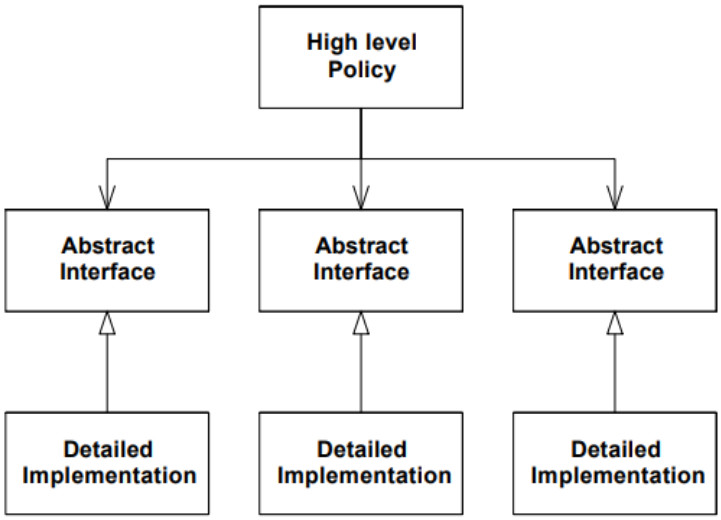

Figure 25 – Example of a Dependency Injection Principle. Source: (Martin, 2000).

In Figure 25, an object-oriented architecture shows a quite different dependency structure in which most dependencies point towards abstractions. Moreover, the modules that contain detailed implementation are no longer dependent upon them; instead, they depend upon abstractions. Thus, the dependency upon them has been inverted.

4.2.3 The Clean Architecture

"Uncle" Bob Martin named The Clean Architecture in 2012 as an architecture that adheres to extreme forms of separation of concerns (Martin, 2012).

Bob Martin describes these architectures as being (Martin, 2012), shown in figure 26:

- Independent of Frameworks.
- Testable.
- Independent of User Interface.

- Independent of Database.
- Independent of any external agency.

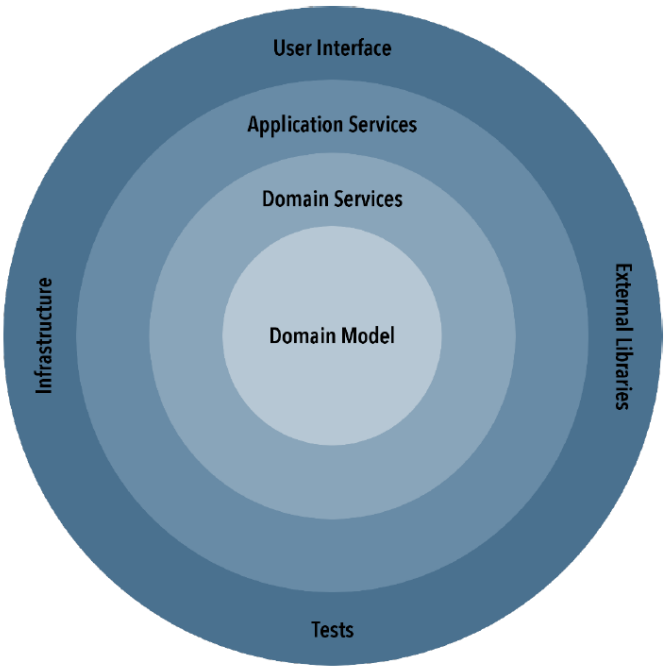

Figure 26 – The Clean Architecture. Source: (Wilson, 2015).

4.3 Case Study

The main objective of this dissertation is to meet the requirements of ISO 13485:2016. The IT department decided to implement software validation using BDD module by module because the legacy project has plenty of code, and it would not be possible to make it all testable for the duration of this dissertation. This implementation, however, will create a foundation for the IT department to build upon and continue this work.

The module selected for this dissertation as a case study is the integration of orders between the production system and the Navision ERP.

4.3.1 Production workflow

The Customer Service department is currently responsible for manually integrating orders between the production system and the Navision ERP.

The order integration process happens after Customer Service translates the order and uses the fields filled on the order forms to generate the information to be integrated into the Navision.

Figures 27 and 28 present the flowcharts of the Customer Service processes for MCO and orthopedic/infinity orders and the moment when the customer service does the integration.

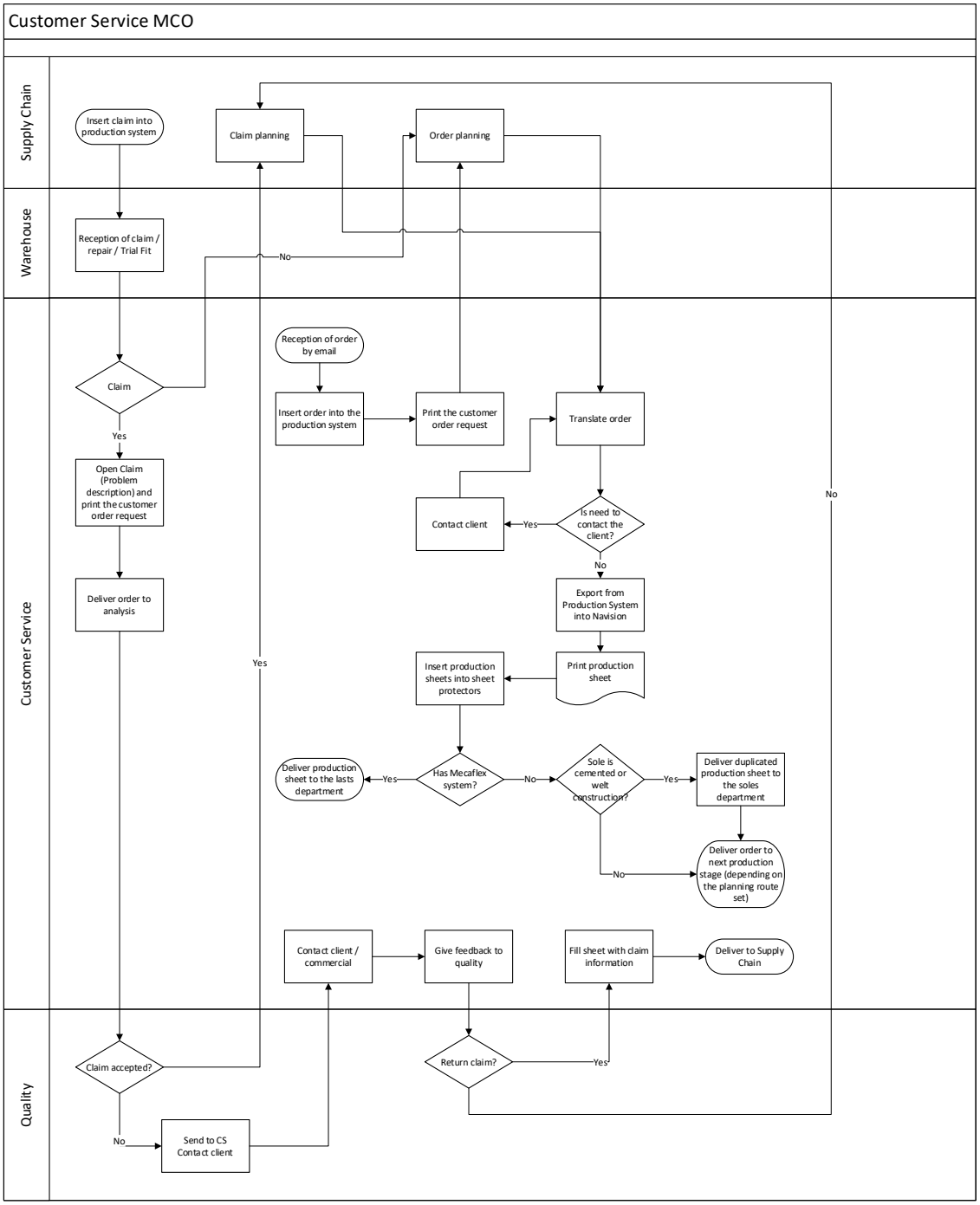

Figure 27 – Customer Service MCO orders flowchart.

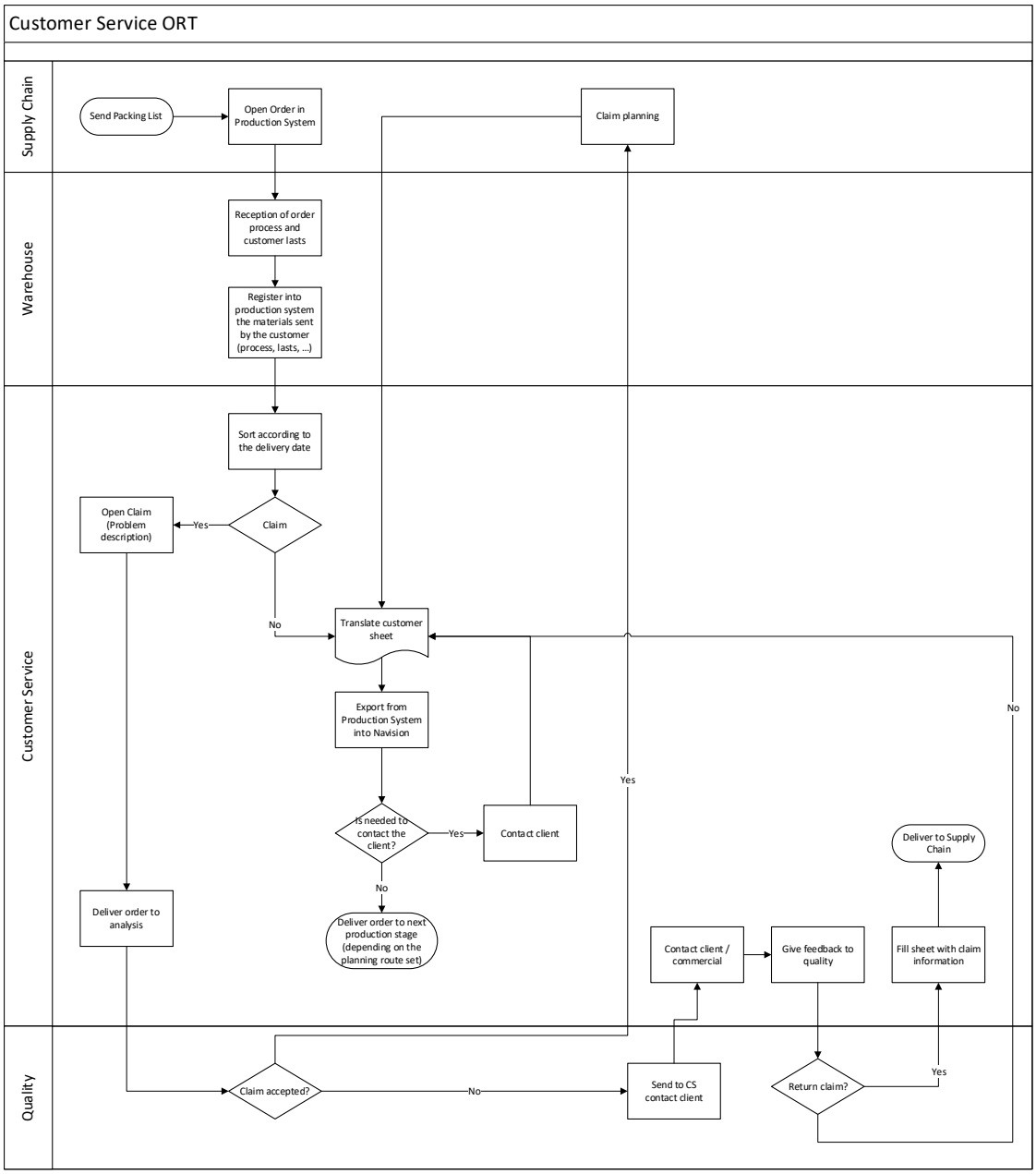

Figure 28 – Customer Service Orthopedic orders flowchart.

4.3.2 Financial data and integration file structure

This subsection details the financial items sent in the XML file and its connectivity between the order forms and Navision and where and the XML file structure for the integration.

4.3.2.1 Financial items

The financial items in Navision generally are composed of:

- Code.
- Description.
- Unit Cost

These items are invoiced depending on the order, for example, if:

- The customer changed the model in any regard.
- A particular field in the form is selected.
- Shipping costs.
- Observations and any other particular reason.

