

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Improving quality and agility of safety-critical software development using domain-specific languages

João Ricardo Faria Mendes Almeida Reis



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ademar Aguiar

February 20, 2021

Improving quality and agility of safety-critical software development using domain-specific languages

João Ricardo Faria Mendes Almeida Reis

Mestrado Integrado em Engenharia Informática e Computação

February 20, 2021

Abstract

Safety-critical systems require a higher level of trust regarding its proper functioning compared to non-critical contexts because failures in their hardware and software components may lead to loss of large amounts of funds, serious environmental damage, user harm or death.

Domain-Specific Languages (DSL) may help improve both the quality and agility in the development of safety-critical software systems. To understand if and how this is possible, a research about the use of Domain-Specific Languages connected to safety-critical systems is made, trying to figure out ways to do it in a concrete set of cases.

A DSL is a small computer language that specializes in a specific application domain, focused on a specific aspect of a software system. It is used to express the requirements and solutions of an architectural domain. There are two types of Domain-Specific Languages, external and internal. External Domain-Specific Languages are a language that's parsed independently of the host general-purpose language, for example, CSS. Domain-Specific Languages that are written in the host language to make it feel like a different one is known as Internal Domain-Specific Languages.

A well-designed DSL is more straightforward to program with than a traditional library. Domain-Specific Languages have been used on earthquake sensors making the development of software for wireless sensors less complicated. Furthermore, models made with domain-specific languages are simpler to validate, this improves programmer productivity and consequently can improve the agility.

This dissertation focus is to analyze the key activities of safety-critical software systems development that would benefit from Domain-Specific Languages in general, and to select and customize tools and techniques to support the development and validation of safety-critical systems.

Taking advantage of Behaviour Driven Development, an extension to Gherkin, a domain-specific language which helps to describe business behaviour without the need to go into detail of implementation, will be done providing a tool that is suitable to the Safety-Critical Systems domain. A test suite is developed through three case studies to see its behaviour and how it can improve the quality and agility in these systems.

Resumo

Os sistemas críticos de segurança exigem um nível mais alto de confiança em relação ao seu funcionamento adequado, em comparação com contextos não críticos, devido a falhas nos componentes de hardware e software que podem levar à perda de grandes quantidades de fundos, graves danos ambientais, danos ao utilizador ou morte.

As linguagens específicas de domínio podem ajudar a melhorar a qualidade e a agilidade no desenvolvimento de sistemas de software críticos para a segurança. Para entender se é possível, é feita uma pesquisa sobre o uso de linguagens específicas de domínio conectadas a sistemas críticos, com o objectivo de descobrir se devem ser usadas ou não no nosso caso.

Uma linguagem específica de domínio é uma pequena linguagem de computador especializada num domínio de aplicação específico, focada num aspecto específico de um sistema de software. É usado para expressar os requisitos e soluções de um domínio arquitetural. Existem dois tipos de idiomas específicos do domínio, externo e interno. Os idiomas externos específicos do domínio são analisados independentemente da linguagem de uso geral do *host*, por exemplo, CSS. Os idiomas específicos do domínio que são escritos no idioma do *host* de modo a que se pareça com um idioma diferente, são conhecidos como linguagens específicas de domínio interno.

Uma linguagem específica de domínio bem elaborada é mais simples de programar do que uma biblioteca tradicional. As linguagens específicas de domínio já foram usados em sensores de terremotos, tornando o desenvolvimento de software para sensores sem fios menos complicado. Além disso, os modelos feitos com linguagens específicas de domínio são mais simples de validar, o que melhora a produtividade do programador e, conseqüentemente, pode melhorar a agilidade.

O foco da dissertação é analisar as principais atividades de desenvolvimento de sistemas de software críticos que podem beneficiar das linguagens específicas do domínio em geral, selecionar e customizar ferramentas e técnicas para apoiar o desenvolvimento e a validação de sistemas críticos.

Com recurso ao Behavior Driven Development, uma extensão de Gherkin, uma linguagem específica de domínio que ajuda a descrever o comportamento dos negócios sem a necessidade de entrar em detalhes de implementação, será feita. Para que seja possível fornecer uma ferramenta adequada ao domínio de Sistemas Críticos. A suite de testes criada é desenvolvida ao longo de três casos de estudo para analisar o comportamento da mesma. Tal como o modo em que esta pode melhorar a qualidade e agilidade nesses sistemas.

*“If we did all the things we are capable of,
we would literally astound ourselves.”*

Thomas A. Edison

Contents

1	Introduction	1
1.1	Context & Motivation	1
1.2	Problem	3
1.3	Objectives and Expected Results	3
1.4	Document Structure	3
2	State of Art	5
2.1	Domain-Specific Languages	5
2.1.1	Domain-Specific Languages applied to Safety-Critical Systems	6
2.1.2	Existing tools to support Domain-Specific Languages	8
2.1.3	Summary	8
2.2	Agile	9
2.3	Software Quality	9
2.3.1	Verification and Validation	11
2.3.2	V&V in Waterfall	13
2.3.3	V&V in Agile	13
2.4	Behaviour-Driven Development	16
2.4.1	Extending BDD	16
2.4.2	Cucumber and Gherkin	20
2.4.3	Summary	26
3	Problem Statement	29
3.1	Current Issues	29
3.2	Assumptions	29
3.3	Hypothesis	29
3.4	Research Questions	30
3.5	Methodology	30
4	Proposed Solution	31
4.1	Case Studies Overview	31
4.2	The Cucumber for Java Book - ATM	32
4.3	Russell C. Bjork - ATM with GUI	35
4.4	Boeing 737 MAX - MCAS System	43
4.5	Implementation Guide	46
4.6	Summary	49

5	Validation	51
5.1	Agility Validation	52
5.2	Quality Validation	54
5.3	Summary	58
6	Conclusions	59
6.1	Contributions	59
6.2	Future Work	60
	References	61

List of Figures

2.1	ISO/IEC 25010 categorization of software quality requirements [12].	10
2.2	Verification & Validation Framework (IEEE 1012).	12
2.3	Validation VS Verification [10].	12
2.4	Validation & Verification in Waterfall [44].	13
2.5	Verification & validation processes implemented in each iteration of Agile development [42].	14
2.6	Waterfall model vs. Agile methods life cycle [38].	15
2.7	Verification & validation techniques [6].	15
2.8	BDD steps example [2]	17
2.9	Layer approach proposed [37].	19
2.10	Aquila Functional Flowchart [58].	19
2.11	Aquila Functional Flowchart [50].	20
2.12	Structure from features down to automation library. [45]	21
2.13	Feature file example [45].	22
2.14	Repeated Given. [2]	22
2.15	Use of And and But. [2]	23
2.16	Scenario outline. [2]	23
2.17	Background Example. [2]	24
2.18	Rule Example. [2]	25
2.19	Example of a Cucumber Expression with a Java method connected to it, known as Step Definition. [2]	26
2.20	Scenario Hook Example. [2]	26
2.21	Step Hook Example. [2]	26
2.22	Conditional Hook Example. [2]	26
4.1	Functional Requirement. [1]	31
4.2	Functional Requirements vs Non-Functional Requirements. [1]	32
4.3	Use Case Diagram - Adapted Cucumber ATM.	33
4.4	Generic ATM test.	34
4.5	ATM test with Compliance Check.	34
4.6	Use Case Diagram - ATM with GUI.	36
4.7	Session Sequence Diagram. [46]	37
4.8	Transaction Sequence Diagram. [46]	37
4.9	Mock object example using mockito - List class mock. [39]	38
4.10	Adapter example. [49]	39
4.11	"Real-World" Facade analogy. [49]	40
4.12	Angle-of-attack Sensors. [48]	43
4.13	Activity Diagram - Roles	47

4.14	Source folder organization.	48
4.15	Facade implementation in the system.	49
5.1	Waterfall Example [43]	52
5.2	V-Model Example [8].	52
5.3	Impact on Agile Manifesto Principles.	54
5.4	Impact on Quality.	57

Abbreviations and Symbols

DSL	Domain-Specific Language
DSEL	Domain-Specific Embedded Language
GPL	General-Purpose Language
BDD	Behaviour Driven Development
SCS	Safety-Critical System
TDD	Test Driven Development
DSML	Domain-Specific Modeling Language
UML	Unified Modeling Language
IMA	Integrated Modular Avionics
LWB	Language Workbenches
QA	Quality Assurance
V&V	Verification and Validation
FSS	Fire Sensor System

Chapter 1

Introduction

Agile methodologies argue that the most important is to seek client satisfaction through continuous deliveries of value-adding software, keeping up constant communication with the client and between team members [35]. In opposite to traditional safety-critical development, Agile methodologies do not require the complete definition of a product, a complete analysis, or a definition of all requirements a priori, but instead, adopt a dynamic interaction between all involved that allows constant evolution and frequent releases.

1.1 Context & Motivation

Verification and Validation is intensively used in agile processes as one of the core strengths of these methodologies. Verification and validation are two procedures used together to guarantee that a service or system meets requirements and specifications and if it is the right product. Furthermore, automated testing is emphasized by using of a separate software from the system software that need to be tested in a way that it can control the execution of tests and compare the outcomes with the expected ones. It can increase the depth and scope of tests and thus it will improve software quality.

A safety-critical system (SCS) is a system in which failure or malfunction may result in death, serious injury to people, loss or severe damage to equipment or environmental harm.

SCS have to be developed very carefully to prevent these malfunctions due to system failures. Some mechanisms of SCS are challenging to design, test and use correctly in the system context [30]. Thus, they require a higher level of trust regarding their proper functioning compared to non-critical contexts.

SCSs are incredibly dependent on computers, and it is up to these computers to ensure that there are no failures in their usage. The most important property of the system is that it is dependable, which is a measure of a system's availability, reliability, maintainability and in some cases, other characteristics such as durability, safety and security are also important. Dependability shows the users trust in that system.

Grant, Emanuel analyses the ethical considerations in SCSs Development where he states that the real concerns with the failure in operations and mal-performance of SCSs have heightened a review of the used methodologies and strategies applied in the development of these systems. After examining the issues in the development of reliable software systems, focusing on SCSs, the author concluded that the methodologies, standards and notations demonstrated that there is a necessity, in those domains, but sufficiency is not achieved. Furthermore, there should be an adherence to licensing policy for software developers with a based core set of ethical principals, wherein the violation of these principles by the practitioner would incur in penalties [19].

In addition to the software development tools and techniques usually used (Unified Modeling Language and Formal Specification Techniques), many safety-critical application domains have defined software development strategies that are specific to their respective application domains such as DSLs [15]. As the software engineering methodologies and notations evolved, there was a need for domain-specific strategies in different development domains. ISO established several standards for the field of information technology, some of them developed for specific application domains, like the Medical field or Fire Alarm Systems.

Quality standards are used to ensure that companies fulfil the minimum requirements to become an integral part of almost every industry from healthcare to aerospace. These standards are documents that provide requirements, specifications, guidelines, or characteristics that should be used to ensure that materials, products, processes or services have the necessary quality.

Standards are essential in any organization. Standards present precise descriptions and terminology, of how the final product is done. We can think of standards as a formula that describes the best way known of doing something. Committee ISOTC21/SC3 contains the standards for Fire Detection and Alarm Systems, ISO7240, these standards must be followed to develop the sensors and the alarms, to ensure their safety and quality.

The book written by [34] contains the most critical safety certification required by commercial and military aircraft's, the DO-178C standards. They describe the requirements for certification of software systems in airborne operation. However it does not specify how these requirements are met, this means that software developers need to incorporate process workflows, notations, and the methodologies that comply with all the objectives stated in DO-178C.

Building confidence in a critical system can be done by following particular well-defined architectures, development processes and tools, and is conducted by several regulatory bodies, different organisational and cost structures, and different standards, such as DO-178C for aviation or ISO 7240-14 for Fire Detection and Alarm Systems. Domain-Specific Languages (DSL) can provide a language in the specific domain of the critical system, improving the workflow and reducing costs.

DSLs are widely used to deal with a specific domain or a set of concerns. They offer substantial gains because they are easy to use compared with the programming languages with general purposes due to the higher-level of abstractions, implementation efficiency, and less effort in validation and verification [36].

DSLs can be used by experts in the domain, not necessarily developers, potentially leading to faster iterations, shorter feedback loops (from domain experts to system implementers), and

therefore may improve the overall agility of the software development cycle. Furthermore, we can use them to improve the quality of software using it in automated tests to verify and validate it.

The objective of this work is to determine if and how Domain-Specific Languages can help to improve the agility and quality on safety-critical software systems development.

1.2 Problem

Safety-critical systems require a higher level of trust, compliance with several standards is needed to guarantee that the systems are trustable and have the needed quality . Currently, the use of agile methodologies in safety-critical systems development is growing, but it is not used to its full potential. Despite existing some techniques, there might exist other agile ways to improve verification and validation of the SCS implemented in real-world environments.

1.3 Objectives and Expected Results

The main objective is to research how DSLs can help improve the quality and agility in the development of safety-critical software systems, using them to do the specification, verification and validation. In concrete, we aim to find a way to give an increase in productivity writing the acceptance tests, guarantee that the tests are easy to understand by anyone involved in the project, improving the development by giving a better understanding of the requirements to developers as well as automating the testing of the system. The strategy and expected results in order to meet these objectives are:

- Select a subdomain of functional and non-functional requirements for an ATM system:
 - derived from standards for ATM systems;
- Design and implement a test suite based on Gherkin to support specification, validation and verification of concrete ATM systems;
- Use the test suite in a safety-critical system related with avionics;
- Analysis and selection of the key activities of safety-critical software systems development that would benefit from DSLs and Agile, more specifically Verification & Validation;
- Create a test suite to support the above activities.

1.4 Document Structure

Along with the introduction, this report is composed of four other chapters. Chapter [State of Art](#) presents an overview of the Domain-Specific Languages (DSL), Software Quality, Agile, and Verification and Validation using DSLs through Behaviour-Driven Development. Chapter [Problem Statement](#) presents the current issues, our assumptions, our hypothesys, the research questions

of this dissertation and the methodology used to develop it. Chapter [Proposed Solution](#) presents the problems found and the proposed solutions for it. It ends with Chapter [Conclusions](#), where conclusions of our state of the art are presented along with the planned work to do until the end of the dissertation.

Chapter 2

State of Art

This chapter overviews the contents related with Domain-Specific Languages and the current status of what has already been explored related to safety-critical systems. In addition, we discuss software quality, agile methods, Behaviour Driven Development and also the existing tools to support the usage of DSLs for the purpose of V&V.

2.1 Domain-Specific Languages

A Domain-Specific Language (DSL) is usually a small computer language that specializes in a specific application domain (e.g. Railways, Healthcare, Robotics, Aerospace, Mathematical Computing and Stocks) focused on a specific aspect of a software system, and it is used to express the requirements and solutions of an architectural domain [15].

It allows solutions to be expressed in the idiom and at the level of abstraction of the problem domain, offering substantial gains in ease of use compared with other types of languages, reducing the complexity of software development in a domain. Contrasting, General-Purpose Language (GPL), (e.g. Python, C++, Java, Scala) can be used across domains as they are not specific, which is more complex to use in specific domains derived due to their generality. A well-designed DSL is more straightforward to program with than a traditional library, which improves productivity [15].

Two types of DSLs are known [14], related with the way they are implemented, internal and external. An External DSL is a language that is parsed independently of the host general-purpose language (e.g. Cucumber and CSS). DSLs that take advantage of a host language (GPL), using its tools and capabilities, making it feel like a different one, are known as Internal DSLs.

Visic et al [55] introduce a DSL supporting the realization of modelling methods [31] that abstracts away from typical meta-modelling environments. A metamodel is a model of a model and meta-modeling is the process of creating such metamodels. It allows the method engineer to focus on the conceptual building blocks of the modelling method instead of meta-modelling platform's technical specificities. It is stated by the authors that modelling methods may evolve iteratively based on changing modelling requirements and feedback loops supporting an agile modelling

method. It was concluded that using modelling method engineering as an application domain for a DSL enables code-based platform-independent definition of conceptual modelling methods. The language designed was flexible to include native code for the targeted meta-modelling platform if its features are insufficient for specific cases. However, compilers need to be written in order to deploy the modelling method definition on known meta-modelling platforms.

Im et al [26] study on an Automating test cases using DSLs demonstrate a toolchain to support a model-driven development process for creating test cases. The technic uses a DSL in which use cases and test cases are written taking advantage of the test patterns. It is possible to create templates that support the automatic generation of executable test cases. However, there is still some work to do incorporating the patterns. Also, the templates of all patterns need to be generalized.

