TRABAJO FIN DE MÁSTER

MÁSTER UNIVERSITARIO EN DESARROLLO DE SOFTWARE

Agile software development techniques to be proposed for validation with a case study of the vehicle driving automation industry

Acronym

Autor
Zain Ulabdeen Mohammed

Director
Manuel I. Capel Tuñón

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, February 2024

Se propondrán técnicas ágile de desarrollo de software para su validación con un estudio de caso de la industria de la automatización de la conducción de vehículos

Zain Ulabdeen Mohammed

Keywords: Ágil, sistemas críticos respecto de la seguridad, ISO-26262, FDD, STPA, TDD, sistema de control de crucero

Resumen

Las metodologías de desarrollo de software ágiles han sido consideradas durante la última década por la comunidad de Ingeniería de Software una de las más importantes metodologías para realizar de forma confiable y eficazmente la compleja actividad que denominamos "desarrollo de software" debido a la flexibilidad que presentan estas metodologías respecto del cambio de los requerimientos, colaboratividad y desarrollo iterativo en los mencionados procesos de desarrollo.

A pesar de las ventajas anteriores, los métodos ágiles no han sido adoptados con generalidad en el desarrollo de algunos tipos de sistemas complejos y, entre estos, cabe destacar los denominados sistemas críticos respecto de la seguridad (SCS), que podemos definir como aquellos sistemas en los que no se puede tolerar su fallo, mal funcionamiento, o errores que puedan llevar a pérdidas económicas significativas o de vidas humanas. Ejemplos de SCS son los sistemas de aviónica o aeroespaciales, sistemas de control de centrales nucleares y sistemas médicos; la razón fundamental de no adoptarse las metodologías ágiles en el desarrollo de los sistemas anteriores se debe a que requieren de una adhesión estricta a los estándares oficialmente aceptados durante el proceso de desarrollo, y este hecho contradice los "principios ágiles".

El principal objetivo de nuestro trabajo es proponer una metodología ágil adaptada a los requisitos de los mencionados SCS. En consecuencia, aquí se propone una nueva metodología basada en el método de desarrollo dirigido por características (FDD), que se adaptaría a las estrictas normas de desarrollo exigidas por SCS.

Para validar esta nueva propuesta de metodología, se ha desarrollado un caso práctico a la refactorización sistemática de un software complejo (control automático de la velocidad del vehículo AVSC), cuyo principal cometido es controlar la velocidad del vehículo con seguridad. Tal caso de estudio se ha realizado de acuerdo con la norma ISO 26262, que es una norma internacional de seguridad en el funcionamiento de vehículos, y que puede servir como guía para el desarrollo de sistemas eléctricos y electrónicos en vehículos de carretera.

Entre los objetivos alcanzados a partir de los resultados de este trabajo está el de contribuir al desarrollo de sistemas seguros siguiendo el mencionado estándar ISO 26262 con un coste menor y en un plazo de entrega más corto, sobre todo teniendo en cuenta que el coste de desarrollo de SCS es muy elevado y requiere un largo plazo que se puede llegar a prolongar años hasta su entrega final validada. La aplicación de esta metodología a otros casos de estudio, con la adición de métodos de verificación formal como futuros estudios, contribuiría a aumentar las posibilidades de evaluación de la utilidad práctica del método propuesto.

Agile software development techniques to be proposed for validation with a case study of the vehicle driving automation industry

Zain Ulabdeen Mohammed

Keywords: Agile, Safety-critical system, ISO-26262, FDD, STPA, TDD, Cruise control system

Abstract

Agile development methodologies have been considered one of the most important methodologies in software development in the last decade due to their flexibility in changing requirements, collaboration, and iterative development processes. Despite these advantages, Agile is still not adopted in the development of some types of complex systems, and among these systems are the so-called safety-critical systems(SCS), which we can define as systems that do not allow failure, malfunction, or error to occur that may lead to significant human or physical losses. Examples of this are aerospace and avionics systems, nuclear plant control systems, and medical systems. the reason for not adopting agile methodologies is that SCS require strict adherence to standards in the development process, and this contradicts agile principles.
The main objective of our work is to propose an agile methodology adapted to the requirements of SCS. A new methodology based on the feature-driven development (FDD) method has been proposed, which would adapt to the strict development standards required by SCS.
To validate this methodology, it was applied practically to refactoring complex software (automatic vehicle speed control AVSC), whose responsibility it is to control the speed of the vehicle, as a case study and in accordance with ISO 26262, an international functional safety standard for the development of electrical and electronic systems in road vehicles.
The results of applying this proposed methodology showed that a safer (AVSC) was obtained compared to what it was while adhering to the requirements of the ISO 26262 standard.

Among the effects of the results of this work is that it contributes to the development of safe systems following the standard at a lower cost and a shorter delivery time, especially since the cost of developing SCS is very high and requires a long time extending to years for delivery. This is considered an inappropriate option for companies that compete with each other at this time, which is characterised by rapid development and changing requirements. applying this methodology to other study cases, with the addition

of formal verification methods as future studies, would help to increase the opportunity for its evaluation

Acknowledgments

I am especially indebted to my supervisor Dr. Manuel I.Capel, I am grateful for his support and how he made me think as a researcher, and I consider myself fortunate to have had him as my supervisor.

Originality statement

Zain Ulabdeen Mohammed

I explicitly declare that the work presented as Master's Thesis (TFM), corresponding to the academic year, is original, in the sense that no sources have been used for the preparation of the work without proper citation.

Granada February 8, 2024

Signed:

D. **Manuel I. Capel (tutor)**, Profesor del Area de Lenguajes y Sistemas Informaticos del Departamento LSI de la Universidad de Granada.

Informa:

Que el presente trabajo, titulado "Se propondrán técnicas ágile de desarrollo de software para su validación con un estudio de caso de la industria de la automatización de la conducción de vehículos", ha sido realizado bajo su supervision por Zain Ulabdeen Mohammed, y autorizo la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expido y firmo el presente informe en Granada a 8 de febrero de 2024 .

El director:

Manuel I. Capel

Contents

Chapter 1

Introduction

1.1 Context

Agile software development (ASD) is currently considered one of the most widely used methodologies in the development of various types of software due to its efficiency in adapting to the changing requirements of the systems, trying to satisfy the customer and providing fast delivery of the software, however, Safety critical system (SCS) adhere to strict standards in their development and try to avoid changing specifications during development to reduce potential risks, therefore, this type of system has not yet made wide use of the advantages of developing software using agile methodologies.

1.2 Problem description

1.2.1 Agile software development (ASD)

ASD offers considerable benefits to the software development, such as rapid delivery of software products, instant feedback to the stakeholders, quick response to change and better coordination and collaboration[34]. In 2001 an "Agile Manifesto" was declared by 17 signatories (see `https://agilemanifesto.org/authors.html`)), which spans in 12 ideals within four fundamental values. The latter ones involve individuals and interactions over processes, software development based on detailed documentation, consultation with consumers supported by contract agreements and respond to change according to schedule[1]. These agile principles are good for software that needs getting speed in achieving the stages of the development cycle or does not depend on strict standards in its development. As it is the case of SCS, which do require extensive documentation and no changes in the development plan, a lot of drawbacks must be overcome to attain compatibility of

(SCS) development with agile principles; and therefore, most research nowadays deals with removing obstacles for an smooth adaptation SCS to agile principles. More specifically, these problems have an impact in four areas often covered in the current literature [32] on the subject:

- **Light documentation.**

  where the documents indicate that light documentation in agile software development is an obstacle in the development of these systems because "agile" focuses on "software works as the complete documentation"[1] and this is one of its principles as in the Agile Manifesto, however, this does not mean that agile ignores the documentation process, however, the effort is less compared to "plan-driven" development processes. However, it has to be considered, that from the point of view of reliable SCS development, documentation is the evidence of compliance with all standards and processes, and also that the system is secure and offers the required quality. Therefore, the long development period of critical systems needs to elaborate a heavy documentation for the system to continue in case of a change of the development team members, but the elaboration of heavy documentation leads to additional costs and less flexibility in the development.

- **Flexible requirements written in user stories.**
  Requirements management differs in agile from the traditional methods used in developing critical safety systems in two ways. Agile processes encourage changing requirements because they must easily and efficiently adapt to changes, although, in the development of critical safety systems it is not encouraged changing requirements because this causes an increase in the cost of redesign, documentation, testing, and red such changes delay delivery. Secondly, agile production can rely on loosely organized requirements such as user stories written in a simple way and thus it arises a conflict with traditional ideas of requirements where the development of critical safety systems requires the identification of complete, well documented and good requirements; and, therefore, documentation becomes very necessary. Changes can cause problems in the system structure, however, adaptation to systems' parameters or variables is also necessary for software development in general. When a complex medical device is developed, it needs years and changes are likely to be needed during this long period of time. Requirements can be divided into two parts, the first part is functional requirements and the second is safety requirements which are often not completely stable as long as system's requirements change dramatically over time

- **Iterative and incremental life cycles.**

In Agile, iterative is used at all phases of the development process in order to be flexible and adapt to changes. The customer can try iterative results in the issuance of a tested version, and, often, a V-Model is used in developing critical safety systems. Therefore, it is not implemented iterative and incremental, as is the case in Agile. V-Model encourages testing preparation to occur in parallel with requirements, design, and development phases, which is the only similarity with Agile methods. V-Model can be rigid and less flexible in comparison to agile methodologies. with a focus on quality management. Thus the life-cycle plan-driven must be adapted to become iterative in order for us to develop critical safety systems using agile. This is one of the barriers that prevent the use of agile in this type of system.