Table 24 and 25 lists some examples of Navision codes for MCO, orthopedic, and infinity orders.

Table 24 – Example of Navision Codes for MCO Orders.

Nav Code	**Description**
9116	9116 Nimco Tex Membrane
9121	9121 BOA closure
9123	9123 Special Soles
9141	9141 Neurological Inlay
9143	9143 Anti-Pronation Inlay
9150	9150 Special Counterfort (exc. 1001-1008)
9154	9154 Mecaflex System

Table 25 – Example of Navision Codes for orthopedic and infinity orders.

Nav Code	Description
8659	8659 Mecaflex System
8649	8649 Safety Toe Caps
8648	8648 Special Insole (exc. Reinforcement Insole)
8647	8647 Special Counterfort (exc. 1001and1008)
8630	8630 X-Sensible Stretch Technology
8667	8667 Special Leathers and Lining -E-Leather Swatch
8452	8452 Steel Reinforcement on the rear

For example, for 8659 code which regards the Mecaflex system for orthopedic orders, if selected in the order form in the Mecaflex field with values of left, right, or both, it should be charged and therefore integrate into Navision when this order gets exported.

Figure 29 illustrates the mecaflex field in orthopedic order form.

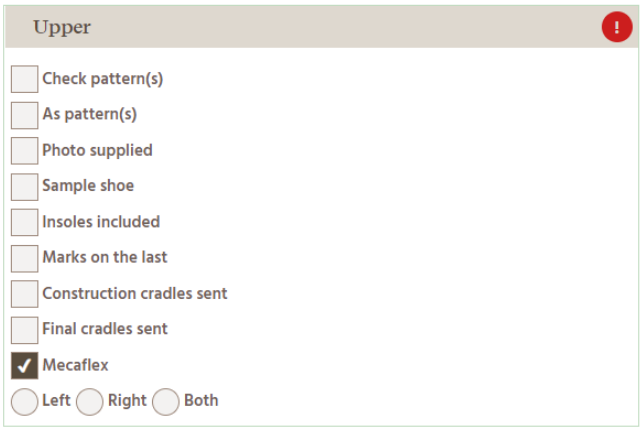

Figure 29 – Mecaflex field in orthopedic order form.

4.3.2.2 XML file structure

The XML file structure, illustrated in figure 30, used to integrate orders into Navision is composed of:

- DocumentType – Has always a value of “order”.
- SellToCustomerNo – This is the Customer Code of the order.
- ShipToCode – This is the Customer Address Code of the order.
- YourReference – This is the order number inserted in the production system.

- OrderDate – Order date in the production system.
- ShipmentDate – The shipment date defined on the planning stage in the production system.
- LocationCode – Has always the value of “NIMCO”.
- ExternalDocumentNo – Order reference in the production system.
- IntegrationStatus – Has two possible values, Pending or Approved. By default, the value is Approved but for some customers, Nimco has a system to notify the customer the estimated price for the order by email.
- SalesOrderLine – All financial codes are sent here. The file can have more than one SalesOrderLine.
    - ItemNo – Item code
    - Description – Always empty, so Navision uses the corresponding translated value for the item code.
    - Quantity – Value depends on the item code.
    - Unit Price – Always empty, so Navision uses the corresponding value in Navision.
    - PurchasingCode – Has always the value “DIRECT”.
    - LineDiscountPerc – Always empty, so Navision uses the corresponding value in Navision.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<SalesOrder>
    <DocumentType>Order</DocumentType>
    <SelltoCustomerNo>CUSTOMER NAV CODE</SelltoCustomerNo>
    <ShiptoCode>CUSTOMER ADDRESS FIL CODE</ShiptoCode>
    <YourReference>ORDER NUMBER</YourReference>
    <OrderDate>ORDER DATE</OrderDate>
    <ShipmentDate>SHIPMENT DATE</ShipmentDate>
    <LocationCode>NIMCO</LocationCode>
    <ExternalDocumentNo>ORDER REFERENCE</ExternalDocumentNo>
    <IntegrationStatus>STATUS</IntegrationStatus>
    <SalesOrderLine>
        <ItemNo>ITEM CODE</ItemNo>
        <Description/>
        <Quantity>QUANTITY</Quantity>
        <UnitPrice/>
        <PurchasingCode>DIRECT</PurchasingCode>
        <LineDiscountPerc/>
    </SalesOrderLine>
</SalesOrder>
```

Figure 30 – XML structure example file.

4.3.3 System's Architecture

Figure 31 represents the intercommunication between systems, where the FTP server is the connecting point. The production system generates XML files and stores them in the FTP Server, and then the middleware from the Navision server reads those files and integrates them into Navision.

Also, Navision generates XML files with the integration's success or failure status, and the production system reads them and integrates them into the production system database.

This integration is the same for Navision PT and Navision NL, and the production system sends the file to the correct folder in the FTP Server depending on the customer's country (Portuguese and Spanish go into Navision PT, and the rest goes into Navision NL)

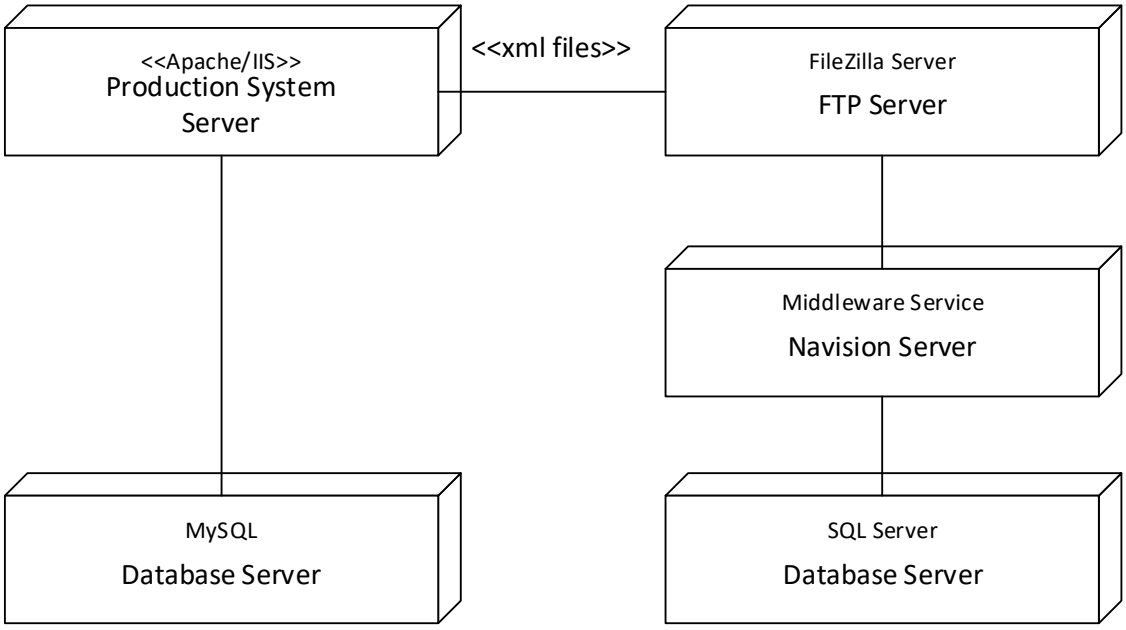

Figure 31 – Systems intercommunication diagram.

4.3.4 Requirements

The process starts with a requirement specification phase, where both functional and non-functional requirements describe the features to be implemented and to apply the principles of behavior test-driven development and clean architecture.

The Functional Requirements are as follows:

- The system must allow the user to browse the orders that are pending integration into Navision.

- The system must be able to show the status of the integration and the motive for the error if it exists.

- The system must integrate orders after the production stage of translation has been completed.

- The system must generate an XML file for Navision for each order by analyzing the order data. A list of financial codes must be integrated into the XML if specific values for the order are filled/selected.

- The system must be able to read XML files generated from Navision and integrate them into the production system.

The non-functional requirements are as follows:

- All components developed must use the SOLID principles in mind and best programming practices.
- All components should be evaluated using behavior test-driven development.

Table 26 – Use Cases of the case study.

Code	**Use Case**
UC001	Browse orders pending integration or in error
UC002	Analyze order data and generate the data to be in the XML file
UC003	Generate an XML file for each order
UC004	Import XML files with the integration Navision result

4.3.5 Code analysis

Due to the feature's implementation being completed previously by the IT department, our goal is to analyze the current codebase and implement tests into it, refactoring any code if needed.

Refactoring is the process of changing the structure of software without changing its behavior (Murphy-Hill et al., 2012).

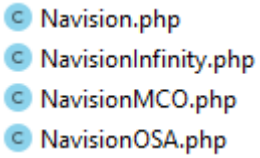

Figure 32 – Classes already in code base.

The structure of this feature is all concentrated between four classes: Navision, NavisionOSA, NavisionMCO, and NavisionInfinity, as presented in figure 32. These classes present that the code has:

- Functions with static attributes.
- There is no separation of concerns, so user interface, database access, domain logic, logging, and integrations are all concentrated inside a function, failing in most SOLID Design Principles.
- No implementation of any design patterns.

The IT department mentioned that this software was previously all procedural. There is an ongoing project inside the company to migrate the codebase to object-oriented, but at this moment, it is on hold. In the scope of this dissertation the focus will be on the class NavisionMCO.

Figure 33 – Code extract of exportOrderToNav existing function.

As an example, figure 33 is a code extract of the exportOrderToNav function where is found the following:

- The database object is instantiated inside the function, meaning that this class is dependent on the Dbo object.
- Querying the database and loop for each line.
- Logging into a file.

- Switch cases that depending on the country, order type, and infinity properties, call a function inside each of the other classes. This implementation looks like the strategy pattern, but without using it properly.

As described, the code heavily depends on many parts, meaning that the objective of simply integrating BDD into the project will be more difficult.

4.4 Approach strategy

This feature is already implemented so the approach strategy will be:

- Implement tests for existing code.
- Refactor the code to align with the SOLID design principles.
- Apply design patterns accordingly.

5 Solution Implementation

This chapter describes the solution implementation, starting with an overview of the TeamCity implementation. Afterward, an explanation of the Codeception integration into the production system and TeamCity.

Finally, a description of each function implemented or refactored, followed by testing.

5.1 TeamCity Configuration

As discussed earlier on chapter four, the IT department has already installed TeamCity, but without configuration. This configuration initially went through creating the project with the name System and is where the production and quality environments will be configured and synchronized with the version control system, Git.

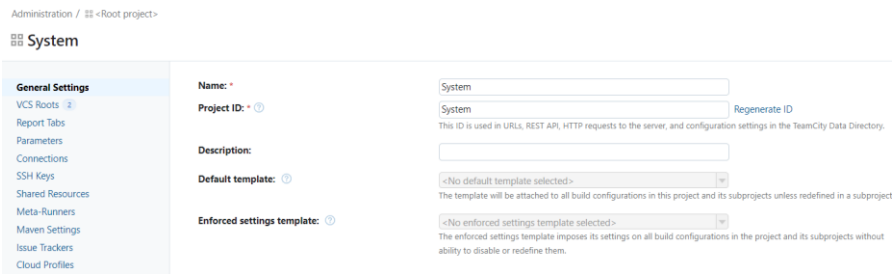

Figure 34 – TeamCity production system project.

Figure 34 presents the root project configuration with the project name and ID. After the project creation, the next step is to create communication with the VCS inside of TeamCity.

5.1.1 Git Branches configuration into TeamCity

The configuration of Git inside TeamCity is separated into two configurations, one for the QA branch and one for the Master branch.

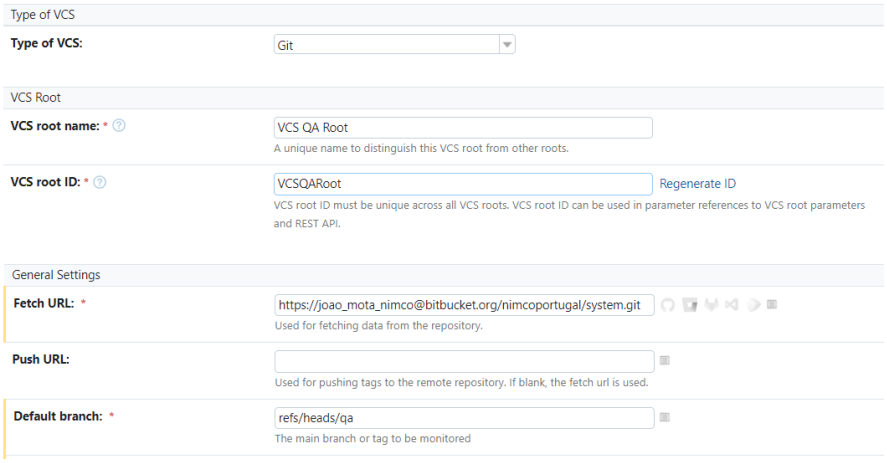

Figure 35 – Git QA branch configuration in TeamCity.

Figure 35 presents the configuration for the QA branch with:

- VCS root name - Named as "VCS QS Root."
- Fetch URL - Pointing towards the Nimco Bitbucket Repository of the production system.
- Default branch - configured to monitor the QA branch, so the configuration is "refs/heads/qa."

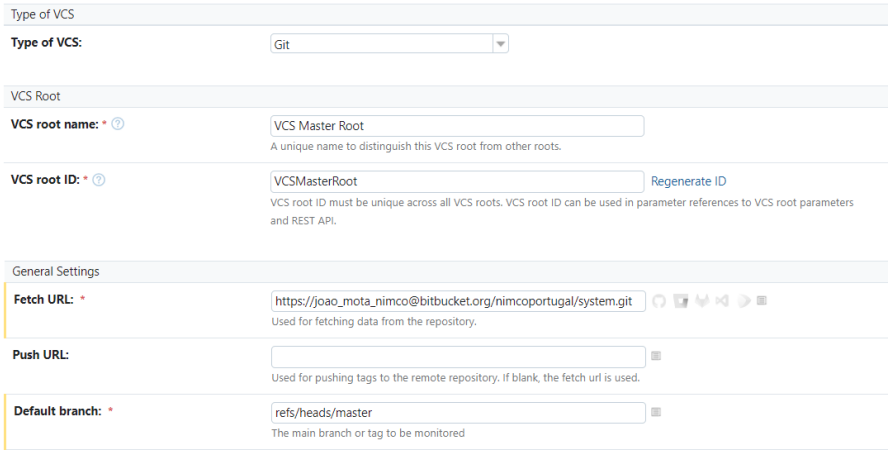

Figure 36 – Git Master branch configuration in TeamCity.

The configuration for the master branch is the same, with the name and default branch fields being different. Figure 36 represents the configuration for the Master branch with:

- VCS root name - Named as "VCS Master Root."
- Fetch URL - Pointing towards the Nimco Bitbucket Repository of the production system.
- Default branch - configured to monitor the Master branch, so the configuration is "refs/heads/master."

The next step is configuring the builds so that when the programmer commits to the respective branch, TeamCity triggers the build and runs the later configured steps.

5.1.2 Build Configuration

The build configuration starts with:

- The definition of the trigger to run the build.
- The attachment of the previously configured VCS Roots to the respective configuration.

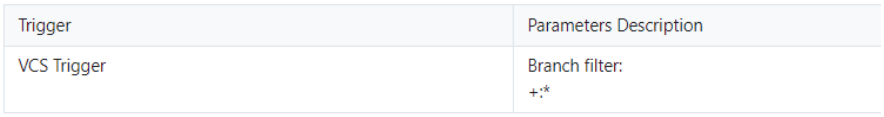

Figure 37 – Git branches trigger build configuration in TeamCity.

Figure 37 represents the configuration of the trigger. The filter presented in the parameter description means that when any change happens to the branch, it triggers the run of the build.

Each branch of the VCS has its build configuration because the code base can be different in both QA and Master.

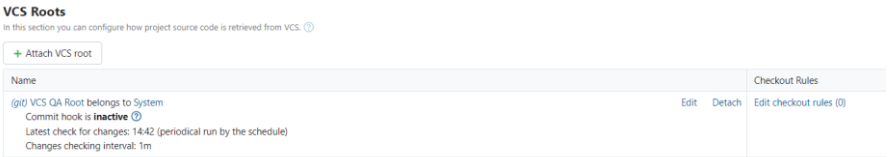

Figure 38 – QA build configuration in TeamCity.

Figure 38 represents the association of the previously created VCS QA Root to the QA Build configuration.

5.1.2.1 Master Build Configuration

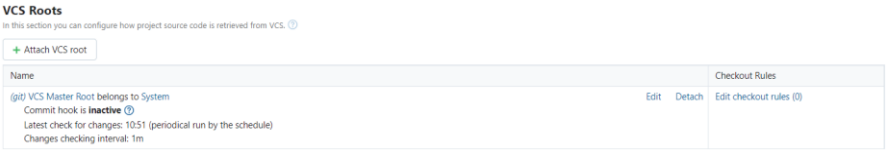

Figure 39 – Master build configuration in TeamCity.

Figure 39 represents the association of the previously created VCS Master Root with the Master Build configuration.

The next step is to add the Codeception into the production system project and respective configurations. Then later, return to TeamCity to add steps to the build configurations.

5.2 Codeception Configuration

Since the production system project already has composer configured, adding Codeception to the project made it easier. To achieve this, a JSON file was added with the following configurations:

- "phpunit/phpunit" package dependency for Codeception.
- "codeception/codeception" is the base package for the testing tool.
- "codeception/module-phpbrowser" and "codeception/module-asserts" are optional modules for the testing tool used to test the code base.
- "neronmoon/teamcity-codeception" is the package used for the testing logs integration into TeamCity.