2.1.1 Domain-Specific Languages applied to Safety-Critical Systems

Bull [7] argues that there is a lack of intuitive and immediate understanding due to software expressed in terms and concepts (variables, types, or memory allocations) which do not reflect the real-world terms and concepts from the software problem domain. Comprehension is essential in SCS, and part of his project was to allow software's safety properties to be audited and assessed easier than it would be using the source code. He concluded that some phenomena affect safety and cannot be improved. However, improving program comprehension, particularly of the way that the software and the real world interacts can have some advantages. The author proposed that the right way of doing it was by using DSLs as they have significant benefits.

DSL applied to Earthquake Sensors

Sadilek [47] proposed a prototype Domain-Specific Language for Wireless Sensor Networks. He stated that developing software for wireless sensor networks is complicated. He defends that using Domain-specific languages may help with this problem. In his article, he proposes a language engineering approach that is inexpensive to define and have a syntax that domain experts understand, creating easy simulations. He combined meta-programmable language Scheme and the metamodeling technology EMF [52] without changes to Scheme representation. A prototypical toolset was implemented with earthquake detection sensors. The results showed how Scheme could be combined with Eclipse EMF. Models in stream-oriented DSL can be executed on a target platform without modification, and that redundancy in the description of a DSL's operational semantics for simulation and a target platform can be avoided. With a meta-programmable language, it is possible to raise the abstraction from platform concepts to domain-specific concepts. He argued that currently famous metamodeling technologies are not sufficient to provide domain experts with DSL's to integrate them into the development process of software for Wireless Sensor Networks and proposed the usage of metaprogramming in combination with metamodeling technologies. However, he concluded that his approach has worse performance than low-level coding. Lots of questions are still open.

DSL applied to Avionics

Grant [18] states that the use of Domain-specific Modeling Languages (DSMLs), which are part of the DSLs family, and some associated methodologies, provide support when the application domain needs to be safe and reliable. For software systems to be used onboard an aircraft, they must be certified using specific protocols, established for the development of these SCS. These protocols represented as textual documents are difficult to apply directly in the development environments. The authors did two case studies to support their theory where they transformed the specifications of DO-178C, for SCS development, from a textual representation to a graphical representation. It was done using the Unified Modeling Language (UML) notation. Furthermore, it was developed a model-driven software methodology adapted to DO-178C specification, making use of its graphical representation, to develop software systems for onboard avionic operation. The result of this research was mainly a DSML for avionic SCS that is compliant with the DO-178C specification. However, the research is ongoing toward developing tools to automate aspects of using DSMLs in this domain development.

Alves et al [3] report a solution for the avionics software industry, using DSL to assist the Integrated Modular Avionics (IMA) developers. IMA are real-time computer network airborne systems, consisting of a network of physical computing modules that allow real-time processing, capable of supporting multiple applications of different criticality levels. This means that a set of avionics functions that may be correctly executed, on the same hardware with guarantees that the external behaviour of each application, is the same as the one performed in specific dedicated hardware. Furthermore, the last generations of aircraft have IMA architecture that provides several benefits. The usability of the generated DSL and the previous programming experience in C was evaluated and lead to a positive result. The solution was well accepted and was stated that a few lines of code have to be introduced to finalise the intended functionality.

DSL applied to Railway

Idani et al [25] proposed a DSML-tool with formal semantics to be used in Railway SCS. In order to assist domain experts, there are several tools available to define graphical or textual DSMLs. The models that result from these tools are useful but are insufficient sometimes for an overall understanding of the systems, especially when there are formal methods [23]. Moreover, formal methods failures often result from a misunderstanding of the requirements from the developers. The paper presented an incremental formal development process involving domain experts during the modelling activities, where coloured Petri-nets were used since their graphical notation is more readable than textual mathematical notations. Then DSMLs were used to assist the domain experts in the design of the domain models. The Coloured Petri-nets and DSMLs are suitable for validation because they favour communication, complementing each other by focusing on particular concerns. Finally, it was concluded that introducing these machines, manually, is difficult and time-consuming when the Coloured Petri-nets scale up. The approach was successfully applied to

a railway SCS and other case studies like automatic car light regulator. Also, the authors planned to make an empirical study with railway experts to validate the usability of the toolset.

2.1.2 Existing tools to support Domain-Specific Languages

Voelter et al [56] state that Language Workbenches (LWB), supports the efficient development of languages [11], support the efficient creation, integration, and use of DSLs. They execute models by code generation to programming language code which can give increased productivity, also higher quality. However, as the SCS need to be trustworthy, the generated code may not be considered. The authors demonstrate an approach to use this tool in critical environments, arguing that the models created with DSLs are easier to validate, mitigating the risk by a suitably designed transformation and verification architecture. An architecture using LWBs and DSLs was proposed and validated with an industrial case study in the healthcare domain.

Pelechano et al [41] empirical study compares the development of a DSL using Microsoft DSL Tools vs Eclipse Modeling Plug-ins. The experiment counted with 48 persons divided into two groups. Each group used one of the tools to develop a DSL (including code generation). After that, a survey was answered in a way it was possible to make some conclusions. Eclipse Modeling Plug-ins was the preferred because it is more straightforward, robust and stable compared to Microsoft DSL Tools. Furthermore, Eclipse Modeling Plug-ins have an open environment making possible to use and integrate external tools, easier to validate and automatically produce software, contrary to Microsoft DSL Tools that are closed to the integration of external tools.

Hudak [24] states that it is possible that an overhead exists in creating an infrastructure needed to support a DSL. The author proposes a methodology using Modular Domain-Specific Languages using an existing programming language, Haskell. Moreover, implementing DSLs in a modular way facilitates the reuse of software components across many DSL design efforts. This is done using Domain-Specific Embedded Language (DSEL), which allows using many tools designed for Haskell. Furthermore, if several DSELs are designed for different domains but derived from the same host language developers can share them, creating a common core language along with the associated tools. Haskell community have done DSELs for a variety of domains and found it surprisingly straightforward to design them. However, this approach needs to convert the Haskell program to Scheme, partially evaluate the Scheme program, and then translate it back into Haskell. Due to this, the proposed methodology is not so effective as it could be.

2.1.3 Summary

In this section, the usage of DSL in SCS was analysed, also some tools that already exist to support the development of them. It was possible to conclude that DSLs can improve the development in SCS as they are easier to understand due to the language relation with the domain. Moreover, there are existing tools to support the development of DSL that aim to improve the quality and productivity in SCS environments.

2.2 Agile

The issue of how software development should be organized to deliver faster, cheaper, and better software has been debated and discussed for decades. Many solutions for improvement of these points have been suggested with multiple tools, practices, and techniques.

In 2001, 17 persons representing different techniques of development joined and claimed an alternative family of methods, Agile. They wrote the Agile Manifesto [16], which contains an alternative to documentation-driven and heavyweight software, following four main values.

- *individuals and interactions over processes and tools;*
- *working software over comprehensive documentation;*
- *customer collaboration over contract negotiation;*
- *responding to change over following a plan.*

As stated in [22], Agile software development is a process in which a team can manage a project by splitting it up into several stages. At the same time, they have constant collaboration with customers and continuous improvement and interaction at every one of these stages. Agile begins with the clients describing how they want the final product and the problem that it solves. Then the work begins, teams start a process cycle of planning, executing and evaluating. Continuous collaboration and communication between the client and the developer team is the key to success and to make right and informed decisions.

We have available different Agile frameworks (e.g. Scrum, XP, Crystal and ASD).

Agile cycles are recommended to be short with two to four weeks of range during which the team makes the development and adjusts to new information. These cycles are combined with feature planning and dynamic prioritization using, for example, the "backlog" if the framework used in Scrum. The core is that agile approaches plan features, and not tasks, as the priority. Customers understand way better features over developer tasks. At the end of each cycle, the customer reprioritizes the features that we find more important to be developed in the next cycle, sometimes discarding features planned in the beginning.

Agility is about creating content and responding to change. Agile methodologies are not really the practices used. It is the recognition of people as the primary drivers for project success combined with an intense focus on effectiveness, manoeuvrability and communication.

It was possible to conclude that the used approach to language design and the test automation makes a better and more accessible connection between the users of the language and the engineers who connect the language to the test automation frameworks.

2.3 Software Quality

Software quality is an abstract and relative concept. The concept depends on what kind of software systems are in question. For defining the software quality there is a need to examine the software

system in question, who will use it, how will it be used, in which area will it be used, what can be the involved consequences in case of failure, et cetera. As a software developer, we need to guarantee the quality from the consumer perspective and also as the producer perspective. The most fundamental quality to the user is that the software system performs the specified functions in a reliable way. For lots of users, the quality nowadays is the usability and the ease of use. However, there is a lot more that needs to be tested to guarantee quality. For a producer, the most important and fundamental quality is to fulfil the contractual obligations by developing software with the characteristics that were requested. Moreover, there are quality frameworks and standards that should be followed depending on the area of the software with different quality views and expectations, defining quality and attributes like features, characteristics or measures. The use of these frameworks and standards can protect the software system from failure and defects.

There are different standards that can be applied such as ISO 9126 that was updated by ISO 25010 [12]. ISO 9126 has been largely used as a quality model in academic papers. However, many alternative models have been published over the years, improving that model. ISO 25010 explores the application of ISO/EIC 25010:2011 quality models to software systems stating that according to the standards developers should focus on functional suitability, efficiency/performance, compatibility, usability, reliability, security, maintainability and portability (Figure 2.1).

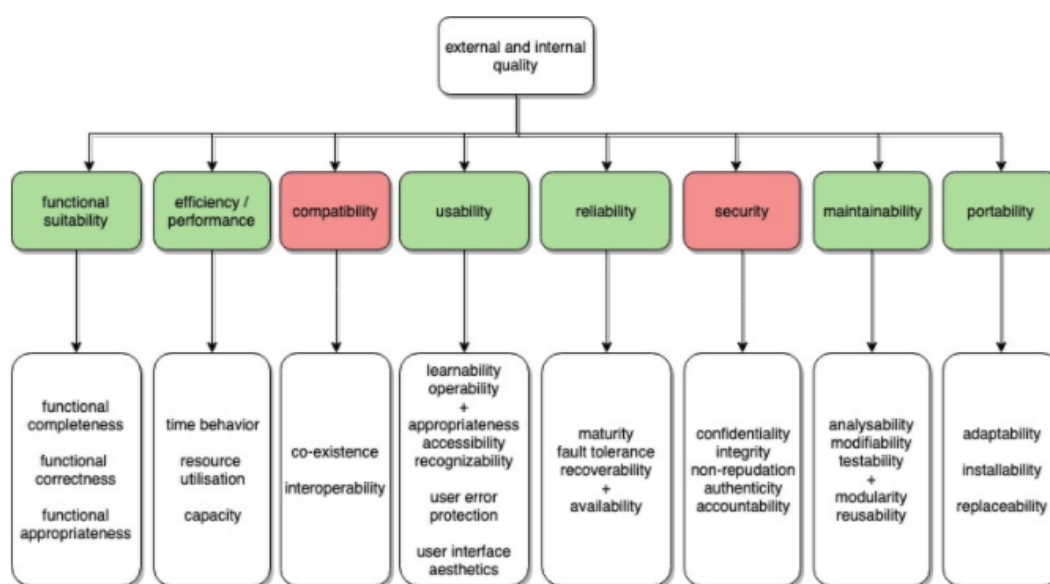


Figure 2.1: ISO/IEC 25010 categorization of software quality requirements [12].

To ensure Software Quality, without defects and damages, the focus activity is Quality Assurance (QA). QA is defined as an activity that tries to ensure that an organization is providing the best possible product or service to the customer. This activity is a process with several steps and methods that can be fitted to specific domains as needed. It is done by ensuring the quality in the engineering process, involving activities related with the implementation of processes, procedures

and standards, making use of preventive techniques with a Testing Environment with carefully selected criteria to find and eliminate specific errors, fault prevention and defect reduction [54]. There are different techniques that can be used to test the software as unit testing, integration testing, system testing, sanity testing, smoke testing, interface testing, regression testing, to mention a few.

IEEE Std. 1012-2012 provides a framework that contains all the extent of Software Development Life Cycle processes, a process that tries to produce software with the highest quality in the shortest time possible time having spent the minimum. It is a well-structured flow divided that help to produce high-quality software, that is tested and ready for production. In IEEE Std. 1012 all the Software Quality Assurance is defined as Verification and Validation (V&V) activities (Figure 2.2).

The objectives of these standards are:

- Help developers introducing quality in the software system during the whole software life cycle;
- Establish a common framework for V&V activities and tasks for software life cycle processes;
- Define V&V processes to provide an objective assessment of the product and processes to demonstrate whether these are correct, complete, accurate and consistent, and also conform with relevant requirements and satisfy their intended use and user needs.

Verification and Validation aims to guarantee the quality of the software since it is being tested in every phase. Regarding safety-critical systems [5], verification and validation is useful, for example, to enable certification of SCS solutions [33].

2.3.1 Verification and Validation

Verification and Validation (V&V) is the process of checking if a software system is according to the specifications and fulfils its intended purposes. The usual definitions for V&V are:

- Verification: Are we building the system right?
- Validation: Are we building the right system?

Sometimes, the differences between Verification and Validation cause confusion, despite it seem trivial to separate the purpose of each task (Figure 2.3). Verification and Validation are related to each other, but they have different objectives.

According to [6], Verification is "the process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase", helping to determine if the software has high quality, meeting the specifications. However, it does not ensure that the system is useful.

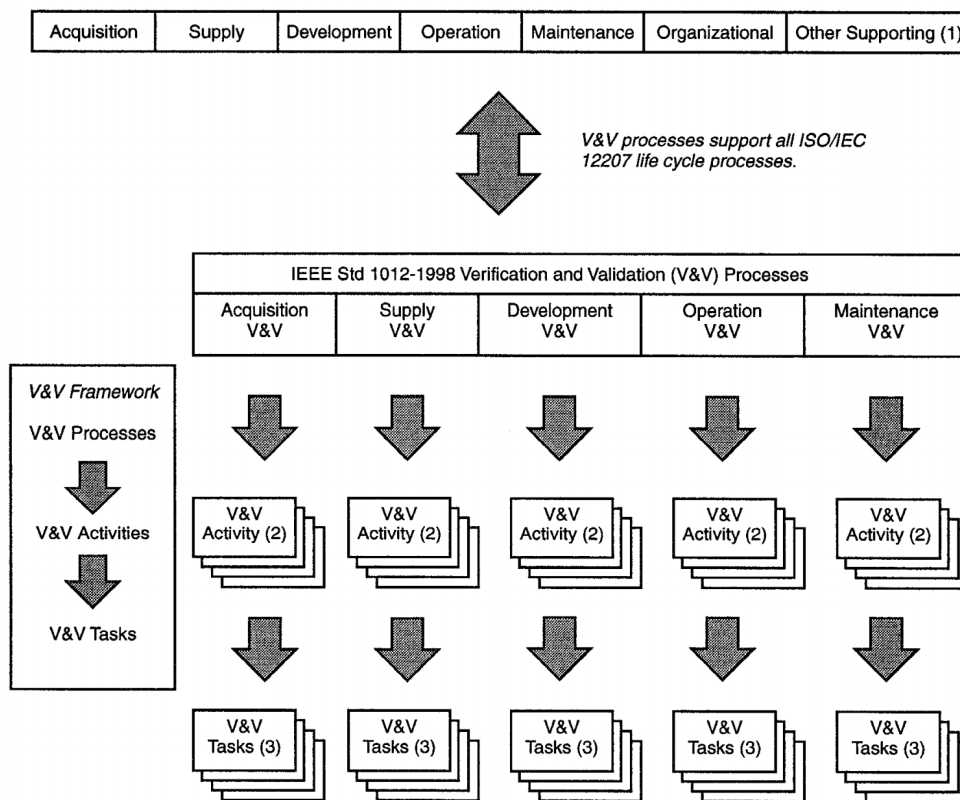


Figure 2.2: Verification & Validation Framework (IEEE 1012).