- **Test-first process.**

  Agile makes extensive use of test-driven development, for example, in the case of extreme programming (XP), testing is an essential practice in which test cases are written before the program, and all stages of testing are tested before code is developed. In contrast, in the development of safety-critical systems, thorough testing of the code is an important issue.. In the V model the testing is in the final stages of software development and the developers themselves write the tests needed to perform the test-driven development and this violates some of the rules in safety-critical systems, where the tester must separate his responsibility from the developer, i.e., tester and developer have to be two different people. In addition to all of the above, to develop SCS, it is necessary to add another test to verify compliance with the strict regulatory process, which is a complicated issue that, in principle, limits the use of agile.

1.2.2 Software architecture(SA)

The architecture of a software system represents the decomposition of the system into "modules" (or software components in general) and their interactions through interface contracts. In safety-critical systems, the architectural design is in accordance with strict standards and does not allow the design to be changed easily, it is possible that this change will result in a system failure in terms of non-compliance with security requirements, even if all functional requirements are still satisfied, and therefore the most important feature of agility when dealing with changing requirements is to prove that strict constraints (security requirements) will continue to be satisfied before accepting the change in the software architecture.

1.3 Stakeholders in this TFM

The main agents involved in this project are the following:

- Developer: the developer is a computer specialist who is in charge of devising and programming the software needed to solve a problem.
- Project supervisors: The supervisor is the person who has overall responsibility for properly defining the strategies for initiating, planning, designing, executing, monitoring, controlling and closing a project.
- Beneficiaries: the people who will benefit from the results of the project implementation. The main beneficiaries of the project are:
  - Interested companies: companies using neural networks can greatly benefit from the results of this project.
  - Spin-Offs of research projects of the group in which the work is framed. Among them, we have to highlight GRX Qualias Technology S.L., whose activity is mainly dedicated to providing quality assurance and software testing services.

1.4 Justification

The application of Agile development methodology in safety-critical systems offers many advantages, especially in today's era, which is characterized by the rapid development of technology. Therefore, implementing a methodology that adapts to urgent changes and produces high-quality software while providing more space for the customer to participate in software development processes at a (relatively) low price would be of great benefit.

1.4.1 Motivation and foreseeable outcomes

Safety-Critical systems (SCS) can be understood simply as those that in the event of a failure their behavior will " result in injury to people, damage to the environment or extensive economic losses". This is is why any error that leads to their failure cannot be tolerated because it causes serious damage. Examples of such systems are pathogen detection, vehicle traffic control, avionics systems, etc.[32] To ensure safety in these systems strict standards are followed[18], such as the IEC 61508 standard, which refers to safety-related systems, where one or more of those systems include electrical and/or electronic and/or programmable devices. Standards like ISO 26262 in vehicle systems, and DO-178 B / C in avionic systems are of special interest for these latter types of SCS. These guidelines govern safety practices that span over the whole life cycle of product production, along with the required technologies. Nowadays, the industry aims to fulfil these criteria in order to maintain a sufficient standard of safety to validate the issuance of certificates . This has led many organizations to follow the traditional waterfall approach in software development. However, it is difficult to make changes to requirements, when this phase is completed, if a traditional approach to developing SCS is used in their life cycle. As consequence of using the traditional waterfall model for the development of SCS software, the cost and delay in the final delivery date and an increased effort to change the product and then re-adopt it will be an impediment to providing new features or the ability to respond to customer requirements during development.[22] The research carried out by different groups working of SCS development indicates that agile methodologies can be suitable to achieve Quality of Service (QoS) and comply with standards and regulatory requirements that open the way for SCS can be developed using an agile approach[33]. The research literature demonstrates that organizations use at moment a variety of agile process elements when developing SCS. However, the development of their processes has to be customized to satisfy the regulatory criteria when using agile methods for this particular purpose, and which is the main motivation of the work to be carried out in this project. The main benefits that come out of the development of critical safety systems by using an agile software development process are the following ones:

- Continuous feedback, whether for clients, for the programmer‘s team, or testing team.

- Re-planning, based on the last requirements elicitation during all the development cycle.

- Be able to relate safety and functional requirements.

- Continuous monitoring (from requirements to coding and from coding to testing).

- Propitiate working as a team and define key-roles and responsibilities, such as: customer, development team, and evaluation team

- Enhance cooperation with the engineer, with regard to ensure reliability, availability, ease of maintenance and safety of the software product, as well as with that of the responsible of assigning the roles of quality assurance.

- Development of "first-test" of safety-critical systems [72]

1.4.2 Technologies to be used

Agile includes many methodologies such as Scrum, XP, FDD, Crystal, etc. Our research will highlight the Feature-Driven Development (FDD) methodology because it is characterized by its adaptability to complex and scalable systems, and this is what we need in critical safety systems because they are complex systems. The FDD methodology is integrated with System-Theoretic Process Analysis (STPA) technology in order to perform safety analysis and assessment of the possible causes of system failure before it occurs. In addition to using Test-Driven Development (TDD), which will allow us to produce more quality software with better productivity and improve the design before the actual code is written, unit tests are performed in TDD using junit. These techniques are integrated with each other as a case study to develop the automatic vehicle speed control (AVSC) program in Java and verify that it is safe according to the ISO 26262 standard.

1.4.3 Project Scope

The general aim of this TFM is to define a methodology of specific software development for the design of systems based on architectural patterns and ASD methods to develop a complex software system that has critical safety characteristics. For its definition we will cover different agile methods of Software development, pattern analysis, development principles, architectural alternatives. A final application proposal will be carried out: a validation supported by a case study based on the development in Java of a software for automatic vehicle driving. The specific research objectives are listed as it follows:

1. Get started, study and deepen the knowledge for the development of SCS software

2. Study state-of-art of selected agile methodologies: eXtreme Programming, Scrum, Feature Driven Development.

3. Define a new agile approach, so that safety-critical agile methodologies can be used in industrial software development of SCS reliably

4. Obtain an agile solution to the automatic vehicle speed control (AVSC) system software provided

5. Quality Assurance in Regulated environments: new proposals adapted to the AVSC

6. AVCS software refactoring according to the Quality assurance (QA) requirements found out in this investigation

7. Carry out an evaluation of the quality of the results obtained

There may be some expected obstacles to the implementation of our project, especially in small or less complex projects. The methodology proposed by us is based on the FDD methodology, which is characterized by its adaptation to large and complex projects and a heavy structure of development compared to other agile methodologies, which require the presence of a team with good experience in order to be able to produce safe and high-quality software [5].

1.4.4 Initial hypothesis

We propose to apply agile methods to re factorize a cruise control system in autonomous vehicles, which software has been delivered at the beginning of this project as the Automatic vehicle speed control AVSC system. Due to the complexity involved in such software and the need to implement safety standards, we will use this prototype as a study case to apply our approach in order to demonstrate final product‘s SCS properties fulfillment. The standard to be applied for QA assessment is ISO-26262 to develop software for electrical vehicles, and to verify that the product will meet the all the safety standards. More specifically, the methodology to deploy in this research work can be structured according to the following steps:

1. Selection of agile methodologies at stake today and can be used for SCS development
2. Study of methodological aspects that must be defined for obtaining a effective SCS- software development cycle according to the agile principles
3. "Implementation" of the new SCS development methodological approach
4. Definition of the "first tests process" considered as a key stone of the new methodology
5. Development of concrete tests, measures and analysis of results regarding AVSC refactoring with the new methodology

Project requirements

Most of the SCS are embedded systems, and that is why hardware plays an important role in the development of these types of systems, which are naturally complex systems that are governed by strict standards and require a very long time to develop with significant risks in the verification and validation process, in addition to very high costs, and for this reason, we have adopted in our project In a simulation of a automatic vehicle speed control to test the possibility of applying agile in the development of a safety system, we see that some of the following requirements are available in order to understand the system and how to implement the standards:

1. It is necessary to fully understand how the system works and how the hardware components interact with each other.

2. Clearly specify the safety requirements.

3. Analyze the system's safety and assess the hazards.

4. The life cycle of the system must be compatible with all phases of development imposed by the standards.

Possible obstacles and risks

One of the most prominent obstacles that we faced during the research is the lack of a methodology for software development in agile for the development of SCS, as most of the methodologies used are still the subject of research more than their actual applied aspect. It looks more theoretically than practical because SCS are characterized by complexity, high risks, and high costs.
In addition to the lack of a clear methodology. Defining safety requirements was one of the obstacles in the project, so we had to research for a mechanism that could adapt to agile methodologies that could extract safety requirements with high efficiency. And to predict possible risks in order to be avoided during development.
We had to do a lot of research to understand how the proposed automatic car speed control system works and how the hardware components and its software architecture interact. We also have to understand the relationship between patterns and software architecture and their impact on safety requirements.
In terms of risks, there are no actual risks during the implementation of the project. In fact, because the applied system is a simulator, there will certainly be potential risks if the application is practically on a vehicle. Potential risks that may arise from the safety point of view of this project during the implementation of this simulator should be prevented by first identifying their causes and then taking measures to control them.