```
"require-dev": {
    "phpunit/phpunit": "9.3.7", 9.3.7
    "codeception/codeception": "^4.1", 4.2.2
    "codeception/module-phpbrowser": "^1.0.0", 1.0.3
    "codeception/module-asserts": "^1.0.0", 1.3.1
    "neronmoon/teamcity-codeception": "^0.1.0" v0.1
},
```

Figure 40 – Codeception packages in composer.json file.

Figure 40 presents the configurations added to the existing composer.json file of the production system project.

The next step is creating a configuration file for Codeception to add the specific and required parameters.

```yaml
paths:
    tests: tests
    output: tests/_output
    data: tests/_data
    support: tests/_support
    envs: tests/_envs
actor_suffix: Tester
extensions:
    enabled:
        - Codeception\Extension\RunFailed
        - Codeception\Extension\TeamCity
coverage:
    enabled: true
    include:
        - ./*
    exclude:
        - tests/*
        - vendor/*
        - utils/*
```

Figure 41 – Codeception.yaml configuration file.

Figure 41 represents the codeception.yaml configuration file created at the root of the project, with the following relevant configurations:

- Configuration of paths.
- Enabled extensions.
- Enabled code coverage including all the code base with the exception of the tests, vendor, and utils folders.

With all the previous configurations and packages now installed, codeception can run in the project. Figure 42 presents a manual run of codeception in the terminal.

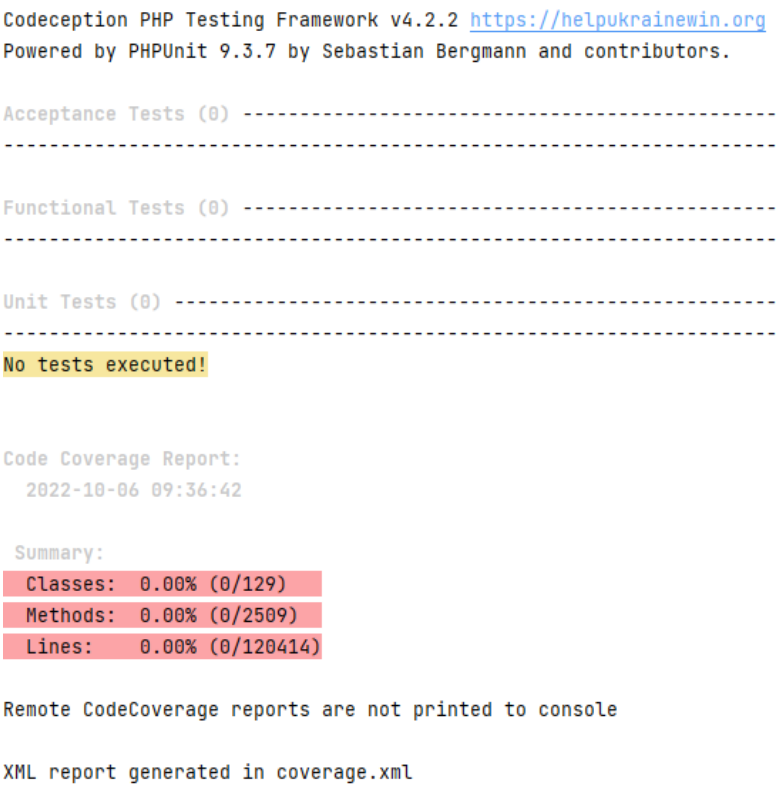

Figure 42 – Codeception run from the terminal.

The goal is to automate this process and will be the following configuration on TeamCity.

5.2.1 QA & Master Build Step

In the build steps section in TeamCity, for each of the builds previously created a step (as illustrated in figure 43) was added with the following configurations:

- Runner type of "command line."
- Step name as "codeception."
- Run a custom script with the command:
    - "C:\php81\php.exe vendor/codeception/codeception/codecept --steps --xml --no-interaction --debug --no-ansi --no-colors run --config=codeception.yml --coverage --coverage-html"

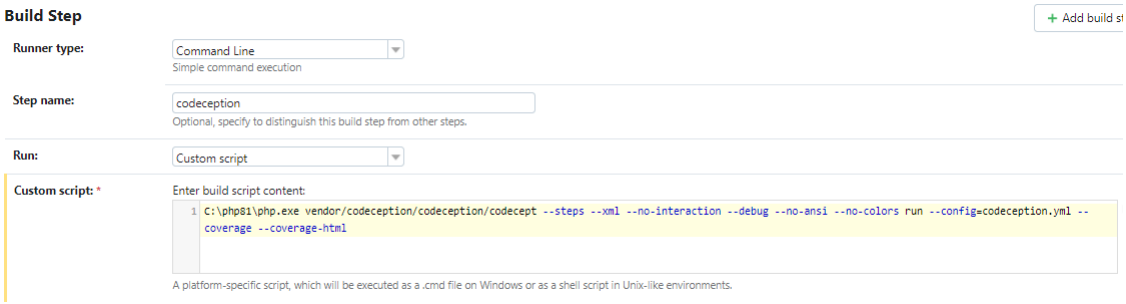

Figure 43 – Build step for codeception in TeamCity for QA & Master branches.

The shown command above uses the PHP executable to run codeception inside the project using our YAML configuration file with no interaction. It generates an XML file with test results and an HTML file with code coverage metrics.

5.2.1.1 Code Coverage into TeamCity

With the build steps configured, for each build, an artifact is generated inside the folder "tests_output\coverage", with the HTML code coverage page. To make this available through the TeamCity GUI was needed to:

- enable the publish artifacts configuration for each build configuration
- add the artifact path of the coverage folder
- create a tab in the UI to access the code coverage.

Figures 44 and 45 represent these configurations.

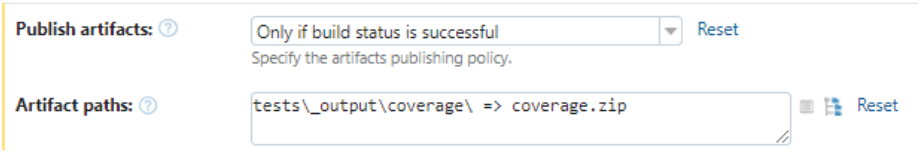

Figure 44 – Artifact configuration for codeception in TeamCity.

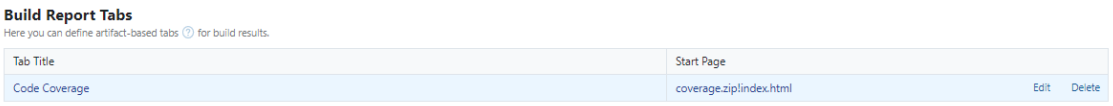

Figure 45 – Custom tab for codeception code coverage in TeamCity.

Figure 46 illustrates the code coverage HTML output after a successful build run with the following values:

- Percentage of lines tested per folder.
- Percentage of functions and methods tested per folder.

- Percentage of classes and traits tested per folder.

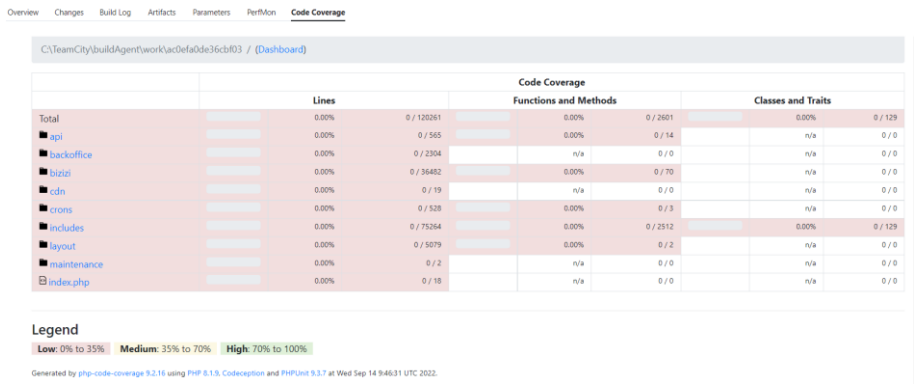

Figure 46 – Codeception code coverage UI in TeamCity.

The user can access each folder and check for each project file and the respective metrics. The UI also has a dashboard with other metrics like code complexity and project risks, represented in figure 47. This final configuration concludes TeamCity and codeception integration into the production system project. The following subsection describes the case study implementation.

Figure 47 – Codeception code coverage Dashboard UI in TeamCity.

5.3 Case Study Implementation

As mentioned in subsection 4.3.5, the focus will be on the NavisionMCO Class. The first step was to extract the functions or lines of code regarding persistence from the domain logic code to another class. For this, a class named NavisionMCODB was created to have all those functions, as represented in figure 48.