Validation is the process of evaluating software at the end of certain phases of the software development cycle process to ensure compliance with software requirements. Also, it is related to checking that the created system meets the customer's needs and requirements. [10]

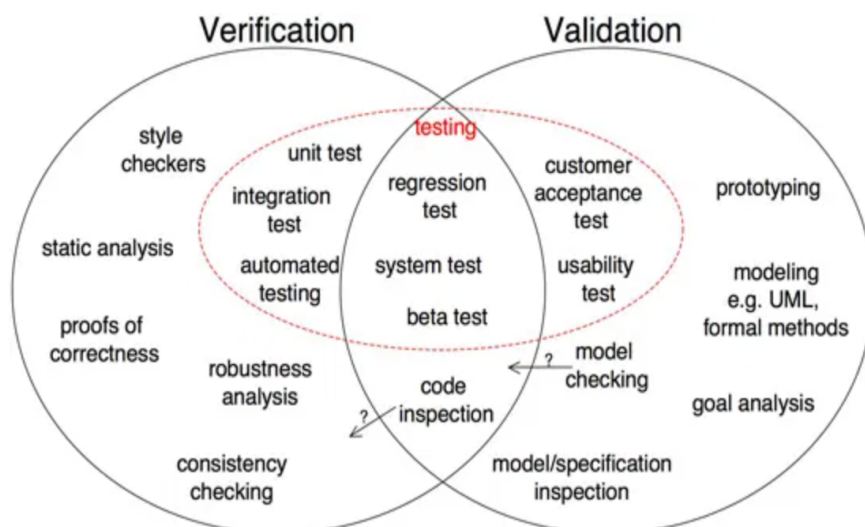


Figure 2.3: Validation VS Verification [10].

2.3.2 V&V in Waterfall

In the traditional phased software lifecycle, like waterfall, validation was only related to the beginning and the end of the development cycle. Verification was used in every phase to ensure the requirements of the previous phase were there (Figure 2.4). However, actually, using Agile methodologies, both verification and validation are used through all the development cycle since the requirements can change at every cycle (Figure 2.5). Agile methods usually start with a set of features described in the form of User Stories, which describe what the customer want and need and how the product will be used, so the validation process is applied to this. Verification can be applied in testing processes and done iteratively since agile methods are iterative and have one to four-week-long development periods where the aim is to make deliverables, also user testing [42]. Figure 2.6 shows the difference between Waterfall and Agile life cycle.

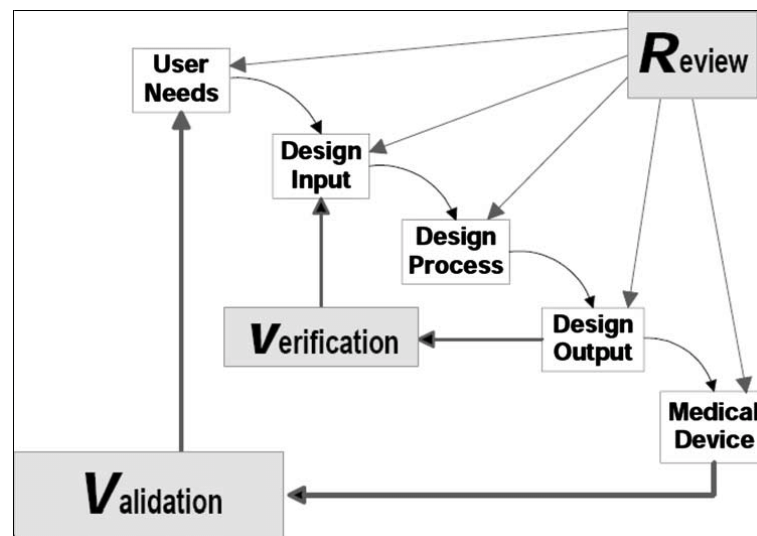


Figure 2.4: Validation & Verification in Waterfall [44].

2.3.3 V&V in Agile

There are several methods that can be used to do V&V (some of them in Figure 2.7). However, this dissertation focus is to improve the quality and agility. As Agile have lots of development cycles, and the requirements can change, on the one hand, if the tests were done manually it would be very time-consuming. On the other hand, automated testing increase the efficiency providing quick responses to requirement changes [9].

Automated testing is different from manual testing, where manual testing consists of a human being responsible for testing the functionality of the software in a way a software user would use it. Automated testing is done through an automation tool. However, despite the time needed for testing being smaller, there is a need for maintaining test scripts coverage with the increase of the developed code.

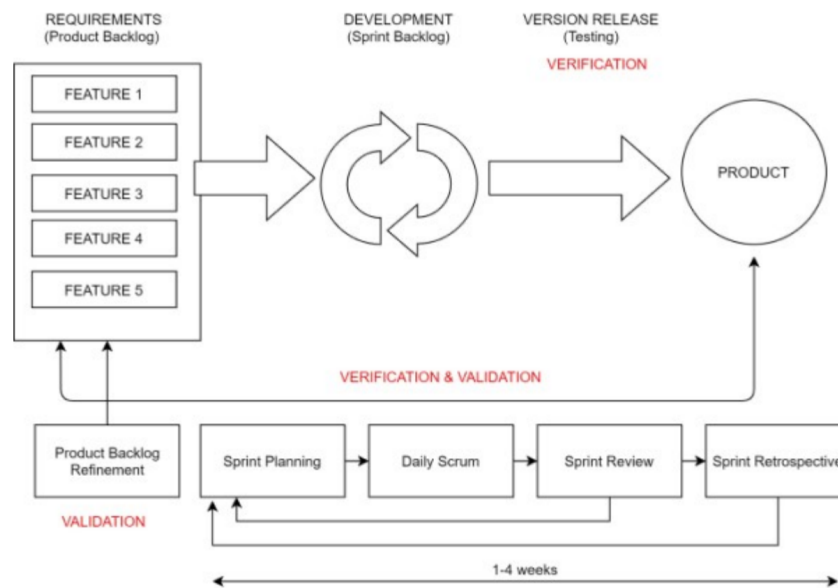


Figure 2.5: Verification & validation processes implemented in each iteration of Agile development [42].

Collins et al [9] state, based on other studies, that "automated software testing is a solution to the problem of maximizing test coverage within the available time and increasing software reliability and quality", resulting in less expensive to test execution since it is possible to reuse the old test on new software versions. Furthermore, testing automation with Test Driven Development (TDD) prevent failures and decrease the development lead-time. However, the studies have a lack of evaluation in scenarios where the development and testing teams perform strategies to automate the tests together inside the agile process.

Huo et al [38] state that Agile methods enter the development phase very quickly, with this, code makes dynamic techniques useful and available early. Furthermore, developers are more close to the QA. The small releases at every cycle brings feedback from the customer, which is positive to the validation of the product also the verification of requirements. There are some QA techniques to the Agile methods that are based on:

- Applying dynamic QA techniques as early as possible, e.g. TDD, acceptance testing;
- Moving more QA responsibility on to the developer, e.g. code inspection in peer/pair programming, refactoring, collective code ownership, coding standards;
- Early product validation, e.g. customer on site, acceptance testing, small release, continuous integration.

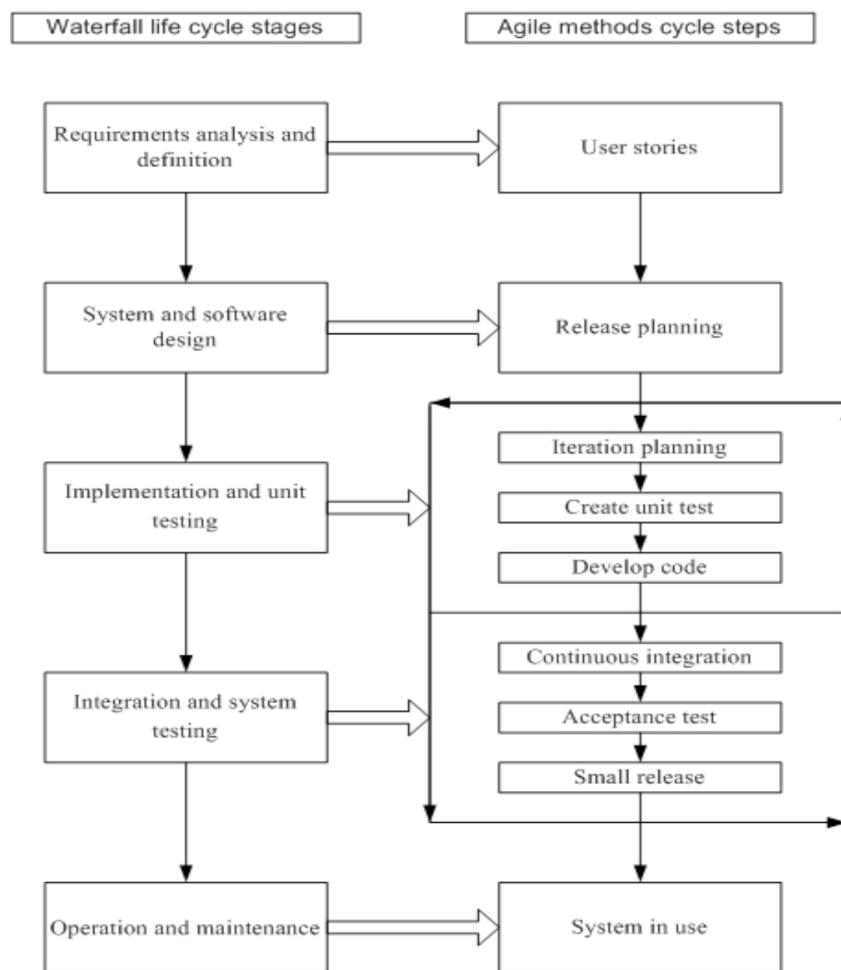


Figure 2.6: Waterfall model vs. Agile methods life cycle [38].

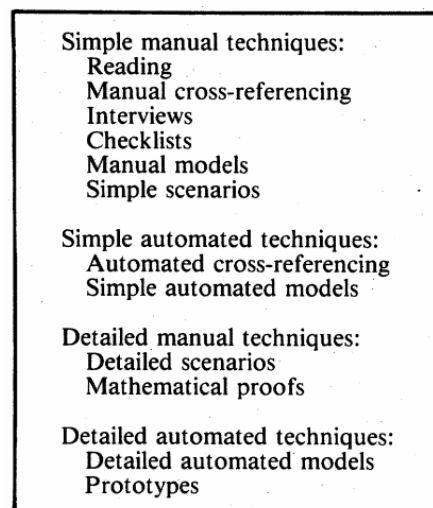


Figure 2.7: Verification & validation techniques [6].

2.4 Behaviour-Driven Development

Test Driven Development (TDD) is a software development practice that consists in creating test cases incrementally before the code implementation. TDD start thinking on how to test from a required functionality, after thinking on it, the automated test cases are written and developers program a code that can pass these created test cases. All the test cases that exist in a certain software system must pass successfully before development to create a new version or to add new functionalities. Furthermore, if a fault or defect is detected in that code unit cases should be done before fixing the code [4].

Behaviour Driven Development (BDD) is a popular technique inside the agile development community, gaining attention in research and practice. It is related to Validation and Verification. North et al [40] introduced it as a response to some of the issues of TDD.

BDD builds upon TDD by using the best habits of TDD practitioners. The focus of BDD is the language and the interactions used in the process of software development. Developers use a native language mixed with an ubiquitous language, a language whose structure comes from a domain model containing terms which will be used to define the behaviour of a system [13]. It gives a path to concentrate on the reasons why the code is created and not in technical details. Furthermore, it diminishes the translations between technical language and the language in which the code is developed.

[51] states that BDD is a combination of different approaches, TDD and automated acceptance testing, having main characteristics as:

- Ubiquitous Language;
- Iterative Decomposition Process;
- Plain Text Description with User Story and Scenario Templates;
- Automated Acceptance Testing with Mapping Rules;
- Readable Behaviour Oriented Specification Code;
- Behaviour Driven at Different Phases.

The acceptance and integration specifications in BDD are written using the steps *Given* that describes the initial context, *When* that describes when a certain event happens and *Then* that is the expected outcome [40] (Figure 2.8 is an example of the use of steps in BDD).

2.4.1 Extending BDD

Felderer et al [21] extended a Behavior Driven Development DSL with business domain concepts. The study compares the use of a DSL for BDD to a modified version of the same DSL that extends with business model concepts, examining whether supporting ubiquitous language features inside a DSL is beneficial over a DSL without it. Despite the study only being done by 22 participants,

```
Example: Multiple Givens
Given one thing
And another thing
And yet another thing
When I open my eyes
Then I should see something
But I shouldn't see something else
```

Figure 2.8: BDD steps example [2]

it was possible to observe that the inclusion of business domain concepts grant significantly faster creation of test case specifications without affecting the document quality significantly.

Tavares et al [53] present a tool stack to follow BDD practices using the Python Language. The tool stack is composed of three different individual tools that compose Pyramid. For now, it is the best-integrated solution to use with Python, since the others are isolated and limited. The main tools of the Pyramid are:

- **Pycukes**, a command-line tool that runs high-level specifications that use *Given*, *When* and *Then* using an External DSL, written in narrative text, which is interesting because any member can write the specifications and pass it to the developers. However, it needs some steps then altering the previous text to make it executable.
- **Ludíbrío**, a tool for Test Doubles, a technique to support the development, that provides a simple way for the configuration of Mocks, simulated objects that mimic the behaviour of real objects in a controlled way, and Stubs, a controllable replacement for an Existing Dependency in the system, using an internal DSL.
- **Should-DSL**, a tool for expectations, using the term "should" to read the expectation in a more fluent way offering a ubiquitous language of the business language.

The advantages of Ludíbrío and Should-DSL on the known alternatives are more features and more appropriate to languages commonly used in Python language and dynamic languages in general. Pyramid and Specloud, a tool that customizes Python unit tests standard library to create the possibility of writing unit specifications and obtain results in BDD style, is still under development.

Cucumber is a testing framework based on Behaviour Driven Development (BDD) that is used to write acceptance tests. It runs files containing executable specifications written in Gherkin. Gherkin is an External DSL that aims to improve the communication between the developers and the stakeholders because of the way of writing the acceptance criteria (in a set of predefined requirements that must be met in order to mark a user story complete) can be automatically used as an acceptance test. An excellent example of how it works is to present usual acceptance criteria, in this case for a credit card "Customers should be prevented from entering invalid credit card details" [57]. In this case, it is not possible to use it to test as it is ambiguous and can be misunderstood by who reads it. However, if it was like this "If a customer enters a credit card number that isn't

exactly 16 digits long, when they try to submit the form, it should be redisplayed with an error message advising them of the correct number of digits" [57], this is not so ambiguous, it is more specific than the previous example, and it is more clear to the developer as well as the stakeholder having a better idea of what is going to be developed. Furthermore, it is almost ready to be used as an acceptance test. This is where Gherkin adds more value by facilitating the comprehension by developers and stakeholders with the primary purpose of being human readable. It provides a way of writing automated tests that are read like documentation in more than forty different languages helping, even more, the connection and understanding between stakeholders and developers [57].

BDD-Security is an open-source security testing framework that uses Behaviour Driven Development concepts to create self-verifying security specifications using Gherkin syntax where security requirements can be tested and automated in a BDD way. It tests web applications and API's from an external point of view without requiring access to the target source code. This means that the application to be tested could be in any language. However, it needs to support HTTP/S [27].

Micallef [37] research discusses five case studies done on industry systems, two using Gherkin-type syntax and the others using a more defined language grammar. The two case studies that use a Gherkin-type syntax were made with a proposed layered approach (Figure 2.9), using it makes it easier to non-technical test analysts read the defined test cases by the product owners. This approach worked well and with some good results. It *"delivered an automated test suite in a few weeks"* without affecting the team productivity. Furthermore, the whole process of creating the DSL for the test suite improved the communication between the team members. However, after some weeks some problems appeared due to Cucumber having some limitations. For example, Given I log in to the system, Given I log on to the system, and Given I log incorrectly, all express the same notion and resulting user actions. Similarly, some team members condensed the sequences of actions into one whilst others used the longer atomic format. They concluded that:

- The used approach works, but it needs a dedicated language owner in the long term;
- There is a need for spending time for the DSL development in the development process;
- If the language grows too much, there is a need for tool support, in a way that it helps the users, the same way a dictionary helps in the natural language.

Härlin [20] made a study to investigate if Gherkin and Cucumber can and should be used for software development projects using agile methodologies. The author states that using Gherkin syntax the acceptance criteria are more precise, giving an overview of what is the functionality for and what it should and should not do. Also, it is stated that Gherkin can be used as a tool to introduce a more collaborative mindset where all the roles contribute from the beginning of the development, avoiding that developers do not know what QA is doing and vice-versa. Moreover, it provides a way of describing specifications abstracting the implementation details in descriptions. This improves communication and collaboration between all participants, being this an essential rule in agile development methodologies.

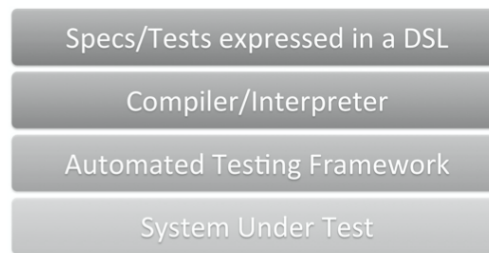


Figure 2.9: Layer approach proposed [37].