1.4.5 General methodology and process to follow in the development of the TFM

Work methods

In this work, we have to propose an agile methodology that can be adapted to develop SCS. The system as a case study is the cruise control system that is present in vehicles, and although we already have the software, we have to re-design or refactor it to comply with ISO 26262-6:2018. The need to develop safe and high-quality software requires taking into account six processes:

1. safety hazard identification,
2. safety requirements engineering,
3. quality requirements engineering,
4. quality requirement fulfilment verification;

5. design implementation; and
6. verification.

TFM tracking tool

Git is a version control system that allows the developer to control all changes made to a project. This allows to have an absolute control of everything that happens in the code, being able to go back to a previous point in time or to continue through different branches of development.

Validation methods

Once the validation of the delivered AVSC software is finished, unit testing and requirements verification will be performed, according to the agile methods that we will use in the development of this project. These tests will consist of tracing backwards, from the system implementation to the functional and security requirements, satisfying them and testing their operation in a simulation environment in order to be able to to evaluate its performance.

1.5 Costs and sustainability study

Here we will discuss the different elements to be considered when calculating the cost of the project. Normally, in an engineering project, an estimate is usually made of the cost of each element with respect to its implication in the complete development of the project. Such costs are classified according to the following categories:

- Personnel
- Hardware
- Software
- Indirect and incidental costs

Personnel costs

Role	**Cost per hour** (€ per hour)
junior programmer	10,0
junior researcher	9,00
project manager	13,00

The above information has been obtained from `https://www.indeed.es/salaries`. Once we have the hourly cost of each of the roles of the participants in this project, we calculate for each one the hours spent in the performance of each task and, in this way, we calculate the cost of each of these tasks.

Hardware costs

This section specifies the costs of the hardware resources shown in the following table,

Hardware	Cost (€)	Useful life (Years)	Amortization (€)
Laptop	1450	4	363
Screen	200	4	50
Keyboard	20	4	5
Logitech mouse	15	4	4

Software costs

All software resources used in this project are free of charge. Some software services offer paid versions but these will not be necessary for the project.

Indirect and incidental costs

This section includes expenses derived from costs that are not directly necessary for the realization of the project.

Product	Hourly cost (€/h)	Hours	Cost (€)
Office			1500
Internet connection	0.08	446	35.68
Electricity	0.07	446	31.22
Total			1566.90

1.6 Planning and task scheduling

In order to plan the time schedule of the project, it is convenient to define in detail the tasks that constitute it as well as the key dates that delimit its duration. The reading of this project is scheduled for July 2023, so this TFM has had an approximate duration of XXX days, starting MM/DD/YYY and ending on 06/30/2023. The estimated time of development of this project has been 450 hours, which implies a dedication of about 20 hours per week. which implies a dedication of about 20 hours per week.

1.6.1 Description of tasks to be performed, estimates and Gantt chart

To carry out the project, the tasks have been grouped as follows,

1. Project management
    1. Introduction, contextualization and definition of the scope of the project.
    2. Project planning: definition of the roadmap; breakdown of the tasks to be performed and their timing.
    3. Budget: the cost of the project is estimated.
    4. Review and grouping of the documents delivered in tasks 1.1, 1.2 and 1.3 for the delivery of a final document.
    5. Meetings: weekly meetings will be held with the project director to follow up on the work carried out and clarify the work carried out and to clarify possible doubts.
2. Analysis and Design
    1. Search and review of similar studies carried out for SCS system developments.
    2. Study of the AVSC software supplied: the operation of the Java project packages was studied in detail in order to know and understand how to validate the implementation of the system.
    3. Definition of functional and safety requirements and internal system functionalities. standards.
3. Implementation, testing and validation
    1. Study agile methodologies for SCS software development.
    2. Define requirements of a new agile methodology for SCS
    3. Define iterations, sprints, scenarios, and organize the development cycle
    4. risk assessment of AVCS by concrete test cases
    5. Refactoring AVCS software using a new agile methodology towards scs
    6. Verify that the refactored software meets safety standards.
4. Documentation
    1. Writing of the project book.
    2. Preparation of the TFG defense presentation

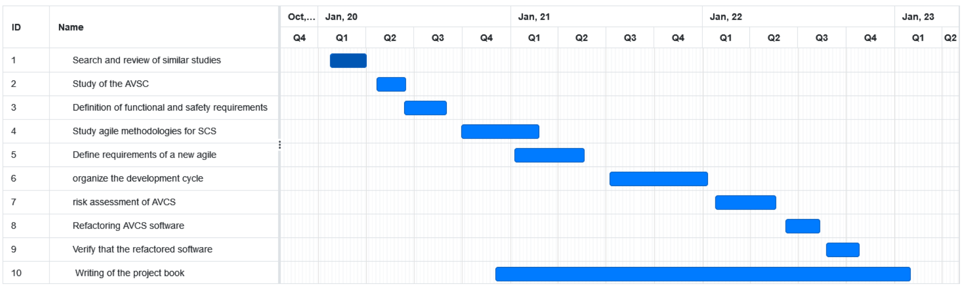


Figure 1.1: Project Gantt chart

Gantt Chart of the project development

1.6.2 Resources

In order to carry out this project, a series of minimum and indispensable resources must be available. The resources can be classified in three types, personal, material and software:

- Personal resources: One person, the developer, will be in charge of carrying out the project following the planning defined above. The project supervisor also participates as a guide and support role.
- Material and software resources: The following table shows the material and software resources of this project and the tasks in which they are needed.

Resource	Task
Office: Main workplace and meeting room at Google Meet	All
Office equipment: chair, desk, monitor, keyboard...	All
Laptop	All
Git: Software for version control	T3
GitHub: Repository where the code will be stored	T3
Overleaf: Software for the preparation of deliverables	T1, T2, T4
Eclipse: Refactoring AVCS and implementation test cases	T2,T3
Visual Paradigm: visual modelling and diagramming	T3
Google Chrome: Browser for information search	All

1.6.3 Risk management

As previously mentioned, in any project of a certain size, unforeseen events and complications can always arise, in this section we describe the possible ways to deal with these obstacles, and we have generously estimated the duration of the main sub-tasks. In case of having to extend the duration of any task, it would be decided to dedicate more hours per day to solve that problem in order to avoid that the other tasks are affected.

The task that can generate more inconveniences is the testing of the AVSC, since new cases could arise that contain structures or code fragments possibly not taken into account during the its implementation, and this implies having to modify the code or even rethink the way it works, so the duration of this task may have a great variability.

Chapter 2

State of the art

2.1 Agile Software Development

At the end of the last century, the agile software development methodology emerged due to the failure of plan-based development processes such as the waterfall model[73, 14], and this failure is a result of changing requirements severely in the world of software development. As researchers consider in [71, 64, 3], one of the additional criticisms added to traditional methodologies is that the software delivery time does not keep pace with the rate of changing requirements. In other words, the software delivery time could be significantly delayed by a small change in requirements.

There are many agile methodologies that share the development process with each other according to the basic principles of the agile manifesto[1]. The most notable methods are Crystal, Feature Driven Development [57], Extreme Programming (XP) [12] and Scrum [64]. Although the most obvious characteristic among them during the development process is that they are concurrent and iterative.
The iteration process typically occurs for two or three weeks, or perhaps less. At the end of each iteration, there is a delivery of a partial product to the customer, who gives his review and feedback.
In contrast to plan-oriented methodologies, multiple development activities (requirement analysis, design, implementation, and testing) in agile methodologies may occur concurrently. However, each of the agile methodologies is distinguished from the others by the practices taken in managing development of processes. During the past decades, it is noted that research in the field of agile methodologies touched on the practical side of applications and in various types of systems, starting from pure intensive-software to complex systems such as modern cyber-physical systems. By completing a comprehensive literature review to summarize recent studies on ASD,

authors [24]provide a current representative picture. Their findings demonstrate that ASD increase developer and customer productivity, as well as satisfaction.

2.2 Safety Critical System software Development

In Knight's view [42] *"Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment."* These kinds of systems are extensively used in a number of vital fields, including medical, aerospace, nuclear, and defense [66],Due to the high risks that can result from failure in the system, SCS are often developed within strict public or private standards according to the field in which the system will be used, such as ISO 26262 for applying to E/E systems in vehicles or IEC62304 for medical device software and IEC61513 for nuclear installations[32], and these systems usually consist of the interaction of their software components with hardware and firmware, and this diversity of components is a major challenge for developers to ensure that reliability and safety certification and standards are met[62].
Safety critical projects are usually organized in sequential stages[50]. The V-model is preferred by many to develop this type of software. The V-model is another form of the waterfall model,however, quality management and testing is extensively emphasized.[30],and it discourages incremental progress and iterations [29],The V-model is chosen for the development of SCS because it generates the high-quality documents required to obtain regulatory certification [50]. Yet, according to the authors[49, 45, 50, 60], regulatory standards processes do not impose a certain development life cycle, However, in the V model, or in the more traditional waterfall models, they are presented as a model that have to meet the requirements of the regulation, and if you decide to follow a different model, you must argue how you are going to meet the requirements. [52].