```php
<?php

class NavisionMCODB
{
    public const EVEREST_M_PRODUCT_ID = 3262;
    public const HIMALAYA_M_PRODUCT_ID = 3264;
    public const WHISTLER_PRODUCT_ID = 3359;
    public const SAFETY_BOOT_PRODUCT_ID = 3308;
    public const HIMALAYA_W_PRODUCT_ID = 3265;
    public const BERG_PRODUCT_ID = 3324;
    public const ROBIN_PRODUCT_ID = 3376;
    public const ELISABETH_PRODUCT_ID = 3298;
    public const EVEREST_W_PRODUCT_ID = 3263;

    public static function navHasStretchLeather($comps, $materials){...}
    public static function devolveCodigoNavisionProduto($id_prod, $id_enc, &$stretch, $temp = ''){...}
    public static function getIdRoteiro($orderId): mixed{...}
    public static function getOrderProduct(array $order): array{...}
    public static function getProductCodes($productId, $codigo, $stretch, $pe_esq, $pe_drt): array{...}
    public static function getConfiguracoesEncomendaMco($ID, $temp = ''){...}
    public static function getConfiguradasTemp(array $order): array{...}
    public static function navTemNeurological($ID, $temp = ''){...}
    public static function navTemSoladoSolto($ID, $temp = ''){...}
    public static function navTemAEncomenda($ID, $temp = ''){...}
    public static function navTemGEncomenda($ID, $temp = ''){...}
    public static function navTemWPEncomenda($ID, $temp = ''){...}
    public static function navTemPronation($ID, $temp = ''){...}
    public static function navTemS15Encomenda($ID, $temp = ''){...}
    public static function navTemForroEspecial($ID, $temp = ''){...}
    public static function checkIfHasSpecialLeathers(int $orderId, string $temp = ''): bool{...}
    public static function hasEmbroidery(int $orderId, string $temp = ''){...}
    public static function getOrderPriority($order_id) {...}
}
```

Figure 48 – NavisionMCODB class.

With persistence out of the main class, the next step was to create unit tests for each function regarding domain logic. Annex A shows the complete code for the Unit Cases created for the functions.

Figure 49 shows the current status of the NavisionMCO class, with the first two functions being the domain services, which should be in their separate classes and the remaining functions being the domain logic for MCO orders integration.

```php
<?php

/** Class NavisionMCO ...*/
class NavisionMCO
{
    public function getNavCodes(array $order): array{...}
    public static function exportOrderMcoToNav(int $navType, array $order, int $fileCounter, bool $integration): string{...}