Zanin et al [58] explore Gherkin making an extension of it named Aquila. Aquila extends Gherkin automatically creating models from scenarios written in a semi-natural language, providing a way to work with model-based testing. In this technique, the run time behaviour of the software under test is checked against predictions made by a model, allied to BDD, testing it in agile software development. Aquila automatically generates models from the scenarios, making possible to automate the creation of test scripts exclusively based on the specified requirement, also a set of inputs and outputs expected for the system, giving information of what the system should and should not do, increasing the testing efficiency. Figure 2.10 represents the usage of Aquila, showing the steps that can be done using it. In order to test if Aquila works, it was tested in a real environment where it worked as the authors wanted.

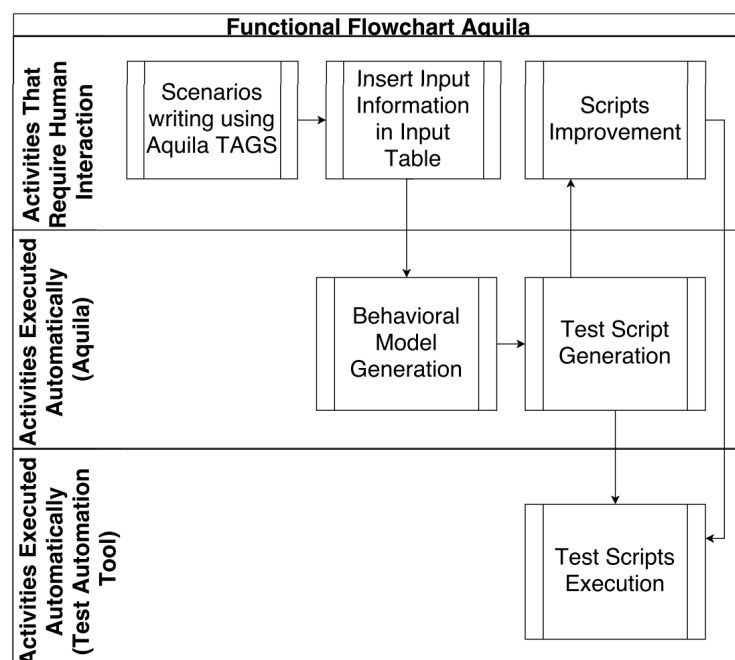


Figure 2.10: Aquila Functional Flowchart [58].

Soeken et al [50] state that in BDD, the acceptance tests provide the starting point to the development and are the essential communication between the software developers and the stakeholders, the authors introduce a way of automatizing the process of mapping the sentences to actual source code by using natural language processing techniques, over the scenarios described in the acceptance tests, that allow a semi-automatic transformation from the tests to stubs. The proposed methodology consists of a design flow where the user enters into a dialogue with the computer. The program processes sentence by sentence and suggests a code block for each of it where the user can accept or refuse the suggestions made (Figure 2.11). Furthermore, the suggestions can be revised to train the computer program. To implement this technique, it is used Cucumber and a Stanford Natural Language Processing software. The method was tested in a case study. Although the method worked, it worked partially as it was unable to classify some words. In any case, they managed to create a reasonable basis for future work.

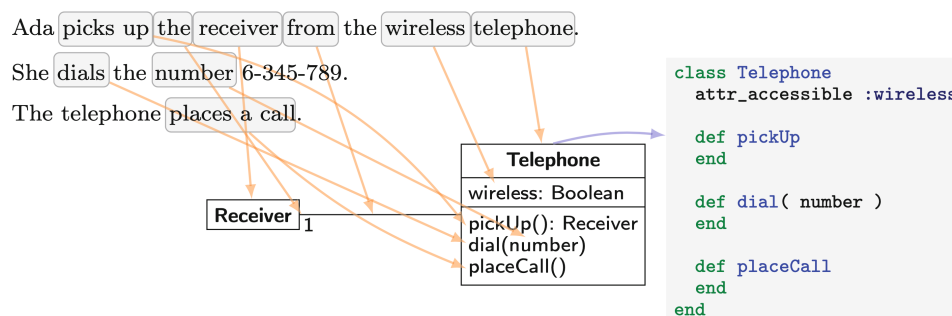


Figure 2.11: Aquila Functional Flowchart [50].

2.4.2 Cucumber and Gherkin

Cucumber is an open-source tool that supports Behaviour-Driven Development. Cucumber differs from other tools because it was explicitly designed to ensure the acceptance tests can easily be read and written by anyone. The easy readability of Cucumber acceptance tests draws business stakeholders into the process, helping developers explore and understand their requirements. The tests are specified as examples of the way a system should behave in particular scenarios, written in an ubiquitous language. The acceptance tests are not just tests, they are also executable specifications. Furthermore, all the team members can read and be involved easily. It was originally created to be used with Ruby. However, it has been translated, and it is now supported in 26 different development environments.

Cucumber reads specifications from plain-language text files called features, examines them for scenarios to test, and runs the scenarios against the system. Each scenario is a list of steps for Cucumber to work through. So that Cucumber can understand these feature files (Figure 2.13), they must follow some basic syntax rules. The name for this set of rules is Gherkin. Gherkin supports more than forty different spoken languages that can be used to write that rules. Along with the features, it is needed a set of Step definitions, which map the business-readable language

of each step into code (using the development environment language) to carry out whatever action is being described by the step. Figure 2.12 illustrates the structure of Cucumber.

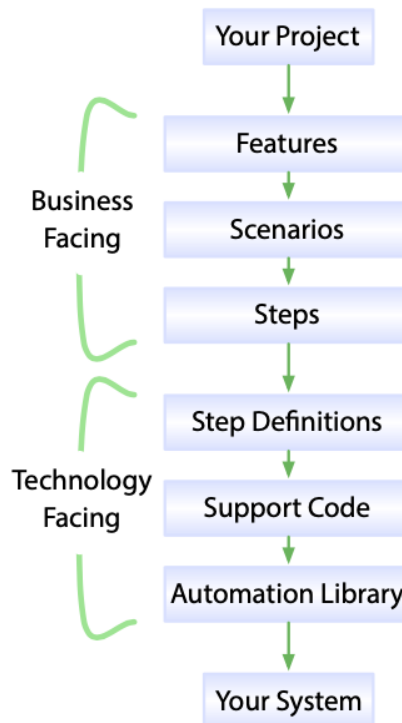


Figure 2.12: Structure from features down to automation library. [45]

Cucumber tools and techniques

Gherkin uses a set of keywords to give structure and meaning to the executable specifications. Each line that isn't a blank line has to start with a Gherkin *keyword*, the only exceptions to this are the feature and scenario descriptions that can be written after a line that contains a Feature or a Scenario *keyword*.

Feature

The purpose of the **Feature** *keyword* is to provide a high-level description of the software feature, also group all the possible scenarios related with that feature. A feature file starts with this *keyword*. This *keyword* is followed by **:** and a short text which describes the feature (Figure 2.13).

Descriptions

There is an optional description that can be made. However, they do not have a meaning to Cucumber. The only purpose of this description is to add important aspects, for example, of the feature to explain what it should do. The description can be also placed underneath [Scenario/Example](#), [Background](#), [Scenario outline](#) and [Rule](#). Everything can be written, excluding a *keyword*.

Feature: Feedback when entering invalid credit card details

In user testing we've seen a lot of people who made mistakes entering their credit card. We need to be as helpful as possible here to avoid losing users at this crucial stage of the transaction.

Background:

Given I have chosen some items to buy
And I am about to enter my credit card details

Scenario: Credit card number too short

When I enter a card number that's only 15 digits long
And all the other details are correct
And I submit the form
Then the form should be redisplayed
And I should see a message advising me of the correct number of digits

Figure 2.13: Feature file example [45].

Scenario/Example

This *keyword* can be used as **Scenario** or **Example**. It is a concrete example of what the business rule is. In addition to being a specification, also documentation, a **Scenario** is also a test. They are executable specifications of the system. **Scenario** is composed of steps that describe the context, the event and the expected outcome of that specification. The keywords for the steps are **Given**, **When**, **Then**, **And**, or **But**. Any sentence written after these *keywords* can have arguments that are passed to [Step Definition](#).

- **Given** steps are used to describe the initial context of that rule;
- **When** steps are used to describe an event, or action that happened after the context used in given, like an interaction with the user;
- **Then** steps are used to describe the expected result from the previous interaction;
- **And** and **But** steps are used to add more information to **Given**, **When** and **Then** instead of repeating that step (Example on [Figure 2.14](#) and [Figure 2.15](#)).

Example: Multiple Givens

```
Given one thing
Given another thing
Given yet another thing
When I open my eyes
Then I should see something
Then I shouldn't see something else
```

Figure 2.14: Repeated Given. [2]

```

Example: Multiple Givens
  Given one thing
  And another thing
  And yet another thing
  When I open my eyes
  Then I should see something
  But I shouldn't see something else

```

Figure 2.15: Use of And and But. [2]

Scenario outline

The **Scenario outline** is a *keyword* that can be used to run the same Scenario/Example multiple times but using different combinations of values. Scenario outlines allow use < > delimited parameters containing the name on an example table (Example figure 2.16).

```

Scenario Outline: eating
  Given there are <start> cucumbers
  When I eat <eat> cucumbers
  Then I should have <left> cucumbers

Examples:
  | start | eat | left |
  |   12 |   5 |   7 |
  |   20 |   5 |  15 |

```

Figure 2.16: Scenario outline. [2]

Background

Sometimes the developer have cases in a Feature where some **Given** steps are repeated. To avoid this repetition **Background** could be used. A **Background** is placed before the first Scenario/Example that will use it, the next ones need to be at the same level of indentation to use the same **Background**. (Figure 2.17)

Rule

The *keyword* **Rule** is used to represent one business rule that should be implemented, providing more information on a certain feature. It is used to group together several scenarios that belong to this business rule. (Figure 2.18)

```
Feature: Multiple site support
  Only blog owners can post to a blog, except administrators,
  who can post to all blogs.

Background:
  Given a global administrator named "Greg"
  And a blog named "Greg's anti-tax rants"
  And a customer named "Dr. Bill"
  And a blog named "Expensive Therapy" owned by "Dr. Bill"

Scenario: Dr. Bill posts to his own blog
  Given I am logged in as Dr. Bill
  When I try to post to "Expensive Therapy"
  Then I should see "Your article was published."

Scenario: Dr. Bill tries to post to somebody else's blog, and fails
  Given I am logged in as Dr. Bill
  When I try to post to "Greg's anti-tax rants"
  Then I should see "Hey! That's not your blog!"
```

Figure 2.17: Background Example. [2]

Step Definition

A Step Definition is a method (in Java, Ruby, Kotlin, etc.) with an expression that links it to one or more Gherkin steps. When Cucumber executes a Gherkin step, it looks for a matching step definition to execute. The match between a Gherkin step and the expression could be done by using a Regular Expression or a Cucumber Expression (Figure 2.19). A Cucumber Expression could have arguments, called **Step Arguments**. These arguments are extracted from the expression and passed to the method. When cucumber finds a step definition it will execute it, the step definition could be marked as **Success**, **Undefined**, **Pending**, **Failed Steps**, **Skipped** and **Ambiguous**. This way the developer could test by knowing the status of every step.

Tag

Tags can be used to organise either features or scenarios. When you use tags, you can opt to run only a subset of scenarios that has that tag, also restrict hooks to a subset of scenarios. They can be placed over Feature, Scenario, Scenario Outline, however it is not possible to use it above Background.

```
Feature: Highlander

  Rule: There can be only One

    Example: Only One -- More than one alive
      Given there are 3 ninjas
      And there are more than one ninja alive
      When 2 ninjas meet, they will fight
      Then one ninja dies (but not me)
      And there is one ninja less alive

    Example: Only One -- One alive
      Given there is only 1 ninja alive
      Then he (or she) will live forever ;- )

  Rule: There can be Two (in some cases)

    Example: Two -- Dead and Reborn as Phoenix
```

Figure 2.18: Rule Example. [2]

Hooks

Hooks are blocks of code that can run at various points of the Cucumber execution cycle. They are typically used for setup and tear-down of the environment before and after each scenario.

- **Scenario hooks** run in every scenario, they can run before the first step of each Scenario or after the last step of each Scenario. (Figure 2.20)
- **Step hooks** run before or after a specific step. (Figure 2.21)
- **Conditional hooks** can run based on a Tag of the scenario, also before or after the scenario. (Figure 2.22)

Preprocessor

Preprocessing can reduce redundancy in feature files. The preprocessor transforms files in an input folder to files in an output folder. For Cucumber, these would be feature files. Cucumber would then run the feature files in the output folder. Preprocessing features allows to define symbols with replacement string, define symbols as a result of a calculation, create local functions that can be used in calculations and include common files in multiple feature files.

```
@Given("I have {int} cukes in my belly")
public void i_have_n_cukes_in_my_belly(int cukes) {
}
```

Figure 2.19: Example of a Cucumber Expression with a Java method connected to it, known as Step Definition. [2]

```
@Before
public void doSomethingBefore() {
}
```

Figure 2.20: Scenario Hook Example. [2]

```
@BeforeStep
public void doSomethingBeforeStep(Scenario scenario){
}
```

Figure 2.21: Step Hook Example. [2]

```
@After("@browser and not @headless")
public void doSomethingAfter(Scenario scenario){
}
```

Figure 2.22: Conditional Hook Example. [2]

2.4.3 Summary

In this section, Agile methodologies and methods that aim to use these methodologies in software development were explored. It was possible to conclude that:

- Agile
 - Emphasizes continuous collaboration and communication between the client and the developer team;
 - Agile cycles are short and focus on planning features over tasks.
- TDD
 - TDD consists in creating test cases incrementally before the code implementation;
 - Developers only write the code after the test cases implementation;
- BDD
 - BDD is a response to some issues of TDD;

- The focus is the language and the interactions used in the process of software development;
 - A native language mixed with a ubiquitous language is used;
 - Using it improves communication and collaboration between all participants, being this an essential rule in agile development methodologies;
 - The acceptance tests provide the starting point to the development and are the essential communication between the software developers and the stakeholders;
- Gherkin
 - Gherkin is known for being simple, introducing a more collaborative mindset from the beginning of the development;
 - Can be expanded and modified;
 - It is Open Source;
 - Provides a way of describing specifications abstracting the implementation details in descriptions;
 - Provides a way of writing automated tests that are read like documentation in more than forty different languages helping
 - It can be implemented with other tools to get other functions (for example, semi-automatic creation of code blocks from the automated tests using a Language Processing Software).

This chapter was essential to develop our proposed solution, as it gave several insights on the tools that can be used. Furthermore, it increased our knowledge in some of the areas needed for this dissertation.

Chapter 3

Problem Statement

In this chapter, we describe the problems related to the use of Domain-Specific Languages applied to Safety-Critical Systems and the implementation of SCS using agile methodologies, specifically, the current issues found during our research that will influence this dissertation.

3.1 Current Issues

The use of agile methodologies in safety-critical systems development is growing, but it is not used to its full potential. Despite existing some techniques, there is not yet a well established agile way for verification and validation of the SCS implemented in real-world environments, neither tools to support it. Also, as the standards for SCS are very extensive, it takes much time to test all the requirements.

3.2 Assumptions

We believe this work is relevant because it explores the use of Agile methods in SCS development. In order to address the issues previous listed, we assume that:

- BDD may improve quality and agility of SCS development, specifically in the Verification & Validation;
- Cucumber and Gherkin are one of the most suitable tools to implement BDD;
- It is possible to do a Test Suite through the use of Cucumber that can be reused in the development of different SCS that share the same standards.

3.3 Hypothesis

There are ways of improving the agility and quality in SCS development process. In this development process, some requirements come from standards, regulations, and guidelines that are not

common knowledge of developers' teams being more than specialists in the field. This dissertation is based on a hypothesis that serves as the base for the work developed through this dissertation. The respective hypothesis is that these requirements can be verified through test suites in cucumber that developers can reuse, improving the development's quality and agility.

3.4 Research Questions

Our research problem is focused on the following questions:

- Is it easy to implement the verification of the requirements recurring to BDD?
- Is it easy to use BDD to test the developed software compliance?
- Will BDD give good traceability of the error to the Standard/requirement?

Considering these questions, this work aims to find a way, namely practices and tools, to reduce the time needed to verify if the system complies with the needed requirements.