On the other hand, multiple studies have discussed how to present iterative development in SCS [31, 55]. However, studies show a challenge when iterative documentation and validation of the incremental are adopted [56, 39, 16], Beznosov[15] found that is the use of an iterative life-cycle in software development is similar between SCS and traditional systems, but, in the end, it becomes more complex in SCS. To change management is a major challenge with iterative development approaches where documentation must be constantly updated. Most iterations involve changes, which may cause an inability to ensure safety when a system is developed incrementally [39]. When planning to work iteratively, it is also challenging to assess the quality of safety. Therefore, it is recommended to carry out the safety analysis iteratively and incrementally to validate the safety properties of the system

[40].

2.3 Agile development methodologies in safety critical systems

The application of agile methodologies to develop safety-critical systems has led to a lot of discussion among researchers, some of them recognize the difficulties of integrating agile methods into this kind of system:

- Ernst Stelzmann [69]argued that since safety-critical systems involve both hardware and software, the hardware development lifecycle will encounter difficulties if development phases are in the form of small iterative sprints.
- While conducting interviews with 21 participants from different software organizations, Lubna Siddique [65] found that agile methodologies are not the typical choice when developing scalable projects, safety-critical-oriented software projects, or projects that have predefined requirements, and that the most appropriate model is the waterfall; and therefore, it is advised not to abandon the waterfall versus agile model. Nevertheless he suggests that the methodology must be integrated between waterfall and agile, thus achieving the advantages of both methodologies at the same time.
- Regarding the review of agile methods and plan-driven methods, Boehm found the type of project is what determines the development method, and that is why he believes that safety-critical systems require stable requirements and that agile methods do not seem suitable for these types of systems because they do not assume the existence of predetermined requirements. Moreover, he believes that a more comprehensive and advanced planning will reduce the risks of this type of system[17].

On the other hand, there are researchers who have conducted case studies of agile implementation with a variety domain of safety systems:

- The authors provide an analysis of agile practices for European railway software development within the EN 50128 standard. Agile practices all support some of the goals of that standard, according to a mapping between EN 50128 standards and those practices. They concluded that most agile processes need to be adapted to meet standards[38].
- The authors believe that the development of agile methodologies will be adopted to develop software systems for medical devices because agile methodologies provide many advantages, the most important of which

are a rapid response to problems or integration of new requirements and self-management. The outputs of this methodology will be high-quality software and more flexible systems[48].

- In this paper[76], the author discusses how agile and XP(eXtreme programming) practices can enhance hardware software for flights. This shows how developers are constrained by the strict RTCA DO-178B standard, which imposes limitations on traceability and authentication. This is essential in the aviation software industry because authorities need proof that software complies with standards imposed for development. However these two things take their toll in terms of slow work. By following iterative methods, especially by applying them to each iteration, the researcher expects that the adoption of agile processes for aeronautical software development will be a necessity in the future and will produce safe software as in other methodologies.

Some researchers have adopted improved traditional agile methodologies and adapted them to the constraints of safety-critical systems development processes:

- Fitzgerald and others have successfully applied agile concepts in a regulated environment at QUMAS Organization and the R-Scrum methodology has been proposed. Quality Ansurance (QA), security, safety, traceability, effectiveness, verification, and validation are key aspects that showed compliance with ASD if applied to regulated development environments, which can be improved by using this general method compared to traditional use of Scrum[27].

- Several researchers[68] worked on improving the Scrum model to make it better suited to the development of safety systems, so that it is more flexible in terms of requirements specification, planning and documentation and therefore acceptable according to IEC 61508. Among the proposed changes were to make safety analysis processes, define safety requirements, and describe the general scope prior to the completion of an iteration so that validation and verification can be done at the end of the iteration or the entire project, and to recommend the use of Test-Driven Development (TDD) for the design to be considered before being implemented by the developers' and thus make the TDD output a simple documentation as required by the IEC 61508 standard .

- In order to increase flexibility and ensure safety, Wang has integrated System-Theoretic Process Analysis (STPA) technology with Safe-Scrum, as he found that this step may stimulate future research on whether moving Safe-Scrum from academia to its application in the software industry could be useful and mandatory to obtain a better development practice of SCS. Wang concluded that Safe-Scrum helps ensure safety, but it is less agile compared to traditional Scrum. Agility and Safety are improved with Scrum Security amendment[74].

- The authors [19] presented a new methodology, S4S, for developing software with critical safety features, especially in railway systems. This methodology is an extension of Scrum. The results of this research showed the possibility of safe iterative development while providing updated documentation and making all processes safe and responsive to human errors. The researchers suggested applying this methodology to other projects, especially those that include external components.

Chapter 3

Background

3.1 Software Development Methodologies (SDM)

A SDM is considered a framework for the application of software engineering concepts, whose purpose is to provide a roadmap for software development. That is why SDM is considered an essential part of software engineering, which demonstrates the way to implement in a timely manner and work in an orderly manner according to the different technologies used[58].

3.1.1 Plan-driven software development methodologies

The plan-driven software development approach is characterized by a sequential set of activities mentioned by Hirsch[35]:

1. To specify requirements in advance with respect to performing a detailed project planning,i.e., from start to finish.
2. Requirements are specified in detail, and changes to requirements are strictly implemented afterwards.
3. Design of the system (at different levels of detail) is carried out before implementation, as well as the specification of the architectural software.
4. Coding is only performed in the programming phase.
5. At the end of the project, different tests are conducted in the produced software and the entire system.

Examples of methodologies that adhere to the principles of the plan-driven software development approach include the waterfall model, V-model, and Rational Unified Process (RUP) .

Waterfall model

One of the most famous software development models is the waterfall model, also known as the cascading model. It was first introduced in 1956 by Felix Torres and D.Benington, but as the first process diagram developed by W.Royce in 1970[61]. The waterfall model explained the phases of big software development projects by starting in a sequence of:

- requirements identification
- analysis
- design
- coding
- testing
- and operations

W.Royce indicated that at the same time, despite his belief in this concept, he does not hide his concern that the testing is the last phase. In addition to the fact that this model faces difficulty in the event of changing requirements and therefore must be re-designed, in addition to the difficulty of making use of an iterative approach. The following figure3.1 illustrates a diagram of the waterfall model.

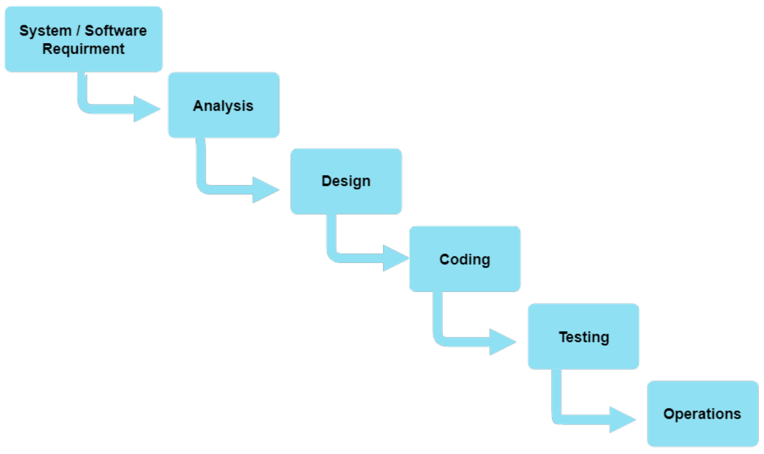


Figure 3.1: Waterfall model diagram

V-model

The V-Model was developed by NASA in the year 1991 and is also called the Vee Model, which is another variant of the waterfall model that is V-shaped as in the figure 3.2. The model descends as in the waterfall model until the low top of the model is reached, then upwards towards the right side of the figure, where the component assembly, verification, validation, and integration processes take place ,The verification is carried out in accordance with the level of the stem in the form of the corresponding side of the V stem at each phase. upwards[28].

On the V-model, testers and developers collaborate simultaneously. One benefit of the V-model is that the testers will actively participate in identifying the requirements and writing tests. Changes to the requirements are acceptable under this approach. Despite this, the drawback of the V-model form is its rigidity and inflexibility; if the requirements change, the test documentation and the requirements must be updated as well. This may result in significant costs and a delay in delivery. Therefore, this model is typically used for long-term projects[21].

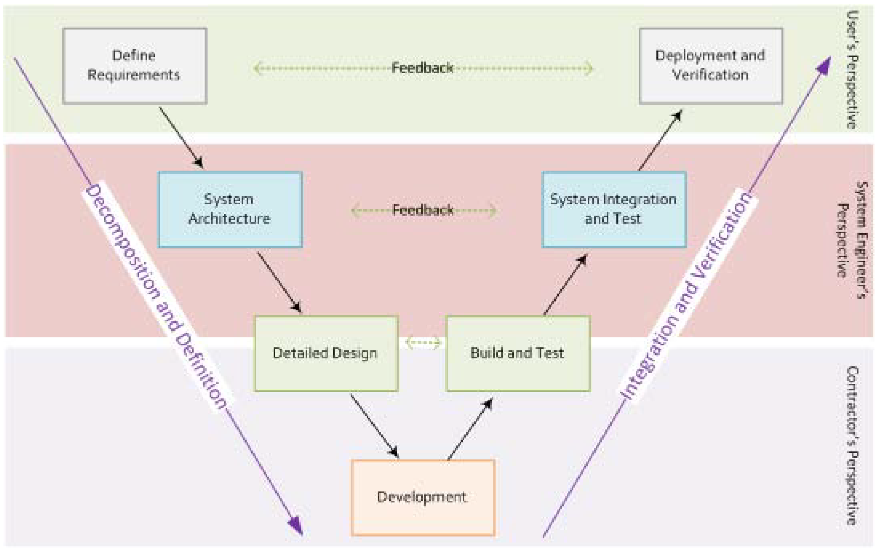

Define Requirements
Feedback
Deployment and Verification
User's Perspective
System Architecture
Feedback
System Integration and Test
System Engineer's Perspective
Decomposition and Definition
Integration and Verification
Detailed Design
Build and Test
Development
Contractor's Perspective

Figure 3.2: V-model process diagram [63]

3.1.2 Agile development

Based on the "Agile Vision" defined in the Agile Manifesto, ASD is a combination of an incremental and iterative approach. In 2001 [1], the "Agile Manifesto" was signed, which introduced four key values and derived twelve principles from them.