    private static function navCode9161(array $navItems, int $hasEmbroideryOrLaser): array{...}
    private static function navCode9160(array $navItems, bool $hasSpecialLeathers): array{...}
    private static function navCode9141(array $navItems, int $hasNeurological): array{...}
    private static function navCode9143(array $navItems, int $hasPronation): array{...}
    private static function navCode9801(array $navItems, int $soladoSolto): array{...}
    private static function navCode9150(array $navItems, array $extras, float $shoeHeight): array{...}
    private static function navCode9106(array $navItems, array $extras): array{...}
    private static function navCode9115(array $navItems, array $extras): array{...}
    private static function navCode9102(array $navItems, array $extras, int $isCrianca): array{...}
    private static function navCodeLastChanges(array $navItems, array $extras): array{...}
    private static function navCode9107(array $navItems, int $id_prioridade): array{...}
    private static function navCode9116(array $navItems, array $extras): array{...}
    private static function navCode9121(array $navItems, array $extras): array{...}
    private static function navCode9154(array $navItems, array $extras): array{...}
    private static function navCode9110(array $navItems, array $extras): array{...}
    private static function navCode9101(array $navItems, array $extras): array{...}
    private static function navCode9111(array $navItems, array $extras): array{...}
    private static function navCode9113(array $navItems, array $extras): array{...}
    private static function navCode9123(array $navItems, array $extras): array{...}
    private static function getShippingCodes(array $navItems, int $clientId, int $countryId, int $zoneId): array{...}
    private static function getAdministrativeCostsCode(array $navItems, int $routeId, int $customerId, int $internalUserId): array{...}
    private static function getRockerCodes(array $navItems, array $extras, array $defaultSoleRocker): array{...}
}
```

Figure 49 – NavisionMCO class.

6 Solution Validation

This chapter starts with a design subsection on how the analysis will be conducted and describes the metrics used and the evaluation criteria. The following subsection, experiments, shows an insight into the experiments carried out. The last subsection, the summary, provides the conclusions for each experiment.

6.1 Design

Solution Validation is performed based on the Goal, Question, Metrics (GQM) approach.

GQM is a method used to measure a specific goal based on a set of metrics, performed on three levels (Calabrese et al., 2018):

- Conceptual Level (Goal) – The first step is to define a goal for each quality attribute studied, based on the proof of concept developed.

- Operation Level (Question) – To understand how the defined goals can be met, a set of questions are introduced.

- Quantitative Level (Metric) – Last level consists on defining the metrics (objective or subjective) to answer the questions and conclude how the goals are met.

Table 27 – Goals and Questions defined for each Quality Attribute.

Quality Attribute	**Goals**	**Questions**
Code Coverage	The solution must be validated with tests.	Is the solution's bug free by using tests?
Testability	The solution needs to be easily testable.	How complex is the solution in terms of testing?

After defining the Goals and Questions in Table 27, the next step is establishing metrics and specific evaluation criteria.

6.1.1 Code Coverage

Code coverage measures the degree to which a test suite executes into a software system. Although coverage is well established in software engineering research, deployment in the industry is often inhibited by the perceived usefulness and the computational costs of implementation.

Therefore, Codeception's Code Coverage is a good measure for this case study and will be conducted for each file created or refactored. Table 28 shows a relation between the metric, rating, and evaluation for Code Coverage.

Table 28 – Relation between metric, rating and evaluation for Code Coverage.

Metric	**Evaluation**
100%	Acceptable
> 80%	
> 50%	
> 40%	Not Acceptable
> 20%	
> 0%	

6.1.2 Testability

Cyclomatic Complexity is a metric that measures the number of independent paths through a program, providing a quantitative measure for the logical Complexity, usually associated with test planning and test-case design (Pressman, 2014). The Change Risk Anti-Patterns (C.R.A.P.) score is designed to analyze and predict the amount of effort, pain, and time required to maintain an existing body of code (Savoia, 2007).

The metrics are then related to the evaluation to validate the case study, as shown in Table 29.

Table 29 – Relation between metric and evaluation for Testability.

Metric	Description	Evaluation
1 - 10	Minimal Complexity	Acceptable
10 - 20	Moderate Complexity	
21 - 30	High Complexity	Not Acceptable
> 30	Very High Complexity	

The experiment is conducted by analyzing the source code for complexity using Codeception's C.R.A.P. Index, and the evaluation is then applied to the file with the higher cyclomatic complexity.

6.2 Experiments

This section provides insight into the results of the experiments performed on each quality attribute.

6.2.1 Code Coverage

Maintainability experiments were performed using Codeception code coverage.

Table 30 provides a summary of the analysis and figure 50 and 51 shows the complete results.

Table 30 – Summary of Maintainability analysis results.

File	Maintainability Rating
NavisionMCO.php	Acceptable
NavisionMCODB.php	Not Acceptable

	Code Coverage									
	Lines			Functions and Methods				Classes and Traits		
Total		50.18%	275 / 548		91.67%	22 / 24	CRAP		0.00%	0 / 1
NavisionMCO		50.18%	275 / 548		91.67%	22 / 24	11882.82		0.00%	0 / 1

Figure 50 – NavisionMCO code coverage results.

	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		0.00%	0 / 280		0.00%	0 / 21	CRAP	0.00%	0 / 1
NavisionMCODB		0.00%	0 / 280		0.00%	0 / 21	8190	0.00%	0 / 1

Figure 51 – NavisionMCODB code coverage results.

From the results of the analysis, it is possible to notice technical debt on some files that could easily be fixed.

6.2.2 Testability

Testability experiments were performed using Codeception. This experiment was conducted on the source code created or refactored in this case study but focused on the file with the higher cyclomatic complexity, which is explained its meaning in Section 6.1. Table 31 shows a summary of the testability results and figures 50 and 51, the results.

Table 31 – Summary of Testability results.

File	C.R.A.P. Score
NavisionMCO.php	11882.82 (original value was 162006)
NavisionMCODB.php	8190

The results were unacceptable overall as the max cyclomatic complexity on each case study file is usually above 30, meaning very high complexity. A drastic reduction in the NavisionMCO C.R.A.P. Score of about 93% was noticed.

6.3 Summary

The time frame of this dissertation, and the fact that it is a case study, justify that overall results of the experiences were merely acceptable.

In terms of maintainability, the reduced number of lines of code in the source code of each feature is an advantage. Moreover, a good separation of concerns also acts as an obligation of not coupling the code base, consequently increasing overall maintainability. Experienced developers should even be able to increase the maintainability of each feature present in the production system, focusing on good separation of concerns and good code practices with the aid of tools such as Codeception and continuous delivery and integration.

Finally, on the testability side, although there are fewer lines of code, cyclomatic complexity increases if the code is not properly implemented, and consequently, testability degrades. Regarding this case study, mainly due to my experience in web development and the limited duration of the analysis period, cyclomatic complexity is relatively high in some classes, which should be a topic that requires special care in a real development scenario.

7 Conclusion

This chapter presents the main conclusions concerning this work: goals achieved, difficulties, limitations, and proposals for future work.

7.1 Summary

This dissertation provides an insight into what medical devices are and how the Norm categorizes the products of Nimco as Medical Devices. It also introduces ISO 13485:2016 and the current implementation status of the company, the software used and the IT architecture, and a risk assessment of the IT structure and systems.

To solve the non-conformity of software validation, the author analyzed the advantages and disadvantages of several agile test methodologies, in this case, TDD, BDD, and ATDD. The decision by the IT department was to use BDD. Also, it was made a Value Analysis and a Value proposition for this dissertation and how it is relevant to the company.

Regarding the software validation design and implementation, the company decided to implement it on the already implemented feature of integrating orders between the production system and the ERP Navision.

The author then evaluated the solution to determine if it met the ISO compliance requirements.

7.2 Goals Achieved

The main objective was to evaluate and implement solutions to meet the requirements of the ISO 13485:2016 Medical devices - Quality management systems - Requirements for regulatory purposes.

To achieve this objective the author of this dissertation studied the ISO and compiled a report of the findings for the whole organization focused on the IT requirements. After that, the author created a risk-based approach for the software and physical architecture of the company, identifying non-compliance with the norm.

After all the analysis and design being completed, the author developed a solution and supported it with documentation to solve one of non-conformities detected by the external audit regarding software validation, based on the agile test-driven development methodology selected, in this case BDD.

Finally, the author evaluated if the solution met the requirements for ISO compliance and according to the APCER audit it does but must be extended and turn it into a common practice.

For the author, the development of this project proved to be very important for his personal and professional growth. He hopes that with the acquired knowledge here, it will be possible to transform more features and make the production system more robust and bug-free. The author intends to continue researching this subject and continually seek new ways to face the impasses and solve them in a good way.

7.3 Limitations and future work

Regarding the limitations encountered in the project's development, we highlight the further obstacle of implementing a test-driven environment on an outdated application and obtaining more results. At the time of writing this document, it had not yet been possible to implement BDD into the production system fully.

Future work may include continuing this iterative development of integrating BDD into the production system project.

References

(ISO 13485 Certification Services, Training Courses & Resources, 2021)	ISO 13485 Certification Services, Training Courses & Resources. (2021). LRQA. https://www.lrqa.com/en/iso-13485/
(Market Research Report, 2021)	Medical devices market size, share, trends: Analysis, 2028. Medical Devices Market Size, Share, Trends | Analysis, 2028. (n.d.). Retrieved February 13, 2022, from https://www.fortunebusinessinsights.com/industry-reports/medical-devices-market-100085
(EU 2017/745, 2017)	REGULATION (EU) 2017/745 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL. (2017). Retrieved December 26, 2021, from https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32017R0745&from=PT
(RAPS, 2021)	New ISO 13485: Device Companies Have Three Years to Transition. (2021). RAPS. https://www.raps.org/regulatory-focus%E2%84%A2/news-articles/2016/3/new-iso-13485-device-companies-have-three-years-to-transition
(Peffers et al., 2007)	Ken Peffers, Tuure Tuunanen, Marcus A. Rothenberger & Samir Chatterjee (2007) A Design Science Research Methodology for Information Systems Research, Journal of Management Information Systems, 24:3, 45-77, DOI: 10.2753/MIS0742-1222240302
(Peffers et al., 2006)	Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. (2020). Design Science Research Process: A Model for Producing and Presenting Information Systems Research. arXiv preprint arXiv:2006.02763.
(Diabetes Atlas, 2022)	Tenth edition. IDF Diabetes Atlas. (2022). Retrieved February 23, 2022, from https://diabetesatlas.org/
(ISO Members, 2021)	Members. ISO. (2021, January 22). Retrieved December 26, 2021, from https://www.iso.org/members.html
(ISO Wiki, 2021)	Wikimedia Foundation. (2021, December 19). International Organization for Standardization. Wikipedia. Retrieved December 26, 2021, from https://en.wikipedia.org/wiki/International_Organization_for_Standardization
(IPQ, 2021)	IPQ. ISO. (2021, December 7). Retrieved December 26, 2021, from https://www.iso.org/member/2054.html
(ISO 13485, 2016)	ISO 13485. (2016). ISO 13485 - Quality management for medical devices. Vernier, Geneva; ISO. Retrieved December 26, 2021, from https://www.iso.org/obp/ui/#iso:std:iso:13485:ed-3:v1:en
(ISO Survey, 2021)	ISO Survey of certifications to management system standards. Committee 09. ISO Survey of certifications to management system standards - Full results. (2021, September 6). Retrieved December 27, 2021, from https://isotc.iso.org/livelink/livelink?func=ll&objId=18808772&objAction=browse&viewType=1
(ISO/TR 80002-2:2017 2017)	ISO/TR 80002-2:2017. ISO. (2017, June 13). Retrieved September 13, 2022, from https://www.iso.org/standard/60044.html
(ISO 14971:2019, 2019)	ISO 14971:2019. ISO. (2019, December 10). Retrieved January 2, 2022, from https://www.iso.org/standard/72704.html
(Khanam et al., 2017)	Khanam, Z., & Ahsan, M. N. (2017). Evaluating the effectiveness of test-driven development: advantages and pitfalls. International Journal of Applied Engineering Research, 12(18), 7705-7716.

(Al-Saqqa et al., 2020)	Al-Saqqa, S., Sawalha, S., & AbdelNabi, H. (2020). Agile Software Development: Methodologies and Trends. International Journal of Interactive Mobile Technologies, 14(11).
(Fowler, 2006)	Fowler, M., & Foemmel, M. (2006). Continuous integration.
(Moe, 2019)	Moe, M. M. (2019). Comparative study of test-driven development (TDD), behavior-driven development (BDD) and acceptance test–driven development (ATDD). International Journal of Trend in Scientific Research and Development, 231-234.
(Chen, 2022)	Chen, G. (2022, January 19). A primer to Acceptance Test Driven Development (ATDD). Product Manager HQ. Retrieved September 12, 2022, from https://productmanagerhq.com/a-primer-to-acceptance-test-driven-development-atdd/
(Fowler and Foemmel, 2006)	Fowler, M., & Foemmel, M. (2006). Continuous integration.
(Saleh et al., 2019)	Saleh, S. M., Huq, S. M., & Rahman, M. A. (2019, February). Comparative study within Scrum, Kanban, XP focused on their practices. In 2019 International Conference on Electrical, Computer and Communication Engineering (ECCE) (pp. 1-6). IEEE.
(Bell, 2001)	Bell, J. T. (2001). Extreme programming.
(Lindstrom & Jeffries, 2003)	Lindstrom, L., & Jeffries, R. (2003). Extreme programming and agile software development methodologies. In IS management handbook (pp. 531-550). Auerbach Publications.
(Anwer et al., 2017)	Anwer, F., Aftab, S., Waheed, U., & Muhammad, S. S. (2017)." Agile Software Development Models TDD, FDD, DSDM, and Crystal Methods: A Survey". International journal of multidisciplinary sciences and engineering", 8(2), 1-10.
(Rahman & Gao, 2015)	Rahman, M., & Gao, J. (2015, March). A reusable automated acceptance testing architecture for microservices in behavior-driven development. In 2015 IEEE Symposium on service-oriented system engineering (pp. 321-325). IEEE.
(Rich and Holweg, 2000)	Rich, N. and Holweg, M. (2000) Value Analysis. Cardiff, United Kingdom: Lean Enterprise. Research Centre, pp. 1–32. Available at: https://www.urenio.org/tools/en/value_analysis.pdf (Accessed: January 10, 2022).
(Koen et al., 2002)	KOEN, P. A. et al. Fuzzy Front End: Effective Methods, Tools and Techniques. In: BELLIVEAU, P.; GRIFFIN, A.; SOMERMEYER, S. The PDMA Toolbook for new product development. New York: John Wiley & Sons, 2002.
(OECD, 2021)	OECD (2021), Health at a Glance 2021: OECD Indicators, OECD Publishing, Paris, https://doi.org/10.1787/ae3016b9-en. Retrieved February 26, 2022, from https://data.oecd.org/healthres/health-spending.htm
(Nicola et al., 2012)	Nicola, S., Ferreira, E. and Ferreira, J. (2012). A Novel Framework For Modeling Value For The Customer, An Essay On Negotiation. International Journal of Information Technology & Decision Making, 11(03), pp.661-703.
(Zeithaml, 1988)	Zeithaml, V. (1988). Consumer Perceptions of Price, Quality, and Value: A Means-End Model and Synthesis of Evidence. Journal of Marketing, 52(3), pp.2-22.
(Woodroof, 1997)	Woodruff, R. (1997). Customer value: The next source for competitive advantage. Journal of the Academy of Marketing Science, 25(2), pp.139-153.
(Osterwalder & Pigneur, 2003)	A.Osterwalder & Y.Pigneur, (2003). Modeling Value Propositions in EBusiness. Proceedings of the 5th International Conference on Electronic Commerce, ICEC 2003, Pittsburgh, Pennsylvania, USA.

(Saaty, 1990)	Saaty, T. L. (1990). How to make a decision: the analytic hierarchy process. European journal of operational research, 48(1), 9-26.
(Nicola, 2019)	Nicola, S. (2019). Método de Análise Hierárquica. Instituto Superior de Engenharia do Porto, 2019.
(BrowserStack, 2022)	TDD vs BDD VS ATDD : Key differences. BrowserStack. (2022, February 18). Retrieved February 27, 2022, from https://www.browserstack.com/guide/tdd-vs-bdd-vs-atdd
(Wilson, 2015)	Wilson, K. (2015). The Clean Architecture in Php. Leanpub.
(Domareski, 2021)	Domareski, H. S. (2021, May 16). Monolithic & Microservices Architecture. Medium. Retrieved September 7, 2022, from https://henriquesd.medium.com/monolithic-microservices-architecture-239e8799d3e1
(PHP Supported versions, n.d.)	Supported versions. php. (n.d.). Retrieved September 7, 2022, from https://www.php.net/supported-versions.php
(Wiki PHP, 2022)	Wikimedia Foundation. (2022, September 4). PHP. Wikipedia. Retrieved September 7, 2022, from https://en.wikipedia.org/wiki/PHP
(Wiki Bootstrap, 2022)	Wikipedia. (2022, August 7). Bootstrap (front-end framework). Wikipedia. Retrieved September 7, 2022, from https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)
(JavaScript, n.d.)	JavaScript. MDN. (n.d.). Retrieved September 7, 2022, from https://developer.mozilla.org/en-US/docs/Web/JavaScript
(jQuery, n.d.)	js.foundation, J. S. F.-. (n.d.). jQuery. JQuery. Retrieved September 7, 2022, from https://jquery.com/
(CSS: Cascading style sheets, n.d.)	CSS: Cascading style sheets. MDN. (n.d.). Retrieved September 7, 2022, from https://developer.mozilla.org/en-US/docs/Web/CSS
(Wiki MySQL, 2022)	Wikimedia Foundation. (2022, August 25). MySQL. Wikipedia. Retrieved September 7, 2022, from https://en.wikipedia.org/wiki/MySQL
(Composer, n.d.)	Introduction#. Composer. (n.d.). Retrieved September 7, 2022, from https://getcomposer.org/doc/00-intro.md
(Wiki Bitbucket, 2022)	Wikimedia Foundation. (2022, June 15). Bitbucket. Wikipedia. Retrieved September 15, 2022, from https://en.wikipedia.org/wiki/Bitbucket
(Git, n.d.)	Git. (n.d.). Retrieved September 7, 2022, from https://git-scm.com/
(PhpStorm, n.d.)	PhpStorm: The lightning-smart php ide. JetBrains. (n.d.). Retrieved September 7, 2022, from https://www.jetbrains.com/lp/phpstorm-japan/
(Codeception, n.d.)	Codeception - Introduction. Introduction - Codeception Docs. (n.d.). Retrieved September 7, 2022, from https://codeception.com/docs/Introduction
(Refactoring.Guru - Factory, n.d.)	Guru, R. (n.d.). Factory method. Refactoring.Guru. Retrieved September 8, 2022, from https://refactoring.guru/design-patterns/factory-method
(Fowler, 2015)	Fowler, M. (2015). Patterns of enterprise application architecture. Addison-Wesley.
(Martin, 2000)	Martin, R. C. (2000). Design principles and design patterns. Retrieved September 8, 2022, from http://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf
(Martin, 2012)	Martin, B. (2012, August 13). The Clean Architecture. Clean Coder Blog. Retrieved September 8, 2022, from https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
(Murphy-Hill et al., 2012)	E. Murphy-Hill, C. Parnin and A. P. Black, "How We Refactor, and How We Know It," in IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 5-18, Jan.-Feb. 2012, doi: 10.1109/TSE.2011.41.
(Calabrese et al., 2018)	Calabrese, J., Muñoz, R., Ariel, P., Silvia, E. (2017) Assistant for the Evaluation of Software Product Quality Characteristics Proposed by

ISO/IEC 25010 Based on GQM-Defined Metrics. Computer Science – CACIC 2017 (pp.164-175)

(Ivanković et al., 2019) Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. 2019. Code coverage at Google. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 955–963. https://doi.org/10.1145/3338906.3340459

(Pressman, 2014) Pressman, R., (2014). Software Engineering: A Practitioner's Approach, 8th
Ed, p.503-505. Boston, Mass.: McGraw Hill.

(Savoia, 2007) Savoia, A. (2007). Let the C.R.A.P. out of the bag. Pardon My French, But This Code Is C.R.A.P. (2). Retrieved October 2, 2022, from https://www.artima.com/weblogs/viewpost.jsp?thread=210575

Annex A – Unit Tests

```php
public static function callMethod($obj, $name, array $args) {
    $class = new \ReflectionClass($obj);
    $method = $class->getMethod($name);
    $method->setAccessible( accessible: true);
    return $method->invokeArgs($obj, $args);
}
```

Figure 52 – Unit tests helper function.