3.5 Methodology

This dissertation aims to provide insights on how to build a test suite through the use of BDD that can improve the quality and agility of SCS development. We believe that using BDD should improve the quality of SCS development having the system easily tested according to the standards. Furthermore, it is easier to bring the knowledge of the standards experts to the system developers through the use of BDD. In order to build our solution, the work is divided in three parts:

State of Art. This part of the dissertation, introduces and explores the most important concepts for this work. The state of art explores *Domain Specific Languages, Software Quality, Agile methodologies*, more in depth *Behaviour-Driven Development*. The research made on this topics gave important insights on which tools could be used in our implementation. We could not find implementations in the academic environment that used BDD to improve SCS development.

Case Studies. The implementation of the Test Suite in the case studies will be done using action research and design science research, as the suite was implemented through the case studies, the implementation was evaluated and re-planned in order to improve and adapt the suite to be reused in different systems with the less effort possible from the developers. Making the path between standard experts and developers shorter.

Evaluation. This part aims to validate the solution proposed in this dissertation. This validation is divided into two parts. Each of them evaluates whether our proposed solution improved or not agility and quality. The overall validation of the thesis is achieved through the partial validation of these two parts.

Chapter 4

Proposed Solution

This dissertation aims to provide insights on how to do verification and validation on Safety-Critical Systems, improving the quality of the software and also its development agility. After the state-of-art analysis, we concluded that BDD could potentially improve the development of Safety-Critical Systems. Furthermore, it seems a very suitable approach that could have more benefits than only testing, for instance, to help developers create the code skeleton from the created acceptance tests.

The proposed solution is to use Gherkin, creating a test suite with the objective of translate specific Standards into a Gherkin DSL that is capable of verifying and validating the SCS of the referred domain. The implementation of this test suite was done and improved over the four mentioned case studies.

4.1 Case Studies Overview

The proposed solution was refined along three case studies. In these case studies, the objective is to assess which and what for Gherkin features should be used to help to improve the quality and agility of SCS development.

In the development process, there are functional and non-functional requirements. A functional requirement (Figure 4.1) define the basic system behaviour. Functional requirements usually define if/then behaviours and include calculations, data input, and business processes.



Figure 4.1: Functional Requirement. [1]

A non-functional requirement is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviors. Sum up, a non-functional requirement

describes how the system works, while functional requirements describe what the system should do (Figure 4.2).

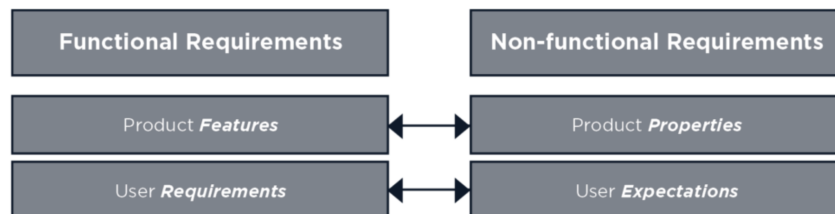


Figure 4.2: Functional Requirements vs Non-Functional Requirements. [1]

Some Standards impose requirements. Our approach focuses on ensuring that the system complies with some of those requirements, in specific, functional requirements. This solution objective is to make it possible to bring the knowledge of standard experts to developers in the simplest possible way through the use of BDD. Furthermore, our objective is to create a tool that makes possible to ensure compliance effortlessly and encourage SCS developers to use BDD in the development process.

In this dissertation we focus on three main roles that are mentioned among the three case studies:

- **Domain Expert** - This role consists in someone whose knowledge about a certain domain is astonishingly good, also the knowledge about the implicit Standards of that domain.
- **Tester** - The objective of this role is to test the developed code extensively. Every features and every piece of code must be tested in SCS. Furthermore, they have to assure that the code follow the Standards that the Standards Experts find mandatory.
- **Developer** - The objective of this role is to develop the SCS. The developed code made by the developer must pass the tests made by the Tester to assure compliance with the Standards and every other feature testing in order to get the system certified to be used in real life.

4.2 The Cucumber for Java Book - ATM

This case study used an ATM system from "The Cucumber for Java Book" adapted to implement our tests (Figure 4.3). The objective was to understand how Cucumber and Gherkin may work in our context, which Gherkin functionalities should or should not be used, how they should be used, and how to implement something that makes a part of the system compliant Standard/ISO recurring to tests.

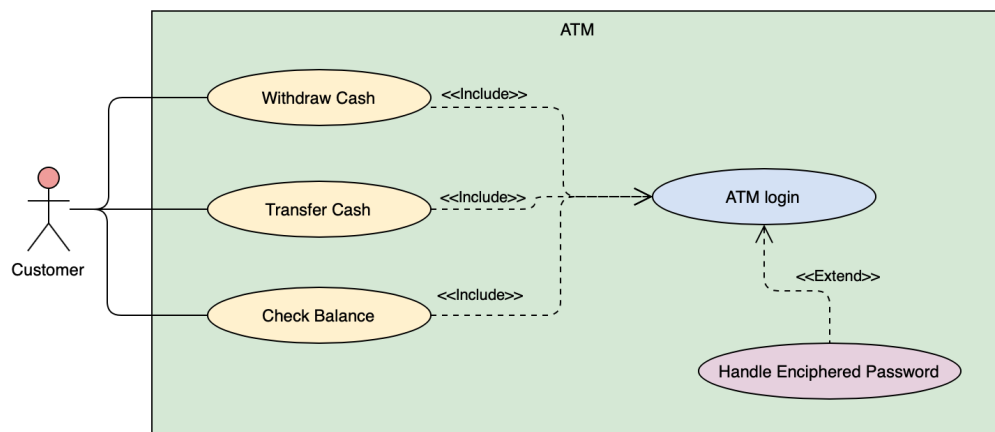


Figure 4.3: Use Case Diagram - Adapted Cucumber ATM.

Motivation

The ATM was chosen for this case study as it was a detailed example of a system that was present through the entire Cucumber for Java Book giving better insights of how Cucumber works, also helping to analyse the expected behaviour of the system.

Problem

At the beginning of this case study, our main question was whether or not we needed to extend Gherkin to support our objectives intuitively and simply so the system developers can easily use it without a significant effort or if we could use a standard Gherkin functionality to do it. During the development of the system we analyzed **tags**, **hooks**, **And** and the **preprocessor**. We concluded that tags and hooks could help the testing process improve the agility as they help test only some parts of the system quickly and execute blocks of code before or after the respective tests, respectively. However, they are not suitable to be used in our solution. The **preprocessor** can simplify some cases reducing the redundancy in feature files. Furthermore, there was no need to extend Gherkin as the default features could meet this dissertation objective.

Solution

We ended up using a standard Gherkin functionality. A step "And compliant with <standard>". This step tests if the part of the system being tested is compliant with a selected part of a standard. This seemed to be a right approach since it is intuitive to include in the BDD tests. Also, it provides a way to check compliance with the needed standard allowing to reuse the knowledge about the standard.

Implementation

Our developed solution for this case study was done in three parts:

- Adapt the existing ATM system in the book, from joining different parts of code that are present through the book to the configuration of the development environment;
- Start the development of our test suite, testing a generic case without a compliant standard to assure that the system was running as supposed (Figure 4.4), also test the different Gherkin functionalities in order to understand if they can be suitable to develop our solution. Furthermore, understand if there are cases where some functionalities would be better than others;
- Search for a Standard that could be used to implement the functionality that seemed most suitable for our solution, determined in the step below. We adapted ISO 9564-2, which specifies approved algorithms for the encipherment of Personal Identification Numbers. Using the step "And compliant with ISO 9564-2", our objective was to test if the system is compliant with the mentioned ISO. The objective was to ensure that the step only passed with success if the PIN met the needed encipherment which is defined by us (Figure 4.5). Without this step, the Scenario passed if the PIN was not enciphered according to our adapted ISO.

Feature: Cash Withdrawal	Test Results	240 ms
Scenario: Successful withdrawal from an account in credit	Feature: Cash Withdrawal	240 ms
Given my account has been credited \$100.00	Scenario: Successful withdrawal from an account in credit	236 ms
When I request \$20	Given my account has been credited \$100.00	228 ms
Then \$20 should be dispensed	When I request \$20	2 ms
And the balance of my account should be \$80.00	Then \$20 should be dispensed	4 ms
Scenario: Unsuccessful withdrawal due to technical fault	And the balance of my account should be \$80.00	1 ms
Given my account is in credit	Scenario: Unsuccessful withdrawal due to technical fault	5 ms
But the cash slot has developed a fault	Given my account is in credit	3 ms
When I request some of my money	But the cash slot has developed a fault	1 ms
Then I should see an out-of-order message	When I request some of my money	1 ms
And \$0 should be dispensed	Then I should see an out-of-order message	0 ms
And the balance of my account should be unchanged	And \$0 should be dispensed	0 ms
	And the balance of my account should be unchanged	0 ms

Figure 4.4: Generic ATM test.

Feature: PIN	Test Results	163 ms
Scenario: PIN enciphering follow the approved algorithms	Feature: PIN	163 ms
Given I enter my enciphered PIN	Scenario: PIN enciphering follow the approved algorithms	163 ms
And compliant with "ISO 9564-2"	Given I enter my enciphered PIN	153 ms
When access my account	And compliant with "ISO 9564-2"	7 ms
Then I should be able to see my balance	When access my account	1 ms
	Then I should be able to see my balance	2 ms

Figure 4.5: ATM test with Compliance Check.

ISO class implements the code related to the compliance verification. The **check** function receives the Standard from the step definition and checks the compliance with that ISO. In this case, we check if the pin (that is hardcoded in the ATM construct and enciphered previously) can be deciphered with the respective key and be equal to the one in the constructor. If the test suite does not have the Standard to test a message informs that it is unavailable.

```
1 public class ISO {
2     private String pin;
3
4     public ISO(Pin pin) {this.pin=pin.getPIN();}
5
6     public boolean check(String standard){
7         switch(standard) {
8             case "ISO 9564-2":
9                 Assert.assertEquals("Pin is not encyphered according to ISO 9564-2", Encrypt.
10 decrypt(pin,"maytheforcebewithyou"), "1234");
11                 break;
12             default:
13                 System.out.println("No ISO Found");
14                 return false;
15         }
16         return true;
17     }
18 }
```

Listing 4.1: ISO9564_2.java

Conclusions

This case study was important as it gave big amount of insights on how Cucumber works and how we can take advantage of it to develop our solution. Furthermore, it made us feel that there was a need to test our approach on an existing ATM made by an external source to ensure that the selected approach is compatible in a different environment and if it is possible to be adapted to another system easily. Furthermore, it allowed us to be more accommodated to the tools to be used in this dissertation's proposed solution. It was possible to find a simple solution to verify compliance with a standard that significantly facilitates this feature's implementation through the use of BDD.

4.3 Russell C. Bjork - ATM with GUI

In this case study, the system used to test was an ATM created by Russel C. Bjork [46], with substantially more features, also a graphic interface (Figure 4.6).

Motivation

The objective of this case study was to use our testing suite on a system developed by another person to support that it can be adapted to other systems different from the one used in the first case study.

Problem

In the previous case study our test suite had a dependency from the ATM system. After having the test suite running on this ATM system, we felt a need to make the test suite independent from the system; this way, it could test any ATM system reusing it.

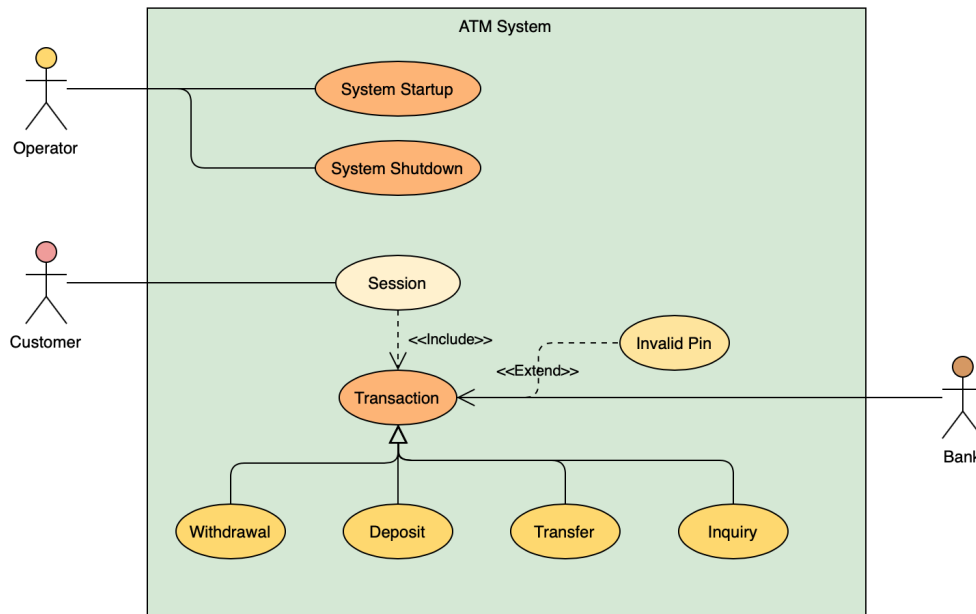


Figure 4.6: Use Case Diagram - ATM with GUI.

There was a need to separate the test suite from the system assuring that it facilitates the system developers to integrate the test suite in their system development. In this case the test was based on standard "ISO 4217", that defines alpha codes and numeric codes for the representation of currencies and provides information about the relationships between individual currencies and their minor units, this standard is a dependency of "ISO 8583" which is related with transaction originated messages. The test guarantees that the message sent has the right currency code.

ISO Standards 8583 and 4217

- ISO 8583 - "specifies a common interface by which financial transaction card-originated messages can be interchanged between acquirers and card issuers. It specifies message structure, format and content, data elements and values for data elements. The method by which settlement takes place is not within the scope of this part of ISO 8583." [29]
- ISO 4217 - "establishes internationally recognized codes for the representation of currencies that enable clarity and reduce errors. Currencies are represented both numerically and alphabetically, using either three digits or three letters. Some of the alphabetic codes for major currencies are familiar, such as "EUR" for Euros." [28]

Figures 4.7 and 4.8 show respectively the session sequence diagram that contains the transaction and the transaction diagram. The standard compliance test was applied in the message sent in the transaction.

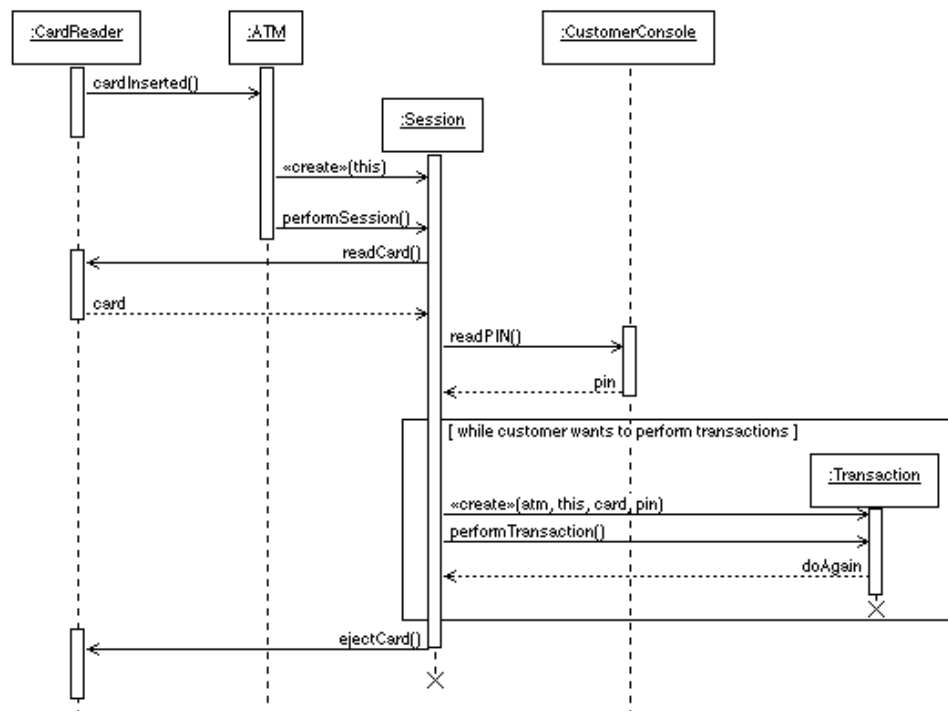


Figure 4.7: Session Sequence Diagram. [46]

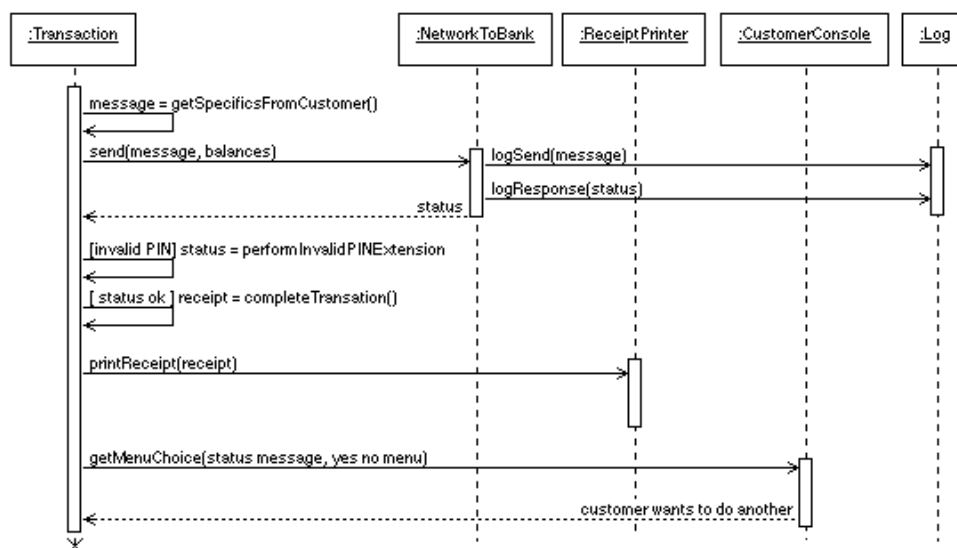


Figure 4.8: Transaction Sequence Diagram. [46]

Solution

However, the compliance test was depending on the system. To provide a better way of reusing the created test suite, three different approaches were explored to make the test suite independent

of the system: mock objects, adapters, and facades.