Agile manifesto values

1. "individuals and interactions over processes and tools."
2. "working software over comprehensive documentation."
3. "customer collaboration over contract negotiation."
4. "responding to change over following a plan."

Agile Principles:

1. "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."
2. "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage"
3. " Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale."
4. "Business people and developers must work together daily throughout the project."
5. "Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done"
6. "The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."
7. " Working software is the primary measure of progress."
8. "Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely."
9. "Continuous attention to technical excellence and good design enhances agility."

10. " Simplicity the art of maximizing the amount of work not done is essential."

11. "The best architectures, requirements, and designs emerge from self-organizing teams."

12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly."

Most agile models share these principles and values and strive to achieve them. The following models have received a great deal of interest by researchers:

Scrum

Scrum was proposed by K Schwaber in the 1990s as a process for software development and is considered one of the most famous methodologies that follows the principles of Agile. Scrum follows an iterative and incremental approach to predicting and controlling potential risks.[64].Scrum is described as a flexible framework that manages the resources available from development teams, technologies and tools to achieve a better product, the following figure 3.3 illustrates the scrum processes, in addition to this model, there are three main stakeholders[10].

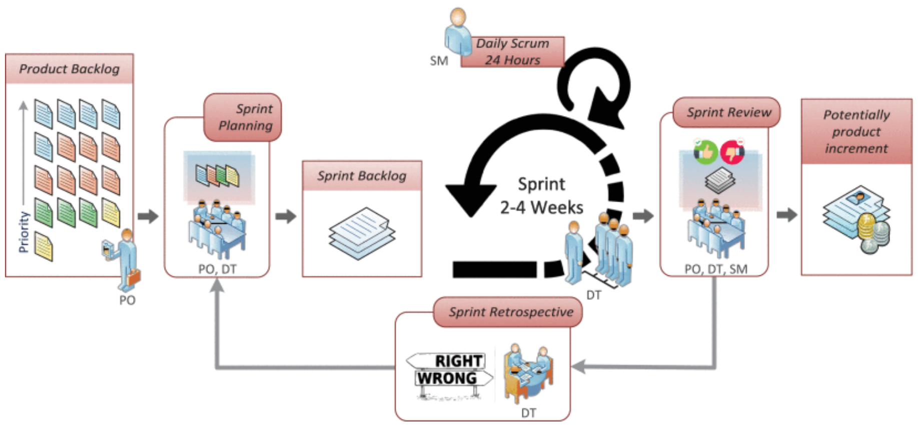


Figure 3.3: scrum life-cycle [10]

- Product owner

  his duty is to determine the functional and non-functional requirements and any requirements that work on them, and arrange these requirements according to the priorities requested in the requirements queue product backlog.

- Master Scrum

  He is the project manager who has to remove obstacles to work, manage timing and conduct meetings, and maintain scrum values and practices.

- Scrum team or development team

  Responsible for developing the requirements defined in the product backlog

The kickoff, the meet sprint planning meeting, the sprint, the daily scrum, and the sprint review meeting are the five main steps in the Scrum process[20].

1. A sprint planning meeting brings together the developer team (scrum team), the product owner, and the scrum master at the start of the iteration (sprint). The duration of the meeting is one day, and its outputs are in two parts. The first one is to determine the product backlog, the goal of the sprint, and the expected results from this sprint, and the second part is the meeting for planning the work and define tasks during the next sprint, often determined by the scrum team. This output is known as the sprint backlog.

2. The kick-off meeting is similar to the sprint planning meeting, but the difference between them is that in the kick-off, the key objectives of the project and the high-level product backlog are defined.

3. After that, the sprint begins, in which tasks are developed and according to what was specified in the sprint backlog. This race usually takes between one and four weeks, and it is not possible to change the requirements during the sprint.

4. Usually,before sprint making a daily scrum meeting with the scrum team and scrum master takes place for a quarter of an hour to answer specific questions, which are what you did, what you will do, and what prevents the work from continuing.

5. A sprint review meeting is held at the end of the sprint, during which the product owner is informed of what has been accomplished. This meeting is characterized as informal and is not disruptive to the Scrum team.

eXtreme Programming (Xp)

One of the widespread agile approaches developed by Kent Beck in 1996, is characterized by highly flexible, adapting to ambiguous and changing requirements, is keen on producing high-quality software, and is oriented towards small development teams.To guide XP practices, there are four core values that must be followed[12]:

1. Communication

   Communication among development team members, managers, and customers is the highest priority in order to avoid failure that could affect the project.

2. Simplicity

   Start writing the program and design work as soon as the requirements are obtained without predicting the future because anticipation means that there is a delay in the work and an additional cost. This means that you should think about making things simple as much as possible

3. Feedback

   XP Practices are designed to provide the development team with early feedback by running early test cases for unit testing and integration testing, and with this feedback, the development team can be assured that work is being done in the right way.

4. Courage

   Encourage taking bold steps in the work without hesitation as a redesign or refactoring of the code can always be carried out at a later date.

The practices of XP
According to Beck[37], XP practices can be divided into 12 practices that are considered the XP lifecycle, and these practices are under the umbrella of the values mentioned earlier[54].

1. *Planning:* Define the next iteration range by working with clients who provide tasks priorities and with developer who provide technical estimates, the story card be useful in this case .

2. *Small Releases:* Getting a quick version of the system leads to getting reviews and comments from the customer to improve the product.

3. *Metaphor:* This is the phase of designing the system architecture, in which it is explained how the system works, whether for developers or the customer point of view.

4. *Simple design:* One of the values that XP depends on is simplicity. In this practice, a simple design is made for what we have as requirements without going into details or thinking about the future.

5. *Testing:*Through this practice, test units are conducted by programmers, and tests need acceptance by the customer.

6. *Refactoring:*is to re-design the system without affecting the design goals, and programmers show this behavior in order to improve quality.

7. *Pair Programming:*To improve the quality of the product, work is done in the form of pair programmers, and this is one of the features of XP, where the code is written by one of them and the other writes tests.

8. *Collective ownership:* The code can be accessed by all the development team in order to contribute to improving the quality by reviewing the coding and accelerating the work.

9. *Continuous integration:* Work on the integration of the program units and testing of them is done upon completion of each task. This process reduces future failures when the integration test will be performed.

10. *40-hour week:*Encouraging not to work long hours

11. *On-site customer:*The customer is part of the work team and, therefore, acceptance tests are approved.

12. *Coding standards:*Sharing the code with colleagues and following the standards to write the code leads to clean and high quality code.

XP roles and responsibilities

In the XP team, there are seven roles and responsibilities that must be defined[7, 12]:

- *programmer:*responsible for writing the code, which is the main effectiveness of the XP model, as well as the programmer to perform the tasks of the designer, analyst, and architect, because the XP model is devoid of these responsibilities.

- *customer:*The customer's job is to write requirements stories and approve the results of functional tests.

- *tester:*Cooperating with the customer to write and verify tests, but he does not do the unit-tests because it is the responsibility of the programmer in the XP model
- *tracker:*His responsibility is to monitor the progress of the iteration and gather information from the team work on each task, how much time was spent, and the remaining missions to the end of the sprint.
- *coach:*He has the ability to communicate between work teams in addition to having the administrative and technical qualifications to maintain the workflow correctly.
- *consultant:*This role is not always available, but an expert is contracted for a short period to guide the developer team in the event of technical problems
- *manager:*He is the actual project manager, and decision-making is his responsibility, by communicating with the work team and assessing the situation.

Xp Process

The lifecycle of XP is divided into six phases, in order and as shown in figure 3.4[3].

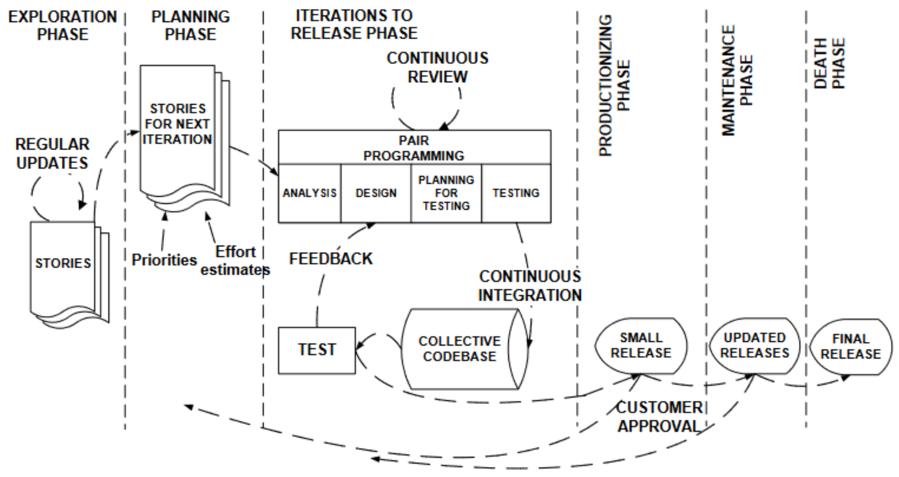


Figure 3.4: Xp life-cycle[3]

1. Exploration. At this *phase:* an exploration of the requirements is made for the version or written by the customer through story cards, and the work team tries to adapt techniques and tools to meet these

requirements in the project, while the system architecture is explored by making a prototype.