```php
public function testNavCode9101(): void
{
    //Given array $navItems and $extras with key 55_0 with velcro value
    $navItems = [];
    $extras = ['55_0' => 'VELCRO'];

    //When NavCode9101 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9101', [$navItems, $extras]);

    //Then assert that $navItems has 9101 in array
    $this->assertArrayHasKey( key: '9101', $navItemsResult);

    //Given array $navItems and $extras empty
    $extras = [];

    //When NavCode9101 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9101', [$navItems, $extras]);

    //Then assert that $navItems is not 9101 in array
    $this->assertArrayNotHasKey( key: '9101', $navItemsResult);
}
```

Figure 53 – Unit test for navCode9101 function.

```php
public function testGetShippingCodes9704(): void
{
    //Given array $navItems and $clientId value 137 and $countryId value 1 and $zoneId with value 1
    $navItems = [];
    $clientId = 137;
    $countryId = 1;
    $zoneId = 1;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9704 in array
    $this->assertArrayHasKey( key: '9704', $navItemsResult);
}
```

Figure 54 – Unit test for getShippingCodes9704 function.

```php
public function testGetShippingCodes9797(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 5 and $zoneId with value 1
    $navItems = [];
    $clientId = 1;
    $countryId = 5;
    $zoneId = 1;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9797 in array
    $this->assertArrayHasKey( key: '9797', $navItemsResult);
}
```

Figure 55 – Unit test for getShippingCodes9797 function.

```php
public function testGetShippingCodes9798(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 208 and $zoneId with value 1
    $navItems = [];
    $clientId = 1;
    $countryId = 208;
    $zoneId = 1;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9798 in array
    $this->assertArrayHasKey( key: '9798', $navItemsResult);

    //Given array $navItems and $clientId value 1 and $countryId value 209 and $zoneId with value 1
    $countryId = 209;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9798 in array
    $this->assertArrayHasKey( key: '9798', $navItemsResult);
}
```

Figure 56 – Unit test for getShippingCodes9798 function.

```php
public function testGetShippingCodes9799(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 74 and $zoneId with value 1
    $navItems = [];
    $clientId = 1;
    $countryId = 74;
    $zoneId = 1;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9799 in array
    $this->assertArrayHasKey( key: '9799', $navItemsResult);
}
```

Figure 57 – Unit test for getShippingCodes9799 function.

```php
public function testGetShippingCodes9701(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 1 and $zoneId with value 2
    $navItems = [];
    $clientId = 1;
    $countryId = 1;
    $zoneId = 2;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9701 in array
    $this->assertArrayHasKey( key: '9701', $navItemsResult);
}
```

Figure 58 – Unit test for getShippingCodes9701 function.

```php
public function testGetShippingCodes9702(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 1 and $zoneId with value 3
    $navItems = [];
    $clientId = 1;
    $countryId = 1;
    $zoneId = 3;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9702 in array
    $this->assertArrayHasKey( key: '9702', $navItemsResult);
}
```

Figure 59 – Unit test for getShippingCodes9702 function.

```php
public function testGetShippingCodes9703(): void
{
    //Given array $navItems and $clientId value 1 and $countryId value 1 and $zoneId with value 3
    $navItems = [];
    $clientId = 1;
    $countryId = 1;
    $zoneId = 1;

    //When getShippingCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getShippingCodes',
        [$navItems, $clientId, $countryId, $zoneId]);

    //Then assert that $navItems has 9703 in array
    $this->assertArrayHasKey( key: '9703', $navItemsResult);
}
```

Figure 60 – Unit test for getShippingCodes9703 function.

```php
public function testGetShippingCodes9123(): void
{
    //Given array $navItems and $extras empty array
    $navItems = [];
    $extras = [];

    //When navCode9123 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9123', [$navItems, $extras]);

    //Then assert that $navItems not has 9123 in array
    $this->assertArrayNotHasKey( key: '9123', $navItemsResult);

    //Given array $navItems and $extras with key 21_0 and value empty
    $extras = ['21_0' => ''];

    //When navCode9123 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9123', [$navItems, $extras]);

    //Then assert that $navItems not has 9123 in array
    $this->assertArrayNotHasKey( key: '9123', $navItemsResult);

    //Given array $navItems and $extras with key 21_0 and value NÃO
    $extras = ['21_0' => 'NÃO'];

    //When navCode9123 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9123', [$navItems, $extras]);

    //Then assert that $navItems not has 9123 in array
    $this->assertArrayNotHasKey( key: '9123', $navItemsResult);

    //Given array $navItems and $extras with key 21_0 and value NADA
    $extras = ['21_0' => 'NADA'];

    //When navCode9123 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9123', [$navItems, $extras]);

    //Then assert that $navItems not has 9123 in array
    $this->assertArrayNotHasKey( key: '9123', $navItemsResult);

    //Given array $navItems and $extras with key 21_0 and value SIM
    $extras = ['21_0' => 'SIM'];

    //When navCode9123 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9123', [$navItems, $extras]);

    //Then assert that $navItems has 9123 in array
    $this->assertArrayHasKey( key: '9123', $navItemsResult);
}
```

Figure 61 – Unit test for getShippingCodes9123 function.

```php
public function testGetShippingCodes9113(): void
{
    //Given array $navItems and $extras empty array
    $navItems = [];
    $extras = [];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9113', [$navItems, $extras]);

    //Then assert that $navItems not has 9113 in array
    $this->assertArrayNotHasKey( key: '9113', $navItemsResult);

    //Given array $navItems and $extras
    $extras = ['dentro_7_0' => 0, 'dentro_7_1' => 0, 'fora_8_0' => 0, 'fora_8_1' => 0];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9113', [$navItems, $extras]);

    //Then assert that $navItems not has 9113 in array
    $this->assertArrayNotHasKey( key: '9113', $navItemsResult);

    //Given array $navItems and $extras
    $extras = ['dentro_7_0' => 1, 'dentro_7_1' => 0, 'fora_8_0' => 0, 'fora_8_1' => 0];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9113', [$navItems, $extras]);

    //Then assert that $navItems has 9113 in array
    $this->assertArrayHasKey( key: '9113', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

    //Given array $navItems and $extras
    $extras = ['dentro_7_0' => 0, 'dentro_7_1' => 1, 'fora_8_0' => 0, 'fora_8_1' => 0];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9113', [$navItems, $extras]);

    //Then assert that $navItems has 9113 in array
    $this->assertArrayHasKey( key: '9113', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

    //Given array $navItems and $extras
    $extras = ['dentro_7_0' => 0, 'dentro_7_1' => 0, 'fora_8_0' => 1, 'fora_8_1' => 0];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'navCode9113', [$navItems, $extras]);
```

Figure 62 – Unit test for getShippingCodes9113 function – part 1.

```
//Then assert that $navItems has 9113 in array
$this->assertArrayHasKey( key: '9113', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
$this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

//Given array $navItems and $extras
$extras = ['dentro_7_0' => 0, 'dentro_7_1' => 0, 'fora_8_0' => 0, 'fora_8_1' => 1];

//When navCode9113 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9113', [$navItems, $extras]);

//Then assert that $navItems has 9113 in array
$this->assertArrayHasKey( key: '9113', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
$this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

//Given array $navItems and $extras
$extras = ['dentro_7_0' => 0, 'dentro_7_1' => 1, 'fora_8_0' => 0, 'fora_8_1' => 1];

//When navCode9113 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9113', [$navItems, $extras]);

//Then assert that $navItems has 9113 in array
$this->assertArrayHasKey( key: '9113', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
$this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

//Given array $navItems and $extras
$extras = ['dentro_7_0' => 1, 'dentro_7_1' => 0, 'fora_8_0' => 1, 'fora_8_1' => 0];

//When navCode9113 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9113', [$navItems, $extras]);

//Then assert that $navItems has 9113 in array
$this->assertArrayHasKey( key: '9113', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
$this->assertEquals( expected: 0.5, $navItemsResult['9113']['Quantity']);

//Given array $navItems and $extras
$extras = ['dentro_7_0' => 1, 'dentro_7_1' => 1, 'fora_8_0' => 0, 'fora_8_1' => 0];

//When navCode9113 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9113', [$navItems, $extras]);

//Then assert that $navItems has 9113 in array
$this->assertArrayHasKey( key: '9113', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
$this->assertEquals( expected: 1, $navItemsResult['9113']['Quantity']);
```

Figure 63 – Unit test for getShippingCodes9113 function – part 2.

```
    //Given array $navItems and $extras
    $extras = ['dentro_7_0' => 0, 'dentro_7_1' => 0, 'fora_8_0' => 1, 'fora_8_1' => 1];

    //When navCode9113 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9113', [$navItems, $extras]);

    //Then assert that $navItems has 9113 in array
    $this->assertArrayHasKey( key: '9113', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9113']);
    $this->assertEquals( expected: 1, $navItemsResult['9113']['Quantity']);

}
```

Figure 64 – Unit test for getShippingCodes9113 function – part 3.

```
public function testGetAdministrativeCostsCode(): void
{
    //Given array $navItems and $routeId value 1 and $customerId value 1 and $internalUserId with value 2
    $navItems = [];
    $routeId = 1;
    $customerId = 1;
    $internalUserId = 2;

    //When getAdministrativeCostsCode is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getAdministrativeCostsCode',
        [$navItems, $routeId, $customerId, $internalUserId]);

    //Then assert that $navItems has 9157 in array
    $this->assertArrayHasKey( key: '9157', $navItemsResult);

    //Given array $navItems and $routeId value 355 and $customerId value 1 and $internalUserId with value 2
    $routeId = 335;

    //When getAdministrativeCostsCode is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getAdministrativeCostsCode',
        [$navItems, $routeId, $customerId, $internalUserId]);

    //Then assert that $navItems not has 9157 in array
    $this->assertArrayNotHasKey( key: '9157', $navItemsResult);

    //Given array $navItems and $routeId value 1 and $customerId value 1 and $internalUserId with value 1
    $routeId = 1;
    $internalUserId = 1;

    //When getAdministrativeCostsCode is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getAdministrativeCostsCode',
        [$navItems, $routeId, $customerId, $internalUserId]);

    //Then assert that $navItems not has 9157 in array
    $this->assertArrayNotHasKey( key: '9157', $navItemsResult);

}
```

Figure 65 – Unit test for getAdministrativeCostsCode function.

```php
public function testNavCode9801(): void
{
    //Given array $navItems and $soladoSolto as zero
    $navItems = [];
    $soladoSolto = 0;

    //When navCode9801 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9801', [$navItems, $soladoSolto]);

    //Then assert that $navItems not has 9801 in array
    $this->assertArrayNotHasKey( key: '9801', $navItemsResult);

    //Given array $navItems and $soladoSolto as one
    $soladoSolto = 1;

    //When navCode9801 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9801', [$navItems, $soladoSolto]);

    //Then assert that $navItems has 9801 in array
    $this->assertArrayHasKey( key: '9801', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9801']);
    $this->assertEquals( expected: -1, $navItemsResult['9801']['Quantity']);

    //Given array $navItems and $soladoSolto as two
    $soladoSolto = 2;

    //When navCode9801 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9801', [$navItems, $soladoSolto]);

    //Then assert that $navItems has 9801 in array
    $this->assertArrayHasKey( key: '9801', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9801']);
    $this->assertEquals( expected: -0.5, $navItemsResult['9801']['Quantity']);
}
```

Figure 66 – Unit test for navCode9801 function.

```php
public function testNavCode9143(): void
{
    //Given array $navItems and $hasNeurological parameter as true
    $navItems = [];

    //When navCode9143 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9143', [$navItems, false]);

    //Then assert that $navItems not has 9143 in array
    $this->assertArrayNotHasKey( key: '9143', $navItemsResult);

    //Given array $navItems and $hasNeurological parameter as false

    //When navCode9143 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9143', [$navItems, true]);

    //Then assert that $navItems has 9143 in array
    $this->assertArrayHasKey( key: '9143', $navItemsResult);
}
```

Figure 67 – Unit test for navCode9143 function.