Mock Objects

A technique commonly used in crafting automated unit tests. An object under test may have dependencies on other (complex) objects. To isolate the object's behaviour, it is needed to replace the other objects by mocks that simulate the real objects' behaviour. It is useful if the real objects are impractical to incorporate into the unit test. In short, mocking consists of creating objects that simulate the behaviour of real objects. However, it did not work for us. A mock object knows in advance what is supposed to happen during the test (e.g. which of its methods calls will be invoked) and the mock object knows how it is supposed to react (e.g. what return value to provide). The mock will indicate whether what happens differs from what is supposed to happen. Even though the system developers can use them to do some testing and improve the agility of the system development, it cannot help improving and facilitating the implementation of our suite in a system to be tested. However, we recommend the system developers to use them in their implementation if they need to simulate an object under construction to improve the agility of the development. Figure 4.9 is an example of a mock object, in this example a widely used framework in TDD and BDD techniques is used, Mockito. This mock object is a simple example where a list is mocked in Java.

```
//Mock creation
List mockedList = mock(List.class);

//Using mock object
mockedList.add("one");
mockedList.clear();

//Verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Figure 4.9: Mock object example using mockito - List class mock. [39]

Adapters

It is a structural design pattern, a typical solution to commonly occurring problems in software design, they are like pre-made blueprints that can be customized to solve a recurring design problem in your code, which allows objects with incompatible interfaces to collaborate. An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object is not even aware of the adapter. For example, it can wrap an object that operates in meters and kilometres with an adapter that converts all of the data to imperial units such as feet and miles. Adapters can not only convert data into various formats but can also help objects with different interfaces collaborate [49]. Behind the scenes, the adapter gets an interface, compatible with one of the existing objects. Using this interface, the existing object can safely call

the adapter's methods. Then, upon receiving a call, the adapter passes the request to the second object, but in a format and order that the second object expects (Figure 4.10 is an example of an adapter use). Adapters could be useful to abstract our test suite from the system being developed. It allows objects with incompatible interfaces to collaborate, making it possible to connect the test suite and the developed code to pass the needed data from the system to the test suite.

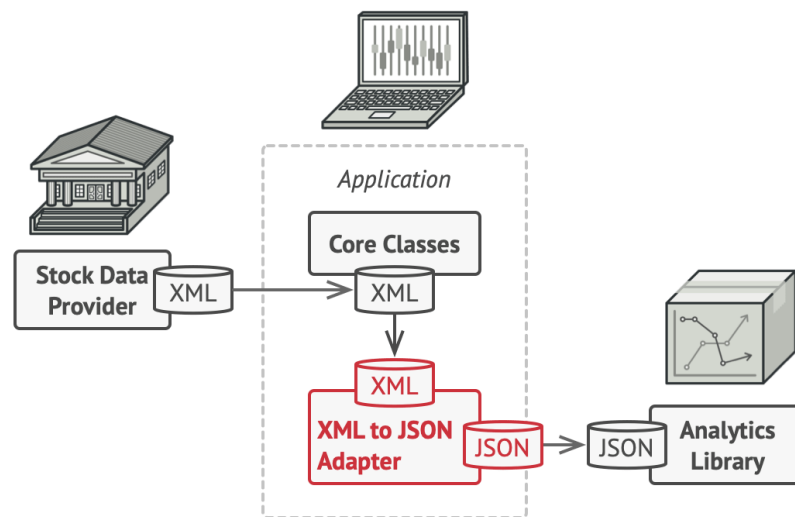


Figure 4.10: Adapter example. [49]

Facades

It is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. A facade is a class that provides a simple interface to a complex subsystem which contains lots of moving parts (Example of a "Real-World" analogy to a Facade 4.11). A facade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients care about which in our case are the functionalities that need to be compliant with a specific standard [49]. The use of a facade could be useful to abstract our test suite from the system being developed. It is a way of simplifying the connection between the test suite and the system. By implementing the test suite facade in the system, it is possible to request the developers to pass the needed data to the suite creating the functions present in the facade.

Implementation

We opted to use facades. With this approach, the system developer needs to call the facade and implement the classes to provide the test suite with the needed data. The drawback of this approach

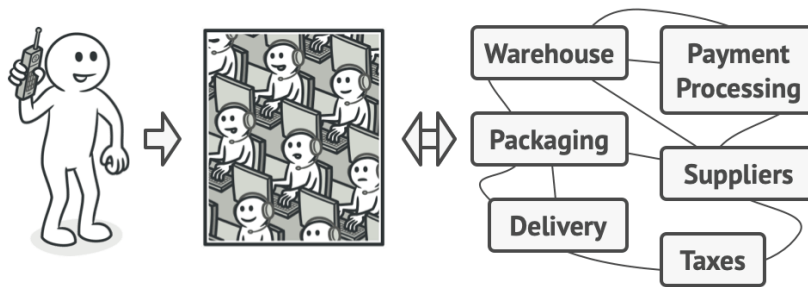


Figure 4.11: "Real-World" Facade analogy. [49]

is related to implementing the facade's interface methods in the system to guarantee that the test suite works correctly. However, in most cases, these methods needed to be there one way or another to implement the system. Adapters could also be used. However, an adapter usually wraps just one object, while facade works with an entire subsystem of objects, making facades more suitable.

ATM Facade+ Test Suite

The test suite is mainly implemented in four files: `standard.feature`, `ISO4217Steps.java`, `System-Facade.java` and `ISO.java`.

Standard is a feature file that contains a created scenario for this case study where a compliance with ISO 4217 is checked.

```

1 Feature: Cash Withdrawal with Compliant
2   Background: The country where the ATM is used
3     Given the ATM is designed to be used in "United States"
4     Then the currency code is USD
5
6   Scenario: Successful withdrawal from an account in credit
7     Given my account has 100.00
8     When I request 20
9     Then 20 should be dispensed
10    And the balance of my account should be 80.00
11    Then a message should be sent
12    And compliant with "ISO 4217"

```

Listing 4.2: Standard.feature

ISO4217Steps is the class that contains the Step Definition related with the compliance test. The objective of this step is to call a function that checks the compliance of the system with the selected standard.

```

1 package atmthesis;
2
3 import io.cucumber.java.en.Given;
4 import io.cucumber.java.en.Then;
5 import iso.ISO;
6 import support.Construct;
7 import org.junit.Assert;
8
9 public class ISOSteps{
10    @And("compliant with {string}")

```

```

11 public void compliant_with_ISO(String standard) throws Exception {
12     Assert.assertTrue(ISO.getInstance().check(standard));
13 }
14 }

```

Listing 4.3: ISO4217Steps.java

SystemFacade implements a Facade through an interface, a completely "abstract class" that is used to group related methods with empty bodies. The function **getCountryCurrencyCode()** is a function that must be in the system to return the country code and **getCurrencyCodeInMessage()** is a function that returns the code that was sent in a message. These two functions allow the test suite to verify if the system is compliant with the adapted ISO that was used in this case study.

```

1 package iso;
2
3 public interface SystemFacade {
4     public String getCountryCurrencyCode();
5     public String getCurrencyCodeInMessage();
6 }

```

Listing 4.4: SystemFacade.java

ISO class implements the code related with the compliance verification, the hashtable contains some examples of the currency codes to be checked in **checkHash** function.

- **check** function receives the standard from the step definition present in **ISOSteps** and checks the compliance with that ISO, if the test suite does not have the Standard to test a message informs that it is unavailable;
- **check_ISO4217** function tries to get the country currency code that is used in the ATM system and calls **checkHash** function passing the country of the ATM system and the currency code in the message as arguments;
- **checkHash** function does an assertion between the corresponding country and the currency code in the message. If the assert is true, the message is in compliance with ISO 4217.

```

1 public class ISO {
2
3     private static ISO theInstance = null;
4     private static SystemFacade systemFacade = null;
5     private ISO() {}
6
7     Hashtable<String, String> currencyTbl = new Hashtable<String, String>(){
8         {
9             put("United States", "USD");
10            put("Portugal", "EUR");
11            put("France", "EUR");
12            put("United Kingdom", "GBP");
13            put("Germany", "EUR");
14            put("Hungary", "HUF");
15        }
16    };
17     public static ISO getInstance() {
18         if (theInstance == null)
19             theInstance = new ISO();
20         return theInstance;
21     }
22     public SystemFacade getSystemFacade() {

```

```

23     return systemFacade;
24 }
25 public void setSystemFacade(SystemFacade facade) {
26     systemFacade = facade;
27 }
28 public boolean check(String standard) throws Exception {
29     switch(standard) {
30         case "ISO 4217":
31             Assert.assertTrue("The code in the card message with currency identification
32             is missing or doesn't exist.\n" + "For more information check iso 4217"
33                 , check_ISO4217());
34             break;
35         default:
36             System.out.println("No Standard Found");
37             return false;
38     }
39     return true;
40 }
41 private boolean check_ISO4217() throws Exception {
42     try{
43         String country = ISO.getInstance().getSystemFacade().getCountryCurrencyCode();
44         return checkHash(country, ISO.getInstance().getSystemFacade().
45             getCurrencyCodeInMessage());
46     }
47     catch (Exception e){
48         System.out.println(e);
49         System.out.println("Currency code getter is missing on Message.");
50     }
51     return false;
52 }
53 private boolean checkCurrencyCode(String country, String messageCurrencyCode){
54     try{
55         Assert.assertEquals(currencyTbl.get(country), messageCurrencyCode);
56         return true;
57     }
58     catch (Exception e){
59         System.out.println(e);
60         System.out.println("Wrong Currency Code. Do it according to ISO 4217");
61     }
62     return false;
63 }

```

Listing 4.5: ISO.java

Conclusions

This case study was essential to understand the need to make the test suite more independent to be reused with less effort. We studied different tools to do it and also understood which of them was more trustworthy and closer to the objective. The core of the test suite is described in this case study. We consider this the most important case study. It improved our way of testing the system, testing our hypothesis solution, the **Facade+ Test Suite**, and making it more trustworthy to implement it in the next case study without a system's existence.

4.4 Boeing 737 MAX - MCAS System

Motivation

This dissertation focuses on SCS development, after evaluating the previous approaches using ATM's, this case study implements a proposed solution related to an issue in a SCS that caused two plane crashes.

Problem

Since the used approach of **Facade+ Test Suite** in the previous case study could be applied to the ATM system without significant efforts and be reused in other ATM systems, it was assumed that it would also work in a plane's subsystem. This case study applies **Facade+ Test Suite** to avionics domain. However, as it was not viable to simulate the MCAS or an Airplane System because it is not open source, only the test suite was created.

Context

This case study aims to address the issue that caused the Boeing 737 MAX disasters. In October 2018 and March 2019, two Boeing 737 MAX jets crashed minutes after takeoff. After the second incident, all 737 MAX planes were grounded worldwide. The crashes were caused by a failure of an Angle of Attack (AOA) sensor and the subsequent activation of new flight-control software implemented in the 737 MAX, the Maneuvering Characteristics Augmentation System (MCAS). The MCAS software was implemented in the MAX version of the plane to compensate for the changes in the engines' size and placement on the MAX compared to the previous versions of the 737. This software, designed to prevent a stall due to the engines' reconfiguration, had a problem, it only read one AOA sensor. In a case where this AOA sensor fails, the MCAS system activates the anti-stall system and pushes the plane's nose down. If the system always did the readings from two sensors, maybe these crashes could have been eventually prevented [48].

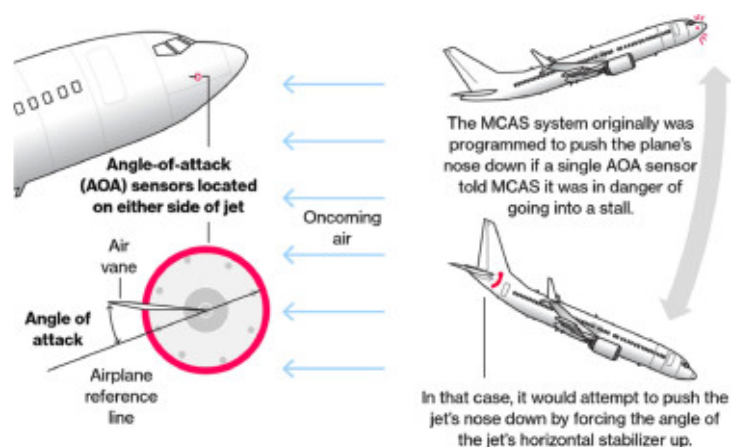


Figure 4.12: Angle-of-attack Sensors. [48]

Implementation - Avionics Facade+ Test Suite

The test suite is mainly implemented in four files: MCAS.feature, SensorSteps.java, SystemFacade.java and Standard.java.

MCAS is a feature file that contains a created scenario for this case study where a compliance with the reading of two sensors is checked.

```

1 Feature: MCAS stall control
2   Maneuvering Characteristics Augmentation System (MCAS) may be a stall
3   identification or protection system, depending on the natural
4   (unaugmented) stall characteristics of the aircraft. It is
5   a system that is designed to provide handling qualities for
6   the pilot, that meet pilot preferences.
7   AoA (Angle of Attack).
8
9   Background: AoA sensors
10    Given two AoA sensors are registered
11    And sense an high angle
12
13   Scenario: Without compliant
14    When the plane speed is not fast enough
15    Then MCAS is activated
16    And MCAS automatically swivels the horizontal tail
17    And moves the nose down
18
19   Scenario: With compliant
20    And compliant with FAASensorRule
21    When the plane speed is not fast enough
22    Then MCAS is activated
23    And MCAS automatically swivels the horizontal tail
24    And moves the nose down

```

Listing 4.6: MCAS.feature

SensorSteps is the class that contains the Step Definition related with the compliance test. The objective of this step is to call a function that checks the compliance of the system with the selected standard.

```

1 package thesisavionics;
2
3 import io.cucumber.java.en.And;
4 import org.junit.Assert;
5 import standard.Standard;
6
7 public class StandardStep {
8     @And("compliant with {word}")
9     public void compliantWithNewFAARule(String standard) throws Exception{
10         Assert.assertTrue(Standard.getInstance().check(standard));
11     }
12 }

```

Listing 4.7: SensorSteps.java

SystemFacade implements a Facade through an interface, a completely "abstract class" used to group related methods with empty bodies. The function **getAoASensorValues()** returns the value of the AoA sensors, whilst **sensorsEqualAoA()** returns if the sensors have the same value. As it is a SCS system, the verification is done in two ways, checking both if the system sensors values are equal and if the system confirms that they are equal. These two functions allow the test suite to verify if the system is compliant with the Standard used in this case study.