2. *Planning:*The first version of the system is planned by prioritizing a set of requirements stories and estimating the number of programmers they need to develop these requirements according to a schedule that usually does not exceed two months, while the planning phase takes two days.

3. *Iterations to Release:*Following the initial planning phase, the work is broken down into a number of iterations in accordance with the timetable. Typically, the implementation phase lasts a week to a month. We acquire an initial release of the system after selecting by the customer which of the stories will be implemented in each iteration, implementing the customer's specified tests at the end of each iteration.

4. *Productionizing phase:*At this stage, many tests and verification are conducted on the system before delivery to the customer, and if new changes are found, they can be implemented in the current version or postponed with the documentation of these changes

5. *Maintenance and Death:*In the maintenance phase, it occurs when the iterations are completed and the initial release is produced. Therefore, the team working with the customer is keen to make this version still work, and during that time, other iterations are made until the customer's stories run out. This is called the "death stage." It is also called "death" in the case that the results do not match the requirements of the customer, because this will be too expensive, and documentation is also being done at this phase.

Feature-Driven Development (FDD)

Jeff De Luca invented it in 1997 for a Bank of Singapore software development project. In 1999, Jeff De Luca, Eric Lefebvre, and Peter Coad published the book"Java modelling in Color with UML" where the world first learned about FDD. Later, S. Ballmer and M. Felsing released "a practical guide to Feature-Driven Developmen" that is distinct from Java modelling, as well as a more detailed explanation of how FDD operates[57].
FDD is a flexible and adaptive approach that follows an incremental, iterative principle in software development processes. Client satisfaction is the goal of this approach, which is characterized by a focus on software development processes at the building and design phases ,and follow the ETVX template,Because it does not cover all development processes, the FDD is

made to be integrated with other activities in the software development process.However, it is not necessary to utilize a particular process model.The FDD has demonstrated its effectiveness in the industrial sector, where it places a focus on quality throughout the whole process and involves regular tangible deliveries as well as precise tracking of the project's progress. [57, 4, 5, 3].

FDD process
FDD processes contain five consecutive processes and provide the principles and techniques that stakeholders need while working on system development, in addition to roles and timelines in project management. This approach is characterized by being suitable for critical systems, as claimed by Palmer and Felsing[57, 4]. The five processes according to Palmer and Felsing, illustrated in the following figure3.5, are:

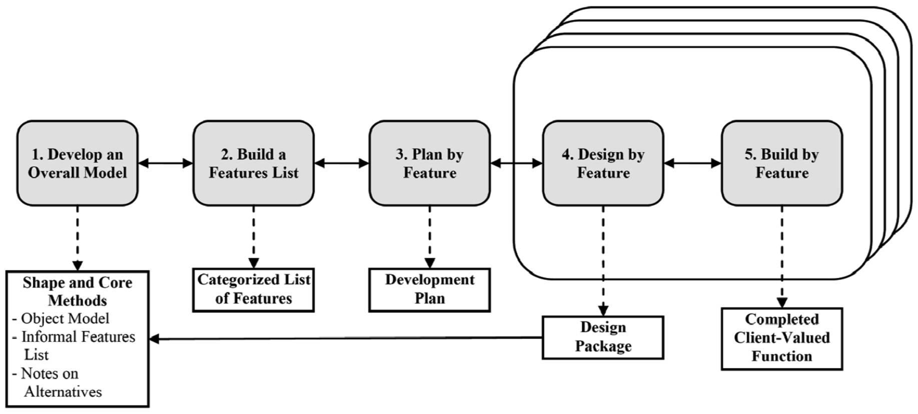


Figure 3.5: FDD process according into Palmer and Felsing [58]

1. Develop an Overall Model.
   It is considered the first stage of this model, where its outputs are the development of an overall model of the system. A walkthrough high-level meeting is held before this model is developed, and afterwards, a set of models are developed by domain experts. One or several models are chosen that are unified into this one comprehensive model, and it is possible to improve this overall model later[4].

2. Build a Features List.
   it is the second phase and from the name it turns out that it is a set of features where it is easy to identify these features based on the domain object model that was previously defined. These features are collected into groups according to the link between them. Usually the

implementation time of each feature is a maximum of two weeks, but if it is expected that the feature needs a longer time, it is divided into sub-features and all these features are approved by the customer[4].

3. Plan by Feature.
   Any feature that will be implemented is planned by assigning each feature to a chief programmer (or programmer), and the choice of which feature will be implemented depends on the dependencies between features. In addition to their priority, their level of the complexity, and the estimated load on the work team[57].

4. Design by Feature.
   The previously identified features are scheduled at this phase, and several features are assigned to a group of programmers, particularly those who use the same classes, and the owners of these classes are defined. The feature assigned programmers team generates a set of sequence diagrams for the chosen features, and the overall object model is refined and the design is verified in accordance with these detailed sequences [58, 57].

5. Build by Feature.
   The final phase of FDD involves working on the features that have decided to make it through the design phase, producing the essential code for them, performing unit testing and code inspection, and if the tests are passed successfully, carrying the features to the building by integrating the features and modules[5].

FDD Roles and responsibilities
There are many roles that are categorized according to FDD [57, 3] into three taxonomies:

1. key roles.

   1. *project manager:*Leading the project is one of his responsibilities, as he is an observer of the progress and financial management and must eliminate anything that could affect the progress of the work.
   2. *chief architect:*Overall design is a chief architect's responsibility and management of a design session, and he has the final word in design decisions.
   3. *development manager:* The team's problems are resolved by the development manager, who also guides daily development efforts. In addition, this position has the duty of addressing resource issues. The duties of this position might be merged with those of the project manager or chief architect.

   4. *chief programmer:* An experienced programmer leads small teams and participates in the effectiveness of analysis, design, identification of features that enter into iteration and are developed, and identification of each class with its owners in the programmers' teams. He/she is also responsible for solving technical problems and reporting on work progress weekly.
   5. *class owner:*Class owners carry out the tasks of designing, coding, testing, and documenting under the direction of the chief programmer. He is in charge of the class's development for which he has been assigned as the owner. The class owners whose classes are chosen as features for the following development iteration are participating in each iteration.
   6. *domain experts:* The customer, business analyst,sponsor, or a combination of these may be the domain expert. His or her responsibility is to understand how the various requirements for the system being developed should operate. Domain experts provide this expertise to the developers so that they may produce a reliable system.

2. supporting roles.

   1. *Release manager:* By evaluating the chief programmer's progress reports and having brief progress meetings with them, the release manager manages the process's progress. He notifies the project manager on the status.
   2. *language lawyer:*A team member responsible for having in-depth knowledge especially when the team is working on a new technology
   3. *build engineer:* a person in charge of installing, running, and maintaining the build process, as well as producing documentation and administering the release control system.
   4. *tool smith:* the toolsmith's job is to create small tools for a project's development, testing, and data conversion.
   5. *system administrator:*his or her responsible for configuring, managing, and troubleshooting the servers, workstation network, and development and testing environments used by the project team.

3. additional roles.

   1. *Tester:* Performs acceptance tests and ensures that requirements are being implemented correctly.
   2. *Deployer:*his or her task is to publish new versions while specifying the format that the system requires in order to function properly and configuring the settings.

3. *Technical Writer:*Performs system documentation.

Test-Driven Development (TDD)

TDD is a popular Agile practice for software development that Kent Beck introduced in 2002, and he claimed TDD will enhance and produce quality software products, both internally and externally (i.e., in terms of functionality), while improving developer's productivity[13, 8].
Following are the phases in the TDD methodology[59]:

1. Before starting to write the actual code, a test is written for the specific function (it was a class or method) that is planned to be executed

2. Execution of the test and the test's output must be a failure (red test) or passing test (green test) to ensure that the test writing was correct.

3. Writing a simple code that leads to the fulfillment of the requirements of the function to be tested. In the event that the test passes, the integrity of the code will be confirmed; in the event of failure, the code must be refactored to achieve the functional requirement.