```php
public function testNavCode9141(): void
{
    //Given array $navItems and $hasNeurological parameter as true
    $navItems = [];

    //When navCode9141 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9141', [$navItems, false]);

    //Then assert that $navItems not has 9141 in array
    $this->assertArrayNotHasKey( key: '9141', $navItemsResult);

    //Given array $navItems and $hasNeurological parameter as false

    //When navCode9141 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9141', [$navItems, true]);

    //Then assert that $navItems has 9141 in array
    $this->assertArrayHasKey( key: '9141', $navItemsResult);
}
```

Figure 68 – Unit test for navCode9141 function.

```php
public function testNavCode9160(): void
{
    //Given array $navItems and $hasSpecialLeathers parameter as true
    $navItems = [];

    //When navCode9160 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9160', [$navItems, false]);

    //Then assert that $navItems not has 9160 in array
    $this->assertArrayNotHasKey( key: '9160', $navItemsResult);

    //Given array $navItems and $hasSpecialLeathers parameter as false

    //When navCode9160 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9160', [$navItems, true]);

    //Then assert that $navItems has 9160 in array
    $this->assertArrayHasKey( key: '9160', $navItemsResult);
}
```

Figure 69 – Unit test for navCode9160 function.

```php
public function testNavCode9161(): void
{
    //Given array $navItems and $hasEmbroideryOrLaser parameter as true
    $navItems = [];

    //When navCode9161 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9161', [$navItems, false]);

    //Then assert that $navItems not has 9161 in array
    $this->assertArrayNotHasKey( key: '9161', $navItemsResult);

    //Given array $navItems and $hasEmbroideryOrLaser parameter as false

    //When navCode9161 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9161', [$navItems, true]);

    //Then assert that $navItems has 9161 in array
    $this->assertArrayHasKey( key: '9161', $navItemsResult);
}
```

Figure 70 – Unit test for navCode9161 function.

```php
public function testNavCodeLastChanges(): void
{
    //Given array $navItems and array $extras
    $navItems = [];
    $extras = [];

    //When navCodeLastChanges is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCodeLastChanges',
        [$navItems, $extras]);

    //Then assert that $navItems not has 9158/9159 in array
    $this->assertArrayNotHasKey( key: '9158', $navItemsResult);
    $this->assertArrayNotHasKey( key: '9159', $navItemsResult);
```

Figure 71 – Unit test for navCodeLastChanges function – part 1.

```php
//Given array $navItems and array $extras
$extras = [
    'MOD1_73_0' => 1,
    'MOD1_73_1' => 1,
    'MOD2_74_0' => 1,
    'MOD2_74_1' => 1,
    'MOD3_75_0' => 1,
    'MOD3_75_1' => 1,
    'MOD4_76_0' => 1,
    'MOD4_76_1' => 1,
    'MOD5_77_0' => 1,
    'MOD5_77_1' => 1,
    'MOD6_78_0' => 1,
    'MOD6_78_1' => 1,
    'MOD7_79_0' => 1,
    'MOD7_79_1' => 1,
    'MOD8_80_0' => 1,
    'MOD8_80_1' => 1,
    'MOD9_81_0' => 1,
    'MOD9_81_1' => 1,
    'MOD10_82_0' => 1,
    'MOD10_82_1' => 1,
    'MOD11_83_0' => 1,
    'MOD11_83_1' => 1,
    'MOD12_87_0' => 1,
    'MOD12_87_1' => 1,
    'MOD13_88_0' => 1,
    'MOD13_88_1' => 1,
    'MOD14_89_0' => 1,
    'MOD14_89_1' => 1
];

//When navCodeLastChanges is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCodeLastChanges',
    [$navItems, $extras]);

//Then assert that $navItems not has 9159 in array
$this->assertArrayHasKey( key: '9159', $navItemsResult);
```

Figure 72 – Unit test for navCodeLastChanges function – part 2.

```
    //Given array $navItems and array $extras
    $extras = [
        'MOD1_73_0' => 1,
        'MOD1_73_1' => 1
    ];

    //When navCodeLastChanges is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCodeLastChanges',
        [$navItems, $extras]);

    //Then assert that $navItems not has 9158 in array
    $this->assertArrayHasKey( key: '9158', $navItemsResult);
}
```

Figure 73 – Unit test for navCodeLastChanges function – part 3.

```
public function testNavCode9150(): void
{
    //Given array $navItems and $extras and $shoeHeight variable
    $navItems = [];
    $extras = [];
    $shoeHeight = 0;

    //When navCode9150 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9150', [$navItems, $extras, $shoeHeight]);

    //Then assert that $navItems not has 9150 in array
    $this->assertArrayNotHasKey( key: '9150', $navItemsResult);

    //Given array $navItems and $extras and $shoeHeight variable
    $extras = [
        'esquerdo_20_0' => 'ESPECIAL',
        'esquerdo_20_type' => '1008',
        'direito_21_0' => 'ESPECIAL',
        'direito_21_type' => '1008'
    ];
```

Figure 74 – Unit test for navCode9150 function – part 1.

```php
//Given array $navItems and $extras and $shoeHeight variable
$extras = [
    'esquerdo_20_0' => 'ESPECIAL',
    'esquerdo_20_type' => '1008',
    'direito_21_0' => 'ESPECIAL',
    'direito_21_type' => '1008'
];

//When navCode9150 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9150', [$navItems, $extras, $shoeHeight]);

//Then assert that $navItems not has 9150 in array
$this->assertArrayNotHasKey( key: '9150', $navItemsResult);

//Given array $navItems and $extras and $shoeHeight variable
$extras = [
    'esquerdo_20_0' => 'ESPECIAL',
    'esquerdo_20_type' => '1022',
    'direito_21_0' => '',
    'direito_21_type' => ''
];

//When navCode9150 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9150', [$navItems, $extras, $shoeHeight]);

//Then assert that $navItems has 9150 in array
$this->assertArrayHasKey( key: '9150', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9150']);
$this->assertEquals( expected: 1, $navItemsResult['9150']['Quantity']);
$this->assertArrayHasKey( key: '9111', $navItemsResult);
```

Figure 75 – Unit test for navCode9150 function – part 2.

```php
    //Given array $navItems and $extras and $shoeHeight variable
    $extras = [
        'esquerdo_20_0' => '',
        'esquerdo_20_type' => '',
        'direito_21_0' => 'ESPECIAL',
        'direito_21_type' => '1022'
    ];

    //When navCode9150 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9150', [$navItems, $extras, $shoeHeight]);

    //Then assert that $navItems has 9150 in array
    $this->assertArrayHasKey( key: '9150', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9150']);
    $this->assertEquals( expected: 1, $navItemsResult['9150']['Quantity']);
    $this->assertArrayHasKey( key: '9111', $navItemsResult);

    //Given array $navItems and $extras and $shoeHeight variable
    $extras = [
        'esquerdo_20_0' => 'ESPECIAL',
        'esquerdo_20_type' => '1022',
        'direito_21_0' => 'ESPECIAL',
        'direito_21_type' => '1022'
    ];

    //When navCode9150 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9150', [$navItems, $extras, $shoeHeight]);

    //Then assert that $navItems has 9150 in array
    $this->assertArrayHasKey( key: '9150', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9150']);
    $this->assertEquals( expected: 2, $navItemsResult['9150']['Quantity']);
    $this->assertArrayHasKey( key: '9111', $navItemsResult);
}
```

Figure 76 – Unit test for navCode9150 function – part 3.

```php
public function testNavCode9102(): void
{
    //Given array $navItems and $extras and $isCrianca variable
    $navItems = [];
    $extras = [];
    $isCrianca = 1;

    //When navCode9102 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9102', [$navItems, $extras, $isCrianca]);

    //Then assert that $navItems not has 9102 in array
    $this->assertArrayNotHasKey( key: '9102', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        'esquerdo_16_0' => '25',
        'direito_17_0' => '25'
    ];

    //When navCode9102 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9102', [$navItems, $extras, $isCrianca]);

    //Then assert that $navItems not has 9102 in array
    $this->assertArrayNotHasKey( key: '9102', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        'esquerdo_16_0' => '26',
        'direito_17_0' => '25'
    ];

    //When navCode9102 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9102', [$navItems, $extras, $isCrianca]);

    //Then assert that $navItems has 9102 in array
    $this->assertArrayHasKey( key: '9102', $navItemsResult);

    //Given array $navItems and $extras and $isCrianca variable
    $extras = [
        'esq_36_0' => '25',
        'dir_37_0' => '26'
    ];
    $isCrianca = 0;

    //When navCode9102 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9102', [$navItems, $extras, $isCrianca]);

    //Then assert that $navItems has 9102 in array
    $this->assertArrayHasKey( key: '9102', $navItemsResult);
}
```

Figure 77 – Unit test for navCode9102 function.

```php
public function testNavCode9106(): void
{
    //Given array $navItems and $extras
    $navItems = [];
    $extras = [];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems not has 9106 in array
    $this->assertArrayNotHasKey( key: '9106', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '22_0' => ''
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems not has 9106 in array
    $this->assertArrayNotHasKey( key: '9106', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '22_0' => 'NÃO'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems not has 9106 in array
    $this->assertArrayNotHasKey( key: '9106', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '22_0' => 'ESQ E DRT'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems has 9106 in array
    $this->assertArrayHasKey( key: '9106', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9106']);
    $this->assertEquals( expected: 1, $navItemsResult['9106']['Quantity']);
```

Figure 78 – Unit test for navCode9106 function – part 1.

```php
    //Given array $navItems and $extras
    $extras = [
        '22_0' => 'ESQ E DIR'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems has 9106 in array
    $this->assertArrayHasKey( key: '9106', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9106']);
    $this->assertEquals( expected: 1, $navItemsResult['9106']['Quantity']);

    //Given array $navItems and $extras
    $extras = [
        '22_0' => 'ESQ'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems has 9106 in array
    $this->assertArrayHasKey( key: '9106', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9106']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9106']['Quantity']);

    //Given array $navItems and $extras
    $extras = [
        '22_0' => 'DRT'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems has 9106 in array
    $this->assertArrayHasKey( key: '9106', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9106']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9106']['Quantity']);
```

Figure 79 – Unit test for navCode9106 function – part 2.

```php
    //Given array $navItems and $extras
    $extras = [
        'esq_90_0' => 'ESQ',
        'dir_91_0' => 'ESQ'
    ];

    //When navCode9106 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9106', [$navItems, $extras]);

    //Then assert that $navItems has 9106 in array
    $this->assertArrayHasKey( key: '9106', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9106']);
    $this->assertEquals( expected: 1, $navItemsResult['9106']['Quantity']);
}
```

Figure 80 – Unit test for navCode9106 function – part 3.

```php
public function testNavCode9115(): void
{
    //Given array $navItems and $extras
    $navItems = [];
    $extras = [];

    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems not has 9115 in array
    $this->assertArrayNotHasKey( key: '9115', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        'dentro_9_0' => 0,
        'dentro_9_1' => 0,
        'fora_10_0' => 0,
        'fora_10_1' => 0
    ];

    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems not has 9115 in array
    $this->assertArrayNotHasKey( key: '9115', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        'dentro_9_0' => 1,
        'dentro_9_1' => 0,
        'fora_10_0' => 0,
        'fora_10_1' => 0
    ];

    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems has 9115 in array
    $this->assertArrayHasKey( key: '9115', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9115']['Quantity']);
```

Figure 81 – Unit test for navCode9115 function – part 1.

```php
//Given array $navItems and $extras
$extras = [
    'dentro_9_0' => 0,
    'dentro_9_1' => 1,
    'fora_10_0' => 0,
    'fora_10_1' => 0
];


//When navCode9115 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

//Then assert that $navItems has 9115 in array
$this->assertArrayHasKey( key: '9115', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
$this->assertEquals( expected: 0.5, $navItemsResult['9115']['Quantity']);

//Given array $navItems and $extras
$extras = [
    'dentro_9_0' => 0,
    'dentro_9_1' => 0,
    'fora_10_0' => 1,
    'fora_10_1' => 0
];


//When navCode9115 is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

//Then assert that $navItems has 9115 in array
$this->assertArrayHasKey( key: '9115', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
$this->assertEquals( expected: 0.5, $navItemsResult['9115']['Quantity']);
```

Figure 82 – Unit test for navCode9115 function – part 2.