```

1 package standard;
2
3 public interface SystemFacade {
4     public float[] getAoASensorValues();
5     public boolean sensorsEqualAoA();
6 }

```

Listing 4.8: SystemFacade.java

Standard class implements the code related to the compliance verification:

- **check** function receives the Standard from the step definition present in **SensorSteps** and checks the compliance with that Standard, if the test suite does not have the Standard to test a message informs that it is unavailable;
- **check_faa_rule** function is divided into three parts, whereas if one of them fails the system is not compliant with the Standard:
 - Check if at least two sensors exist in the system if they do not exist the system is not compliant;
 - Check if the two first sensors have an approximate value;
 - Check if the system also confirms that the values of the sensors are approximate;

```

1 public class Standard {
2
3     private static Standard theInstance = null;
4     private static SystemFacade systemFacade = null;
5     private Standard() {}
6
7     public static Standard getInstance() {
8         if (theInstance == null)
9             theInstance = new Standard();
10        return theInstance;
11    }
12
13    public SystemFacade getSystemFacade() {
14        return systemFacade;
15    }
16
17    public void setSystemFacade(SystemFacade facade) {
18        systemFacade = facade;
19    }
20
21    public boolean check(String standard) throws Exception {
22        switch (standard) {
23            case "FAASensorRule":
24                Assert.assertTrue("AoA problem. MCAS should not be activated. An indicator
25                should alert the pilots.", check_faa_rule());
26                break;
27            default:
28                System.out.println("No Standard Found");
29                return false;
30        }
31        return true;
32    }
33
34    private boolean check_faa_rule() throws Exception {
35        float[] sensorValues = null;
36        try {
37            sensorValues = Standard.getInstance().getSystemFacade().getAoASensorValues();
38        }
39    }
40
41    }

```

```

37         //Check if two sensors exist
38         Assert.assertEquals(2, sensorValues.length);
39     } catch (Exception e) {
40         System.out.println(e);
41         System.out.println("The readings should be from 2 AoA Sensors.");
42     }
43
44     try{
45         //Check if the value is approximately the same from our side
46         Assert.assertEquals(sensorValues[0], sensorValues[1], 2);
47     }
48     catch(Exception e){
49         System.out.println(e);
50         System.out.println("The sensors value is different.");
51     }
52
53     try{
54         //Check if the sensor are equal in their side
55         Assert.assertTrue(Standard.getInstance().getSystemFacade().sensorsEqualAoA());
56     }
57     catch(Exception e){
58         System.out.println(e);
59         System.out.println("The sensors value is different.");
60     }
61     return false;
62 }
63 }

```

Listing 4.9: Standard.java

Conclusions

This case study was essential to understand if our approach would apply to an avionics domain. If it were possible to have access to a real-world system, it would be more trustworthy. However, it is not easy to get access to this type of domain.

4.5 Implementation Guide

Facade+ Test Suite is supposed to be implemented using the three roles mentioned before: domain expert, tester and developer. The implementation should be done using a path present in Figure 4.13. In order to implement **Facade+ Test Suite** there are some steps to be followed:

IDE and Cucumber

To implement our proposed solution Cucumber-JVM is needed, it is published in central Maven Repository or Gradle. It is also recommended to implement it using an IDE (we recommend IntelliJ IDEA or Eclipse) to facilitate the configuration. However, it is possible to use the command line. You should create a new project with Cucumber dependencies. It would help if you had the project's source divided into two different packages, the main source folder that contains the code that you will develop to the system and another one containing the test source folder where the cucumber feature and step definitions files should be (Figure 4.14).

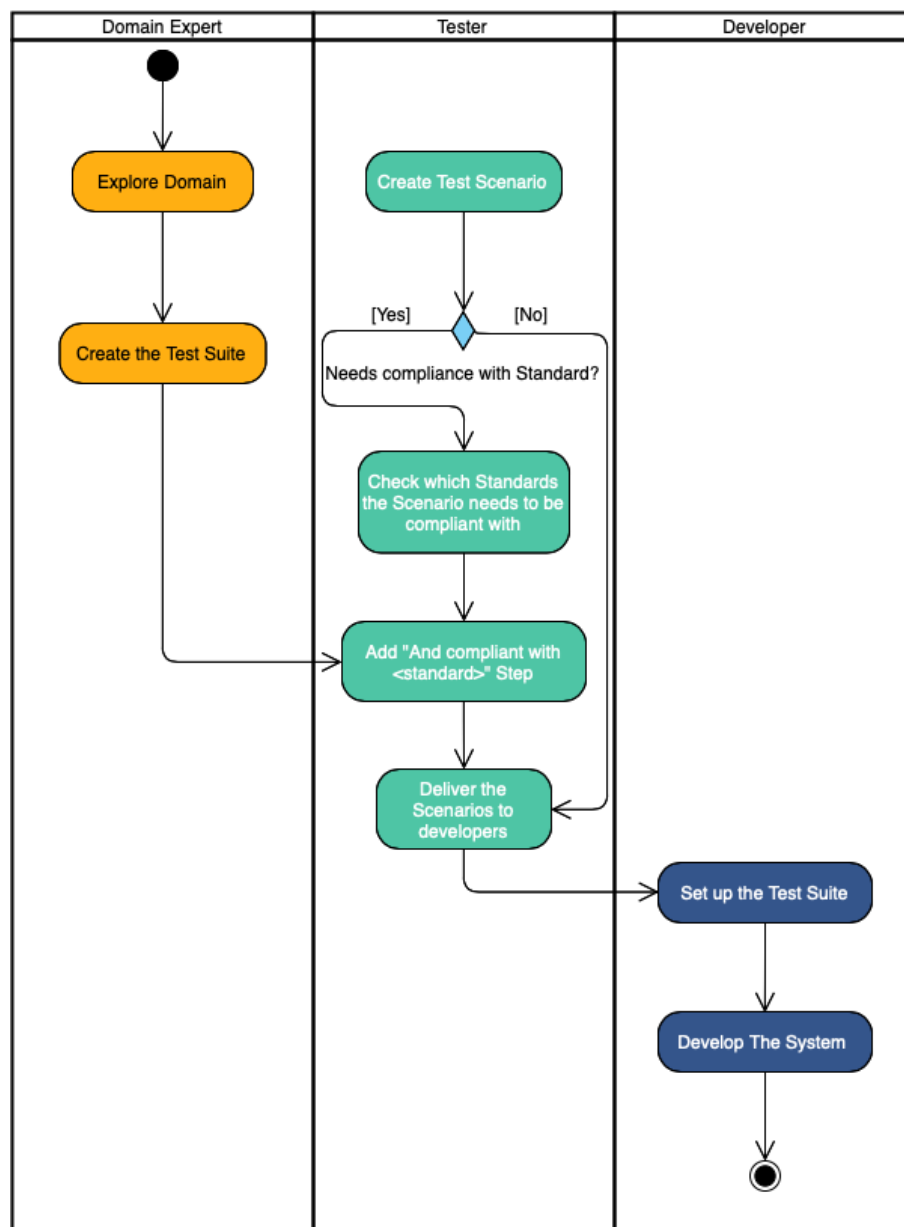


Figure 4.13: Activity Diagram - Roles

Create a Scenario

In order to test the system a feature file must be created. It should be in the resources of test folder. In this file, a feature (Figure 2.13) should be created in order to create a scenario to test.

Add a new Standard

In order to add a new Standard, these steps have to be followed:

- Create two packages, one for the step definitions and another one to the files containing the code that tests the Standard, also the Facade.

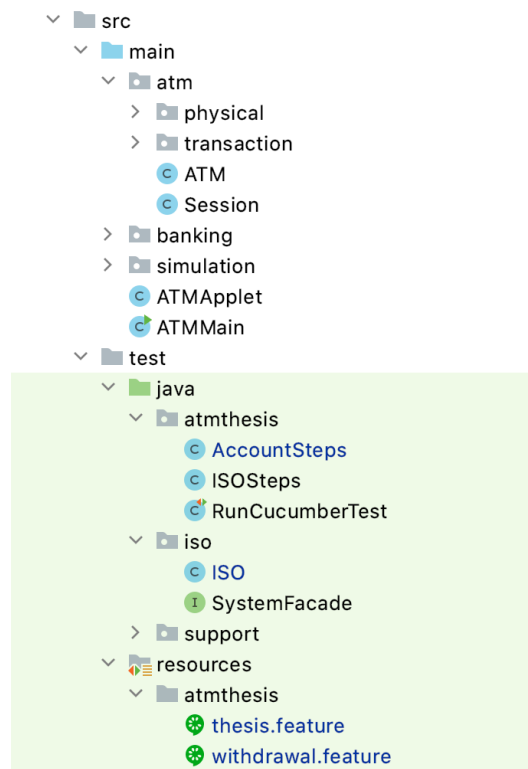


Figure 4.14: Source folder organization.

- In one of the packages that we recommend to have the same name as the project, the files with step definitions should be present (Figure 2.19). All the step definitions to test the Scenario and the step definition from our proposed solution that tests the Standard compliance (Listing 4.3) should be there.
- The other package should have the files containing the Standard verification (Listing 4.5) and the facade (Listing 4.4). The facade should contain the functions that developers should implement in the system in order to use the test suite with the Standard compliance tests too.
- Add the step "And compliant with <standard>" in the needed scenarios

Implement the test suite in the system being developed

In order to implement the test suite in the system being developed, developers need to create a new object of the facade (Figure 4.15).

Following these four steps, the proposed solution can be implemented. After it, there is only the need to run the tests needed and check their output to confirm success or unsuccessful.

```
class Demo {
    public static void main(String[] args) {
        String countryCC="123", codeInMessage="$";
        SystemFacade facade = new SystemFacade() {
            @Override
            public String getCountryCurrencyCode() {
                return countryCC;
            }