4. Improving the code to be easily readable and maintainable.

5. Back to step1

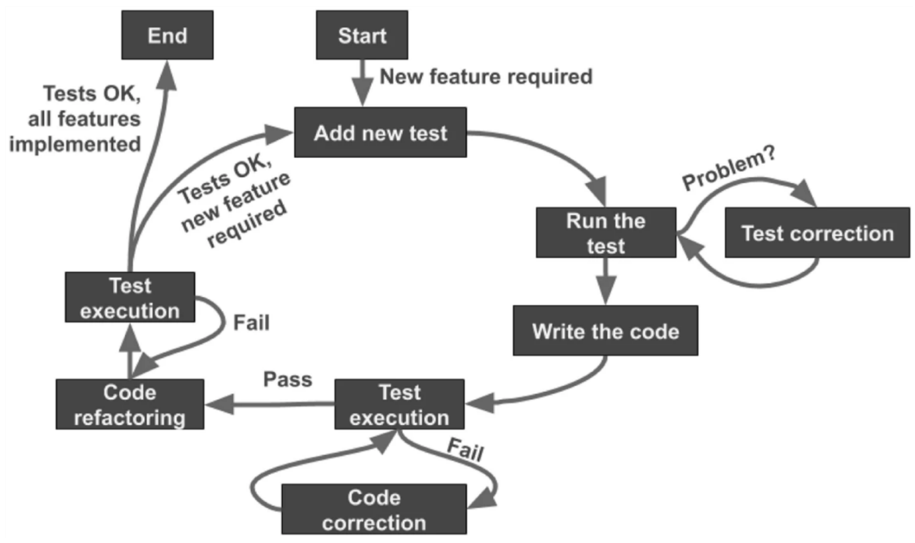


Figure 3.6: TDD phases diagram [59]

3.2 Hazard Analysis

There are a number of different ways that the term "hazard analysis" may be defined, and these definitions change based on the industry and the specific topic being discussed. Hazard analysis is "the process of identifying hazards and their potential causal factors" according to Leveson's definition. The purpose of doing a hazard analysis is to avoid, minimize, and-or regulate the dangers and the factors that produce them [43]. Methods of hazard analysis may either be "inductive" or "deductive," depending on how they arrive at their conclusions. According to Ericson[25],[77], "inductive" procedures are employed for the identification of hazards(bottom-top) while "deductive" studies are used for the identification of root causes(top-bottom). For instance, the FTA and STPA are examples of a deductive approach, whereas the FMEA is an example of an inductive method. Regarding the topic of our study, we will cover the STPA approach in more depth than other methods, since it is the technique used to analyze risks in our project.

3.2.1 System-Theoretic Process Analysis (STPA)

The Systems-Theoretic Accident Model and Processes (STAMP) has been created by Levinson 2004 then she developed a System-Theoretic Process Analysis (STPA). Using STAMP as a base, the STPA method has been created in 2012 to assess the system's safety. For complex systems, the theory of systems is particularly important [36]. When dealing with such complex systems, it is necessary to utilize a method that focuses on the system as a whole, rather than the sum of separate subsystems, as the division of systems into components does not give a clear picture of the safety of the system or conclude an inaccurate result. This technique analyzes risks at the level of the system as a whole. It takes into account the interactions of the components of the system among themselves, whether they are software, hardware, the environment, or even human factors.[43].
In order to do a hazard analysis in accordance with STPA standards, a control action diagram must be created as shown in the figure 3.7. This control diagram must include the components of a system as well as their respective channels of control and response. The STPA is performed in the two phases that are detailed below[70]:

1. Determine if there is a possibility that the system is not being adequately controlled, which might result in a dangerous situation. A condition is considered to be hazardous if it violates the safety criteria or limits of the system and, as a result, has the potential to result in some kind of loss, whether it be in terms of life, the mission, or finances.

2. Find out how each potential risk control action specified in step one could happen (identify the causal factor),then safety determinants that avoid or mitigate the impact of unsafe actions are concluded. Unsafe actions are categorized according to the table3.1

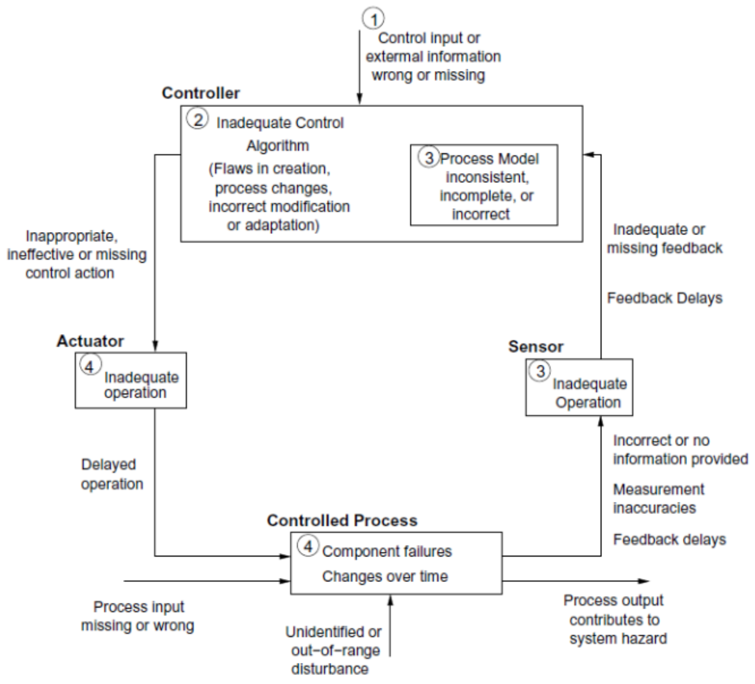

1 Control input or external information wrong or missing
Controller
2 Inadequate Control Algorithm (Flaws in creation, process changes, incorrect modification or adaptation)
3 Process Model inconsistent, incomplete, or incorrect
Inappropriate, ineffective or missing control action
Inadequate or missing feedback
Feedback Delays
Actuator
4 Inadequate operation
Sensor
3 Inadequate Operation
Delayed operation
Incorrect or no information provided
Measurement inaccuracies
Feedback delays
Controlled Process
4 Component failures
Changes over time
Process input missing or wrong
Unidentified or out-of-range disturbance
Process output contributes to system hazard

Figure 3.7: Control structure Diagram illustrate a casual factor[36]

Before beginning the process of safety analysis, we first locate any potential accidents or risks in the whole system. When putting STPA into practice, we begin with the control structure of the system as a kickoff point. Using the control structure as a guide, we examine each control action with regard to four broad categories of potentially hazardous behavior. The following are the four categories of control actions:

1. Required control action is not provided.
2. Provided (not required) is an unsafe control action.
3. Too early or late control actions are provided (wrong time).
4. Stopping control action too soon or applying for a long time.

These four points will be placed in a table as in Table 3.1 and fill in the information that will later be considered basic safety requirements. By applying the second step of the STPA, the causal factors of the hazard are revealed.

Unsafe Control Action(UCA)				
control action	required(not provided)	provided(not required)	Provided too (early/late)	provided too(long/soon)

Table 3.1: STPA Step 1

3.2.2 Fault Tree Analysis

Traditional safety analysis techniques such as Fault Tree Analysis (FTA) are often used to find the underlying reasons and potential of an undesirable event occurring. These are the aims of using FTA. This technique, starts by searching for the events that cause the risks from the top, then it derives the causes to the bottom. We can search for all the possible causes[25], and by the FTA, the design of the system will be improved and be made more reliable. This technique is considered a deductive method because it works by decomposing hazard events into smaller parts. Failure states are represented graphically and in isolation in this method. Failures can be traced back to their root causes using a tree-based approach based on logic gates as in the figure 3.8, probability, and Boolean algebra concepts. Building the Fault Tree (FT) is the most important part of this technique, and it is why this method is referred to as "Fault Tree."[25, 44].

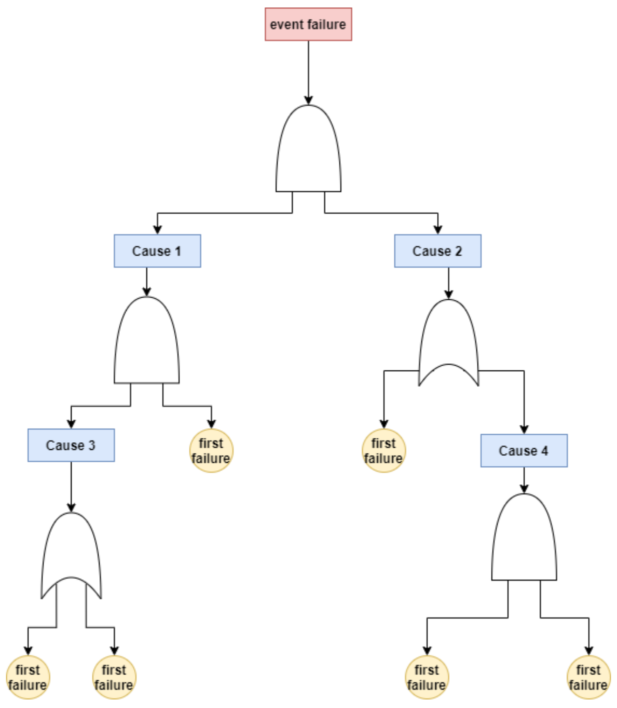


Figure 3.8: simple Example of fault tree

3.2.3 Failure Modes and Effects Analysis

FMEA It is a method that ends the inductive approach in the sense that it analyzes the causes of failure from the bottom-up, unlike (STPA, FTA), by which the causes of potential hazardous and negative effects on the system are determined. This method adopts the inductive approach, that is, starting from the bottom up. With this method, the design can be improved early on, potential hazards can be found, and steps can be taken to deal with them [70]. According to the author's opinion,[6], FMEA can be summarized in the following steps, and to be more clear, as shown in the figure 3.9.This method is considered a systematic and iterative approach that begins with a clear definition of a process that leads to failure, and this process is determined by an experienced team. A classification is made into failure patterns according to the severity of the risks and their frequency, and it helps in this classification by making a graphical map of potential risk events with data entry in a table to make an analysis of them. Then, it comes the process of redesigning or modifying the operations to avoid failures or reduce their impact, and then implementing and evaluating the efficiency of the improved procedures.