```
    //Given array $navItems and $extras
    $extras = [
        'dentro_9_0' => 0,
        'dentro_9_1' => 0,
        'fora_10_0' => 0,
        'fora_10_1' => 1
    ];


    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems has 9115 in array
    $this->assertArrayHasKey( key: '9115', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
    $this->assertEquals( expected: 0.5, $navItemsResult['9115']['Quantity']);

    //Given array $navItems and $extras
    $extras = [
        'dentro_9_0' => 0,
        'dentro_9_1' => 1,
        'fora_10_0' => 1,
        'fora_10_1' => 1
    ];


    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems has 9115 in array
    $this->assertArrayHasKey( key: '9115', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
    $this->assertEquals( expected: 1, $navItemsResult['9115']['Quantity']);

    //Given array $navItems and $extras
    $extras = [
        'dentro_9_0' => 1,
        'dentro_9_1' => 1,
        'fora_10_0' => 1,
        'fora_10_1' => 1
    ];


    //When navCode9115 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9115', [$navItems, $extras]);

    //Then assert that $navItems has 9115 in array
    $this->assertArrayHasKey( key: '9115', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9115']);
    $this->assertEquals( expected: 1, $navItemsResult['9115']['Quantity']);
}
```

Figure 83 – Unit test for navCode9115 function – part 3.

```php
public function testNavCode9107(): void
{
    //Given array $navItems and $priorityId as normal (1)
    $navItems = [];
    $priorityId = 1;

    //When navCode9107 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9107', [$navItems, $priorityId]);

    //Then assert that $navItems not has 9107 in array
    $this->assertArrayNotHasKey( key: '9107', $navItemsResult);

    //Given array $navItems and $priorityId as urgent (2)
    $priorityId = 2;

    //When navCode9107 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9107', [$navItems, $priorityId]);

    //Then assert that $navItems has 9107 in array
    $this->assertArrayHasKey( key: '9107', $navItemsResult);
}
```

Figure 84 – Unit test for navCode9107 function.

```php
public function testNavCode9116(): void
{
    //Given array $navItems and $extras
    $navItems = [];
    $extras = [];

    //When navCode9116 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9116', [$navItems, $extras]);

    //Then assert that $navItems not has 9116 in array
    $this->assertArrayNotHasKey( key: '9116', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '44_0' => ''
    ];

    //When navCode9116 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9116', [$navItems, $extras]);

    //Then assert that $navItems not has 9116 in array
    $this->assertArrayNotHasKey( key: '9116', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '44_0' => 'NÃO'
    ];

    //When navCode9116 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9116', [$navItems, $extras]);

    //Then assert that $navItems not has 9116 in array
    $this->assertArrayNotHasKey( key: '9116', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '44_0' => 'SIM'
    ];

    //When navCode9116 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9116', [$navItems, $extras]);

    //Then assert that $navItems has 9116 in array
    $this->assertArrayHasKey( key: '9116', $navItemsResult);
}
```

Figure 85 – Unit test for navCode9116 function.

```php
public function testNavCode9121(): void
{
    //Given array $navItems and $extras
    $navItems = [];
    $extras = [];

    //When navCode9121 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9121', [$navItems, $extras]);

    //Then assert that $navItems not has 9121 in array
    $this->assertArrayNotHasKey( key: '9121', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '55_0' => ''
    ];

    //When navCode9121 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9121', [$navItems, $extras]);

    //Then assert that $navItems not has 9121 in array
    $this->assertArrayNotHasKey( key: '9121', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '55_0' => 'LACE'
    ];

    //When navCode9121 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9121', [$navItems, $extras]);

    //Then assert that $navItems not has 9121 in array
    $this->assertArrayNotHasKey( key: '9121', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '55_0' => 'QLS'
    ];

    //When navCode9121 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9121', [$navItems, $extras]);

    //Then assert that $navItems has 9121 in array
    $this->assertArrayHasKey( key: '9121', $navItemsResult);
}
```

Figure 86 – Unit test for navCode9121 function.

```php
public function testNavCode9154(): void
{
    //Given array $navItems and $extras
    $navItems = [];
    $extras = [];

    //When navCode9154 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9154', [$navItems, $extras]);

    //Then assert that $navItems not has 9154 in array
    $this->assertArrayNotHasKey( key: '9154', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '52_0' => ''
    ];

    //When navCode9154 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9154', [$navItems, $extras]);

    //Then assert that $navItems not has 9154 in array
    $this->assertArrayNotHasKey( key: '9154', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '52_0' => 'ESQUERDO'
    ];

    //When navCode9154 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9154', [$navItems, $extras]);

    //Then assert that $navItems has 9154 in array
    $this->assertArrayHasKey( key: '9154', $navItemsResult);

    //Given array $navItems and $extras
    $extras = [
        '52_0' => 'DIREITO'
    ];

    //When navCode9154 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9154', [$navItems, $extras]);

    //Then assert that $navItems has 9154 in array
    $this->assertArrayHasKey( key: '9154', $navItemsResult);
```

Figure 87 – Unit test for navCode9154 function – part 1.

```php
    //Given array $navItems and $extras
    $extras = [
        '52_0' => 'ESQ E DIR'
    ];

    //When navCode9154 is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'navCode9154', [$navItems, $extras]);

    //Then assert that $navItems has 9154 in array
    $this->assertArrayHasKey( key: '9154', $navItemsResult);
}
```

Figure 88 – Unit test for navCode9154 function – part 2.

```php
public function testGetRockerCodes(): void
{
    //Given array $navItems and $extras empty array and $defaultSoleRocker empty array
    $navItems = [];
    $extras = [];
    $defaultSoleRocker = [];

    //When getRockerCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
        [$navItems, $extras, $defaultSoleRocker]);

    //Then assert that $navItems not has 9156 or 9155 in array
    $this->assertArrayNotHasKey( key: '9156', $navItemsResult);
    $this->assertArrayNotHasKey( key: '9155', $navItemsResult);

    $extras = [
        'ROCKER-BACK_84_0' => 0,
        'ROCKER-MIDDLE_85_0' => 0,
        'ROCKER-FRONT_86_0' => 0,
        'ROCKER_TYPE_0' => 0,
        'ROCKER-BACK_84_1' => 0,
        'ROCKER-MIDDLE_85_1' => 0,
        'ROCKER-FRONT_86_1' => 0,
        'ROCKER_TYPE_1' => 0
    ];

    //When getRockerCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
        [$navItems, $extras, $defaultSoleRocker]);

    //Then assert that $navItems not has 9156 or 9155 in array
    $this->assertArrayNotHasKey( key: '9156', $navItemsResult);
    $this->assertArrayNotHasKey( key: '9155', $navItemsResult);

    $defaultSoleRocker = [0, 0, 0];

    //When getRockerCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
        [$navItems, $extras, $defaultSoleRocker]);

    //Then assert that $navItems not has 9156 or 9155 in array
    $this->assertArrayNotHasKey( key: '9156', $navItemsResult);
    $this->assertArrayNotHasKey( key: '9155', $navItemsResult);
```

Figure 89 – Unit test for getRockerCodes function – part 1.

```php
$extras = [
    'ROCKER-BACK_84_0' => 6,
    'ROCKER-MIDDLE_85_0' => 6,
    'ROCKER-FRONT_86_0' => 0,
    'ROCKER_TYPE_0' => 2,
    'ROCKER-BACK_84_1' => 6,
    'ROCKER-MIDDLE_85_1' => 6,
    'ROCKER-FRONT_86_1' => 0,
    'ROCKER_TYPE_1' => 2
];
$defaultSoleRocker = [6.0, 6.0, 0.0];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems not has 9156 or 9155 in array
$this->assertArrayNotHasKey( key: '9156', $navItemsResult);
$this->assertArrayNotHasKey( key: '9155', $navItemsResult);

$defaultSoleRocker = [0, 0, 0];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems has 9155 in array
$this->assertArrayHasKey( key: '9155', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9155']);
$this->assertEquals( expected: 2, $navItemsResult['9155']['Quantity']);
```

Figure 90 – Unit test for getRockerCodes function – part 2.

```php
$extras = [
    'ROCKER-BACK_84_0' => 26,
    'ROCKER-MIDDLE_85_0' => 26,
    'ROCKER-FRONT_86_0' => 0,
    'ROCKER_TYPE_0' => 2,
    'ROCKER-BACK_84_1' => 26,
    'ROCKER-MIDDLE_85_1' => 26,
    'ROCKER-FRONT_86_1' => 0,
    'ROCKER_TYPE_1' => 2
];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems has 9156 in array
$this->assertArrayHasKey( key: '9156', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9156']);
$this->assertEquals( expected: 2, $navItemsResult['9156']['Quantity']);

$extras = [
    'ROCKER-BACK_84_0' => 6,
    'ROCKER-MIDDLE_85_0' => 6,
    'ROCKER-FRONT_86_0' => 0,
    'ROCKER_TYPE_0' => 2,
    'ROCKER-BACK_84_1' => 6,
    'ROCKER-MIDDLE_85_1' => 6,
    'ROCKER-FRONT_86_1' => 0,
    'ROCKER_TYPE_1' => 2
];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems has 9155 in array
$this->assertArrayHasKey( key: '9155', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9155']);
$this->assertEquals( expected: 2, $navItemsResult['9155']['Quantity']);
```

Figure 91 – Unit test for getRockerCodes function – part 3.

```php
$extras = [
    'ROCKER-BACK_84_0' => 26,
    'ROCKER-MIDDLE_85_0' => 26,
    'ROCKER-FRONT_86_0' => 0,
    'ROCKER_TYPE_0' => 2,
    'ROCKER-BACK_84_1' => 26,
    'ROCKER-MIDDLE_85_1' => 26,
    'ROCKER-FRONT_86_1' => 0,
    'ROCKER_TYPE_1' => 2
];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems has 9156 in array
$this->assertArrayHasKey( key: '9156', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9156']);
$this->assertEquals( expected: 2, $navItemsResult['9156']['Quantity']);

$extras = [
    'ROCKER-BACK_84_0' => 6,
    'ROCKER-MIDDLE_85_0' => 6,
    'ROCKER-FRONT_86_0' => 0,
    'ROCKER_TYPE_0' => 2,
    'ROCKER-BACK_84_1' => 0,
    'ROCKER-MIDDLE_85_1' => 0,
    'ROCKER-FRONT_86_1' => 0,
    'ROCKER_TYPE_1' => 0
];
$defaultSoleRocker = [6, 6, 0];

//When getRockerCodes is called
$navItemsResult = UnitTester::callMethod($this->navisionMCOObj, name: 'getRockerCodes',
    [$navItems, $extras, $defaultSoleRocker]);

//Then assert that $navItems has 9155 in array
$this->assertArrayHasKey( key: '9155', $navItemsResult);
$this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9155']);
$this->assertEquals( expected: 2, $navItemsResult['9155']['Quantity']);
```

Figure 92 – Unit test for getRockerCodes function – part 4.

```
    $extras = [
        'ROCKER-BACK_84_0' => 26,
        'ROCKER-MIDDLE_85_0' => 26,
        'ROCKER-FRONT_86_0' => 0,
        'ROCKER_TYPE_0' => 2,
        'ROCKER-BACK_84_1' => 0,
        'ROCKER-MIDDLE_85_1' => 0,
        'ROCKER-FRONT_86_1' => 0,
        'ROCKER_TYPE_1' => 0
    ];

    //When getRockerCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getRockerCodes',
        [$navItems, $extras, $defaultSoleRocker]);

    //Then assert that $navItems has 9156 in array
    $this->assertArrayHasKey( key: '9156', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9156']);
    $this->assertEquals( expected: 1, $navItemsResult['9156']['Quantity']);

    $extras = [
        'ROCKER-BACK_84_0' => 0,
        'ROCKER-MIDDLE_85_0' => 0,
        'ROCKER-FRONT_86_0' => 0,
        'ROCKER_TYPE_0' => 0,
        'ROCKER-BACK_84_1' => 26,
        'ROCKER-MIDDLE_85_1' => 26,
        'ROCKER-FRONT_86_1' => 0,
        'ROCKER_TYPE_1' => 2
    ];

    //When getRockerCodes is called
    $navItemsResult = UnitTester::callMethod($this->navisionMCOObj,  name: 'getRockerCodes',
        [$navItems, $extras, $defaultSoleRocker]);

    //Then assert that $navItems has 9156 in array
    $this->assertArrayHasKey( key: '9156', $navItemsResult);
    $this->assertArrayHasKey( key: 'Quantity', $navItemsResult['9156']);
    $this->assertEquals( expected: 1, $navItemsResult['9156']['Quantity']);

}
```

Figure 93 – Unit test for getRockerCodes function – part 5.