            @Override
            public String getCurrencyCodeInMessage() {
                return codeInMessage;
            }
        };
        // ...
    }
}
```

Figure 4.15: Facade implementation in the system.

4.6 Summary

In this chapter, we have described our proposed solution in more depth, using three case studies to implement our **Facade+ Test Suite** gradually. The first two case studies focus on using Cucumber and facades to implement our solution and the path to reach this solution. The last case study focuses on the application of the test suite in the avionics domain.

Chapter 5

Validation

This chapter aims to validate the solution proposed in this dissertation. This validation is divided into two parts. The first part assesses if there was an improvement in the agility of SCS development, according to the Agile Manifesto. The second part assesses if there was an improvement in the quality of SCS development, according to ISO-25010. These two parts will be analysed parts to validate if there was an improvement in all of them, in some of them or none of them. The overall validation of the thesis is achieved through the partial validation of these two parts.

Traditional SCS development usually uses a highly-structured, plan-driven approach such as V-model or waterfall [17] instead of agile methodologies. The waterfall model is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards through the phases 5.1. V-model (5.2) is a sequential process in which the next phase begins only after completing the present phase. In this model, steps do not move linearly while the steps are bent upwards. It is an extension of Waterfall model because V-model goes from left to right as well as it follows a sequential path of execution of processes like in waterfall model.

Our approach suggests the use of agile methodologies in SCS development. Even though agile methods have significantly changed how most software is developed, safety-critical software was widely excluded from this trend. Arguments for this exclusion relate to the need for documenting safety cases and the need to analyze and specify safety requirements upfront. Both thoughts conflict with agile values [32]. Through the use of BDD and after our case studies, we assume that some the arguments that cause the exclusion of agile methodologies can be overcome and that our proposed solution could be an excellent path to follow. The proposed solution seems to be a good way of documenting safety cases, improve the communication and understanding of the requirements between requirements experts, testers, developers and the client. Also, it provides a way to analyze and specify safety requirements upfront.

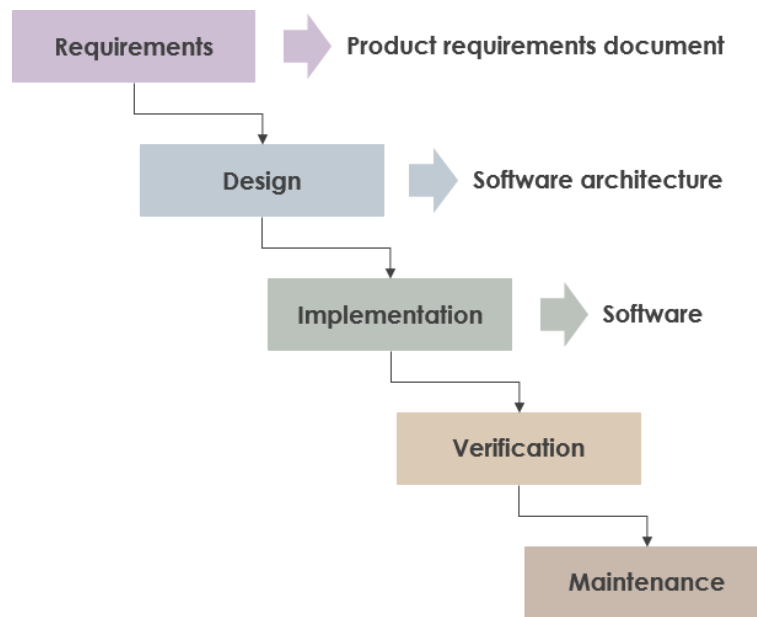


Figure 5.1: Waterfall Example [43]

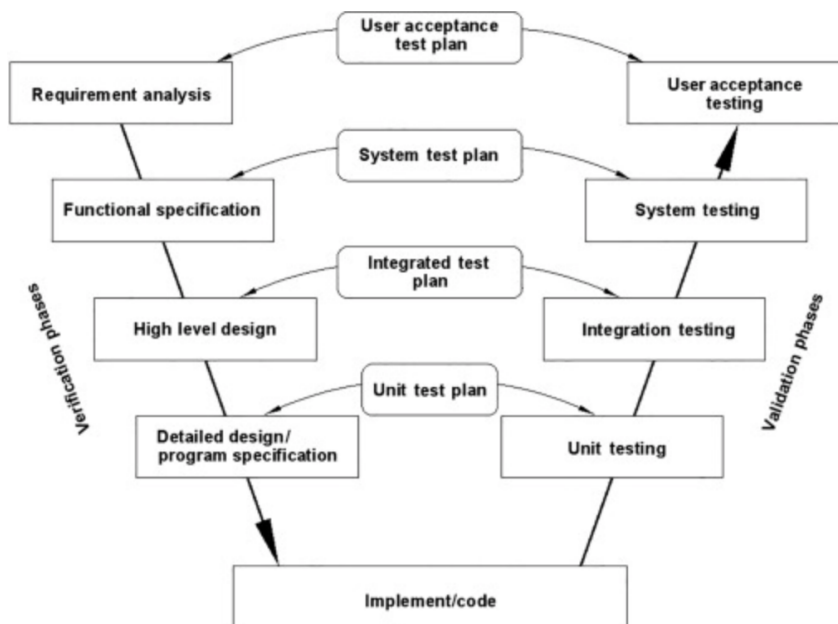


Figure 5.2: V-Model Example [8].

5.1 Agility Validation

The Agile Manifesto aimed to overcome the difficulties encountered by traditional models of software development, such as the Waterfall model. The manifesto would end up with the idea of well defined teams setting up a project and having contact with client only at the end. To document this, then created a list of the twelve principles of the Agile Manifesto.

In this part, the objective is to address which principles, stated in the Agile Manifesto [16],

were improved with our proposed solution. Furthermore, we analyse if the proposed solution goes against some of these principles. This twelve principles are the base of our agility validation:

- P1. Customer satisfaction by early and continuous delivery of valuable software** - Using our proposed solution through the use of BDD, the software is tested from the beginning. Furthermore, the user stories are validated as soon as they are ready and it will support the frequent deliveries, as the developer can deliver them to the client sooner, contrarily to waterfall, is commonly used in SCS development;
- P2. Welcome changing requirements, even in late development** - The requirements are validated by the proposed solution with minor effort, due to this changing in requirements is welcome as it is easier to validate changes at any time of the development cycle;
- P3. Deliver working software frequently (weeks rather than months)** - As in principle **P1**, the continuous validation of the developed system helps to deliver the software frequently as user stories;
- P4. Close, daily cooperation between business people and developers** - BDD facilitates the cooperation and understanding between testers, requirements engineers, developers and the client;
- P5. Projects are built around motivated individuals, who should be trusted** - Our proposed solution could make the individuals motivated as it facilitates the communication. However, we do not think it makes a significant difference;
- P6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation** - BDD is a set of practices that facilitate interactions between individuals in a team. Through conversations, examples and automated tests, the team is able to discover, and then implement, the desired behavior of a given software. It is based on the same principle as the 'three amigos', where business, development and QA collaborate to define what to do, how to do it and how to know if it's been done correctly;
- P7. Working software is the primary measure of progress** - BDD gives a platform to all stakeholders to identify any possible misunderstanding and confusion early in the process, helping the team building the right thing;
- P8. Sustainable development, able to maintain a constant pace** - We do not think that our proposed solution improved or worsened this principle;
- P9. Continuous attention to technical excellence and good design** - Our proposed solution gives continuous attention to technical excellence as it does a continuous validation of the developed software according to standards using the created step definitions;
- P10. Simplicity—the art of maximizing the amount of work not done—is essential** - The proposed solution is reusable in systems of the same domain that depend on a set of standards,

so it is possible to conclude that it minimizes the work to be done related to the validation of the system;

P11. Best architectures, requirements, and designs emerge from self-organizing teams - We do not think that our proposed solution improved or worsened this principle;

P12. Regularly, the team reflects on how to become more effective, and adjusts accordingly - We do not think that our proposed solution improved or worsened this principle;

After the analysis that was done previously, of each principle according to the proposed solution figure 5.3 contains a Likert scale that reflects it. Very low means that we assume that the selected principle was worsened significantly with our proposed, neutral means that we assume the proposed solution did not improve or worsen the selected principle whilst very high means that we assume the proposed solution improved SCS development's agility significantly.

	Very low	Low	Neutral	High	Very High
P1 - Customer Satisfaciton				X	
P2 - Changing Requirements				X	
P3 - Frequent Deliveries				X	
P4 - Cooperation				X	
P5 - Motivated individuals			X		
P6 - Face-to-face conversation					X
P7 - Working Software Progress					X
P8 - Sustainable development			X		
P9 - Continuous technical excellence				X	
P10 - Simplicity					X
P11 - Architectures, and requirements			X		
P12 - Team reflection regularly			X		

Figure 5.3: Impact on Agile Manifesto Principles.

The previous chart reflects the impact that we assume that our proposed solution improved in the agility in SCS development. After analyzing it, we assume that the proposed solution implies an improvement in the area of verification and validation in the development of SCS.

5.2 Quality Validation

In this part, the objective is to address if our proposed solution improves the quality of the development process of SCS. We analyse the quality standards from [12] to acknowledge if the proposed solution improves this process's quality.

• Functional Suitability

- **Functional Completeness** - degree to which the set of functions covers all the specified tasks and user objectives;
- **Functional Correctness** - degree to which the functions provides the correct results with the needed degree of precision;

- **Functional Appropriateness** - degree to which the functions facilitate the accomplishment of specified tasks and objectives.
- **Efficiency/Performance**
 - **Time Behaviour** - degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements;
 - **Resource Utilization** - degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements;
 - **Capacity** - degree to which the maximum limits of the product or system, parameter meet requirements.
- **Compatibility**
 - **Co-existence** - degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product;
 - **Interoperability** - degree to which two or more systems, products or components can exchange information and use the information that has been exchanged.
- **Usability**
 - **Appropriateness Recognizability** - degree to which users can recognize whether a product or system is appropriate for their needs;
 - **Learnability** - degree to which a product or system enables the user to learn how to use it with effectiveness, efficiency in emergency situations;
 - **Operability** - degree to which a product or system is easy to operate, control and appropriate to use;
 - **User Error Protection** - degree to which a product or system protects users against making errors;
 - **User Interface Aesthetics** - degree to which a user interface enables pleasing and satisfying interaction for the user;
 - **Accessibility** - degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use.
- **Reliability**
 - **Maturity** - degree to which a system, product or component meets needs for reliability under normal operation;
 - **Availability** - degree to which a product or system is operational and accessible when required for use;

- **Fault Tolerance** - degree to which a system, product or component operates as intended despite the presence of hardware or software faults;
- **Recoverability** - degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system.

- **Security**

- **Confidentiality** - degree to which the prototype ensures that data are accessible only to those authorized to have access;
- **Integrity** - degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data;
- **Non-repudiation** - degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later;
- **Authenticity** - degree to which the identity of a subject or resource can be proved to be the one claimed;
- **Accountability** - degree to which the actions of an entity can be traced uniquely to the entity.

- **Maintainability**

- **Modularity** - degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components;
- **Reusability** - degree to which an asset can be used in more than one system, or in building other assets;
- **Analysability** - degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified;
- **Modifiability** - degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality;
- **Testability** - degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

- **Portability**

- **Adaptability** - degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments;

- **Instability** - degree of effectiveness and efficiency in which a product or system can be successfully installed and/or uninstalled in a specified environment;
- **Replaceability** - degree to which a product can replace another specified software product for the same purpose in the same environment.

Figure 5.4 shows how we believe that our solution influences the quality of SCS development. Very low means that we assume that the selected principle was worsened significantly with our proposed, neutral means that we assume the proposed solution did not improve or worsen the selected principle whilst very high means that we assume the proposed solution improved SCS development's quality significantly.

		Very Low	Low	Neutral	High	Very High
Functional Suitability	Functional Completeness				X	
	Functional Correctness				X	
	Functional Appropriateness				X	
Performance Efficiency	Time-Behaviour			X		
	Resource Utilization			X		
	Capacity			X		
Compatibility	Co-existence			X		
	Interoperability			X		
Usability	Appropriateness Recognizability			X		
	Learnability			X		
	Operability			X		
	User Error Protection			X		
	User Interface Aesthetics			X		
	Accessibility			X		
Reliability	Maturity			X		
	Availability			X		
	Fault Tolerance				X	
	Recoverability			X		
Security	Confidentiality			X		
	Integrity			X		
	Non-repudiation			X		
	Authenticity			X		
	Accountability			X		
Maintainability	Modularity				X	
	Reusability			X		
	Analysability			X		
	Modifiability					X
	Testability					X
Portability	Adaptability			X		
	Instability			X		
	Replaceability			X		

Figure 5.4: Impact on Quality.

The previous chart reflects the impact that we assume that our proposed solution improved in SCS development quality. After analyzing it, we assume that the proposed solution implies an improvement in the quality of the development of SCS, mainly in **Functional Suitability** and **Maintainability**.

5.3 Summary

Our central hypothesis for this dissertation is that "Domain-specific Languages can be used to improve the quality and agility of Safety-Critical Software development". This hypothesis can be divided into two parts: Agility and Quality. For this reason, we decided to validate the dissertation in two parts as well. After analyzing each of these parts, it was possible to assume that the proposed solution can improve Agility and Quality of SCS development. We believe that our proposed solution can support our central hypothesis and answer all the research questions.

Chapter 6

Conclusions

This last chapter closes this dissertation with the conclusions regarding all the work involved with the research and the proposed solution. Section 6.1 presents an analysis of the final thoughts regarding this dissertation and answers the research questions and our central hypothesis. Section 6.2 presents the future work that should be done in the next possible approaches to improve further the solution developed.

6.1 Contributions

The certification process of critical systems is very time consuming and costly demanding. Throughout this dissertation, we aimed to improve the agility and quality of safety-critical software development through DSL's use to improve this process. By improving the Verification and Validation of the developed software, we believe that we can help this process since the software is tested from the beginning according to the Standards.

Having performed the case studies and analysis of the results obtained with the platform in Chapter 4, we should now look back on the main research questions and hypothesis to test with this dissertation:

- Research questions:
 - Is it easy to implement the verification of the requirements recurring to BDD?
 - Is it easy to use BDD to test the developed software compliance?
 - Will BDD give good traceability of the error to the Standard/requirement?
- Main hypothesis:
 - Requirements from Standards, regulations, and guidelines can be verified through test suites in Cucumber that developers can reuse, improving the development's agility and quality.

Analyzing these goals, we can evaluate how the work implemented during this dissertation and the resulting tool developed, accomplished them. Regarding the first research question and the hypothesis, we were able to implement a tool using BDD through Cucumber that successfully helped verify Standards. Furthermore, it can be reused between domains that depend on the Standards present in the suite. Furthermore, in chapter 5, it was possible to conclude that **Facade+ Test Suite** improves the development's agility and quality. Regarding the second and third research questions, the created tool is also straightforward to use and test if the software is compliant with the needed standards. Moreover, it seems to give excellent traceability to the place where the system is failing to be compliant. Even though it forces developers to create the methods present on the facade, the test suite is reusable and easy to use. This suite's use greatly facilitates the cooperation between domain experts, testers, and developers since using BDD it facilitates passing the knowledge about the domain and the Standards.

6.2 Future Work

Throughout this work, we have identified some possible improvements that could be tested and implemented to improve our solution:

- In order to better clarify why a test failed, it would be good for the developer to have a link to the Standard to read it. This way, the developer could better understand the Standard if he wants to. In our solution, it is just possible to see an error that says that the test did not pass and it is related to a Standard, also that the failure is related to something that is not wrong;
- It would be interesting to find a method that would not force the developer to create the necessary methods on the facade and that somehow the test suite could get the necessary data more efficiently. Perhaps an initial configuration where the developer was asked to indicate where the necessary variables for the tests could configure the test suite without using a facade;
- A case study applied in a real world SCS would be great to test our solution and to analyze the impact of it in the development process;
- It would be interesting to develop a questionnaire on what they thought about our solution's operation and implementation. Helping to improve and adapt our solution to make it the more suitable possible to be used in the real world.

Future researches should focus on making a case study in the real world to analyze how the solution works. It would be essential to know how the domain experts, the testers and the developers feel about using it. Furthermore, it would also be essential to analyze the impact in the system development's agility and quality.

References

- [1] Functional requirements vs non functional requirements. Available at <https://qracorp.com/functional-vs-non-functional-requirements/>.
- [2] Gherkin reference. *Gherkin Reference - Cucumber Documentation*. Available at <https://cucumber.io/docs/gherkin/reference/>.
- [3] Ricardo Alves, Vasco Amaral, João Cintra, and Bruno Tavares. A Family of Domain-Specific Languages for Integrated Modular Avionics. In *Communications in Computer and Information Science*, 2019.
- [4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [5] Christine M Belcastro. Validation and verification (v&v) of safety-critical systems operating under off-nominal conditions. In *Optimization Based Clearance of Flight Control Laws*, pages 399–419. Springer, 2012.
- [6] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75, 1984.
- [7] T. Bull. Comprehension of safety-critical systems using domain-specific languages. In *WPC '96. 4th Workshop on Program Comprehension*, pages 108–122, 1996.
- [8] Peter R.N. Childs. *1 - Design*. Butterworth-Heinemann, second edition edition, 2019.
- [9] E. Collins, A. Dias-Neto, and V. F. d. Lucena Jr. Strategies for agile software testing automation: An industrial experience. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pages 440–445, 2012.
- [10] Steve Easterbrook. *The difference between verification and validation*, 2010.
- [11] Sebastian Erdweg, Tijs Van Der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The state of the art in language workbenches. In *International Conference on Software Language Engineering*, pages 197–217. Springer, 2013.
- [12] John Estdale and Elli Georgiadou. *Applying the ISO/IEC 25010 Quality Models to Software Product: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings*. 01 2018.
- [13] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
- [14] Paul A. Fishwick. *Handbook of Dynamic System Modeling (Cpaman & Hall/Crc Computer and Information Science)*. Chapman & Hall/CRC, 2007.

- [15] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2010.
- [16] Martin Fowler, Jim Highsmith, et al. The agile manifesto. *Software Development*, 9(8):28–35, 2001.
- [17] X. Ge, R. F. Paige, and J. A. McDermid. An iterative approach for development of safety-critical software and safety arguments. In *2010 Agile Conference*, pages 35–43, 2010.
- [18] Emanuel S Grant. Towards domain-specific modeling methodology for avionic safety-critical systems. In *Proceedings of the World Congress on Engineering*, volume 1, 2018.
- [19] Emanuel S Grant. Principles of ethical consideration in safety critical software systems development. In *2019 4th Technology Innovation Management and Engineering Science International Conference (TIMES-iCON)*, pages 1–5. IEEE, 2019.
- [20] Magnus Härlin. Testing and gherkin in agile projects, 2016.
- [21] Florian Häser, Michael Felderer, and Ruth Breu. Is business domain language support beneficial for creating test case specifications. *Inf. Softw. Technol.*, 79(C):52–62, November 2016.
- [22] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.
- [23] Michael G Hinchey and Jonathan Peter Bowen. *Applications of formal methods*, volume 1. Prentice Hall New Jersey, 1995.
- [24] P. Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 134–142, 1998.
- [25] A. Idani, Y. Ledru, A. Ait Wakrime, R. BenAyed, and S. Collart-Dutilleul. Incremental development of a safety critical system combining formal methods and dsmls: Application to a railway system. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11687 LNCS:93–109, 2019. cited By 1.
- [26] Kyungsoo Im, Tacksoo Im, and John D. McGregor. Automating test case definition using a domain specific language. In *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ACM-SE 46, page 180–185, New York, NY, USA, 2008. Association for Computing Machinery.
- [27] IriusRisk. Bdd-security - continuum.
- [28] Codes for the representation of currencies. Standard, International Organization for Standardization, Geneva, CH, 2015.
- [29] Financial transaction card originated messages — interchange message specifications — part 1: Messages, data elements and code values. Standard, International Organization for Standardization, Geneva, CH, 2003.
- [30] Jan Jürjens. Developing safety-critical systems with uml. In *International Conference on the Unified Modeling Language*, pages 360–372. Springer, 2003.

- [31] Dimitris Karagiannis and Harald Kühn. Metamodelling platforms. In *EC-Web*, volume 2455, page 182, 2002.
- [32] Rashidah Kasauli, Eric Knauss, Benjamin Kanagwa, Agneta Nilsson, and Gul Calikli. Safety-critical systems and agile development: a mapping study. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 470–477. IEEE, 2018.
- [33] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5(2):149–161, 2009.
- [34] Hoyt Lougee. Software considerations in airborne systems and equipment certification. 2001.
- [35] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
- [36] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.
- [37] M. Micallef and C. Colombo. Lessons learnt from using dsls for automated software testing. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–6, 2015.
- [38] Ming Huo, J. Verner, Liming Zhu, and M. A. Babar. Software quality and agile methods. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, pages 520–525 vol.1, 2004.
- [39] Mockito. Mockito documentation.
- [40] Dan North et al. Introducing bdd. *Better Software*, 12, 2006.
- [41] Vicente Pelechano, Manoli Albert, Javier Muñoz, and Carlos Cetina. Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plug-ins. In *DSDM*, 2006.
- [42] Antti Perttula and Joni Kukkamäki. Enabling rapid product development through improved verification and validation processes. *Technology Innovation Management Review*, 10:25–36, 03/2020 2020.
- [43] Shari Lawrence Pfleeger and Joanne M Atlee. *Software engineering: theory and practice*. Pearson Education India, 1998.
- [44] Jan Pietzsch, Lauren Shluzas, Marie-Elisabeth Paté-Cornell, Paul Yock, and John Linehan. Stage-gate process for the development of medical devices. *Journal of Medical Devices*, 3, 06 2009.
- [45] Seb Rose, Matt Wynne, and Aslak Hellesoy. *The cucumber for Java book: Behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2015.
- [46] Russell C. Bjork. Atm simulation.
- [47] Daniel A Sadilek. Prototyping domain-specific languages for wireless sensor networks. In *Proc. of the 4th Int. Workshop on Software Language Engineering*, pages 76–91, 2007.

- [48] Tommaso Sgobba. B-737 max and the crash of the regulatory system. *Journal of Space Safety Engineering*, 6(4):299 – 303, 2019.
- [49] Alexander Shvets. *Modern Book on Design Patterns: Dive into Design Patterns*. Refactoring.Guru, 2018.
- [50] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 269–287. Springer, 2012.
- [51] C. Solis and X. Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, 2011.
- [52] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.
- [53] Hugo Lopes Tavares, Gustavo Guimaraes Rezende, Vanderson Mota dos Santos, Rodrigo Soares Manhães, and Rogério Atem de Carvalho. A tool stack for implementing behaviour-driven development in python language. *CoRR*, abs/1007.1722, 2010.
- [54] Jeff Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005.
- [55] Niksa Visic, Hans Georg Fill, Robert Andrei Buchmann, and Dimitris Karagiannis. A domain-specific language for modeling method definition: From requirements to grammar. *Proceedings - International Conference on Research Challenges in Information Science*, 2015-June(June):286–297, 2015.
- [56] Markus Voelter, Bernd Kolb, Klaus Birken, Federico Tomassetti, Patrick Alff, Laurent Wiart, Andreas Wortmann, and Arne Nordmann. Using language workbenches and domain-specific languages for safety-critical software development. *Software & Systems Modeling*, 18(4):2507–2530, 2019.
- [57] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.
- [58] Aline Zanin, Avelino Francisco Zorzo, and Henry Cabral Nunes. An example of aquila usage: A dsl for model-based testing in agile environments. Technical report, Technical Report, Pontifical Catholic University of Rio Grande Do Sul, 2018, 14p, 2018.