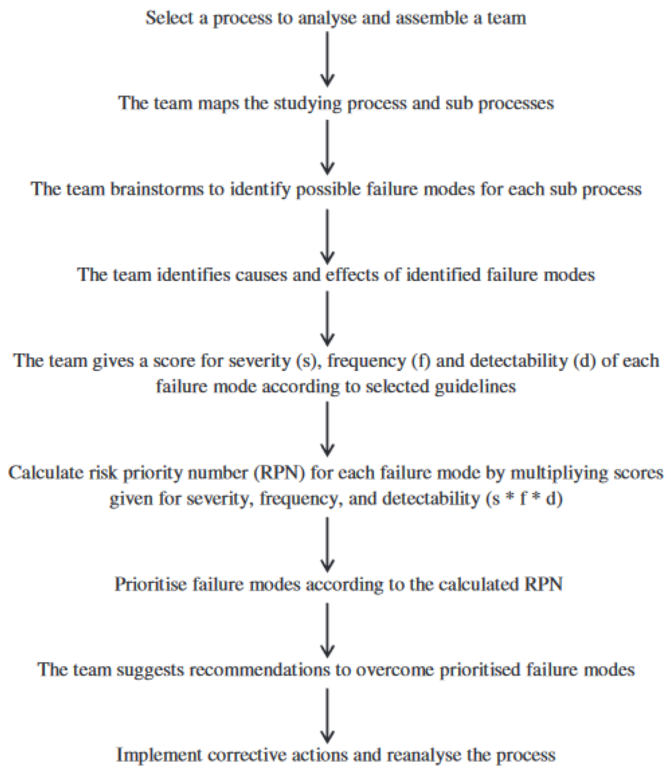

Select a process to analyse and assemble a team
The team maps the studying process and sub processes
The team brainstorms to identify possible failure modes for each sub process
The team identifies causes and effects of identified failure modes
The team gives a score for severity (s), frequency (f) and detectability (d) of each failure mode according to selected guidelines
Calculate risk priority number (RPN) for each failure mode by multipliying scores given for severity, frequency, and detectability (s * f * d)
Prioritise failure modes according to the calculated RPN
The team suggests recommendations to overcome prioritised failure modes
Implement corrective actions and reanalyse the process

Figure 3.9: Steps of FMEA[6]

3.3 ISO 26262

The series of standards is specifically intended to provide a benchmark in the development and design of electrical and/or electronic (E/E) systems intended for use within road vehicles. ISO 26262 was adaptated from IEC 61508.developing automobiles, particularly as vehicles are becoming more complicated due to the incorporation of sophisticated technology and software into their manufacturing. This complexity raises the chances of software or hardware failures, increasing the hazards to passenger safety. This standard, ISO 26262, thus provides guidelines for mitigating or avoiding such risks[2].
To ensure functional safety, the ISO 26262 standard provides:

1. a reference for the automotive safety life-cycle and supports the tailoring of the activities to be performed during the life-cycle phases,
2. a risk-based method tailored specifically to the automobile industry to establish integrity levels. Automotive Safety Integrity Levels (ASIL), utilizes ASIL to identify which of the ISO 26262 requirements are relevant in order to minimize unacceptable residual risk.
3. Functional safety management specifications, design and implementation, as well as verification, verification and confirmation procedures.
4. requirements for customer-supplier relationships.

3.3.1 ISO 26262 parts

This standard consists of 12 parts at different levels of development, ten parts of which are normative (1–9, and 12), and two parts are guidelines (10,11). These parts can be summarized in the following figure3.10. We note in Figure that the ISO 26262 standard specifies the interdependence between its parts by shading the parts (3,4,5,6 and 7), and this shading is in the form of V, where the V-model is the default model for product development[2].

1. "ISO 26262-1:2018, Road Vehicles Functional Safety Part 1: Vocabulary."
2. "ISO 26262-2:2018, Road Vehicles Functional Safety Part 2: Management of functional safety."
3. "ISO 26262-3:2018, Road vehicles Functional safety Part3: Concept phase."
4. "ISO 26262-4:2018, Road vehicles Functional safety Part 4: Product development at the system level."

5. "ISO 26262-5:2018, Road vehicles Functional safety Part 5: Product development at the hardware level."
6. "ISO 26262-6:2018 Road vehicles Functional safety Part 6: Product development at the software level"
7. "ISO 26262-7:2018, Road vehicles Functional safety Part 7: Production, operation, service and decommissioning."
8. "ISO 26262-8:2018, Road vehicles Functional safety Part 8: Supporting processes"
9. "ISO 26262-9:2018, Road vehicles Functional safety Part 9: Automotive Safety Integrity Level (ASIL) oriented and safety-oriented analyses"
10. "ISO 26262-10:2018 Road vehicles Functional safety Part 10: Guidelines on ISO 26262"
11. " ISO 26262-11:2018 Road vehicles Functional safety Part 11: Guidelines on application of ISO 26262 to semiconductors"
12. "ISO 26262-12:2018 ,Road vehicles Functional safety Part 12: Adaptation of ISO 26262 for motorcycles"

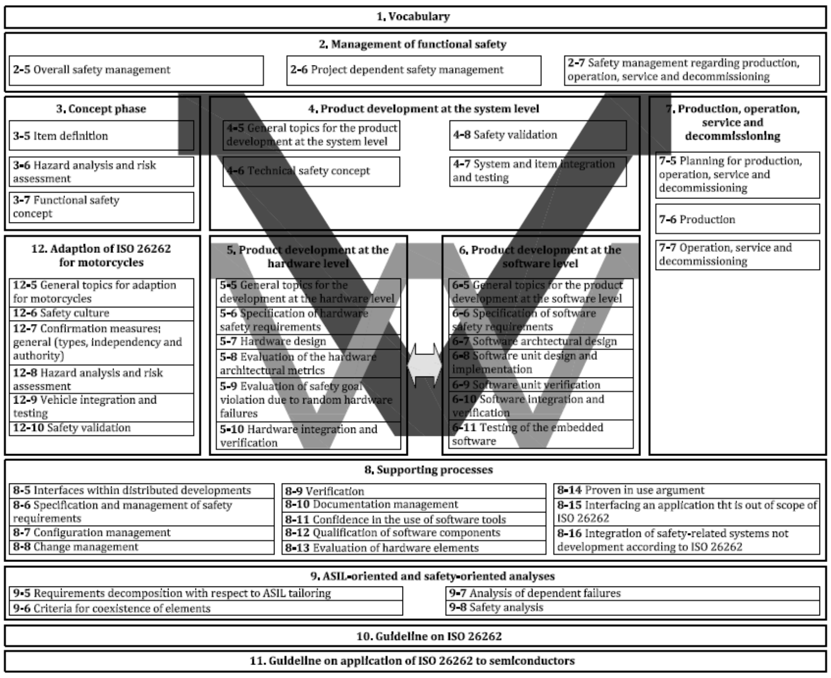

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Project dependent safety management
2-7 Safety management regarding production, operation, service and decommissioning
3. Concept phase
3-5 Item definition
3-6 Hazard analysis and risk assessment
3-7 Functional safety concept
4. Product development at the system level
4-5 General topics for the product development at the system level
4-6 Technical safety concept
4-8 Safety validation
4-7 System and item integration and testing
7. Production, operation, service and decommissioning
7-5 Planning for production, operation, service and decommissioning
7-6 Production
7-7 Operation, service and decommissioning
12. Adaption of ISO 26262 for motorcycles
12-5 General topics for adaption for motorcycles
12-6 Safety culture
12-7 Confirmation measures; general (types, independency and authority)
12-8 Hazard analysis and risk assessment
12-9 Vehicle integration and testing
12-10 Safety validation
5. Product development at the hardware level
5-5 General topics for the development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of safety goal violation due to random hardware failures
5-10 Hardware integration and verification
6. Product development at the software level
6-5 General topics for the product development at the software level
6-6 Specification of software safety requirements
6-7 Software archtectural design
6-8 Software unit design and implementation
6-9 Software unit verification
6-10 Software integration and verification
6-11 Testing of the embedded software
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation management
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Evaluation of hardware elements
8-14 Proven in use argument
8-15 Interfacing an application tht is out of scope of ISO 26262
8-16 Integration of safety-related systems not development according to ISO 26262
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analysis
10. Guideline on ISO 26262
11. Guideline on application of ISO 26262 to semiconductors

Figure 3.10: An overview of the ISO 26262 standard parts[2]

3.3.2 ISO 26262-6 Software Development Level

For automotive applications, this part of document specifies software level development requirements, including the following[2].

- Fundamental themes for software-level product development.
- The software safety requirement specification.
- Software architectural design
- Design of software units.
- Verification of software unit
- Integration and verification of software.
- Embedded software testing.

The following model clarifies the software life cycle, and this V-model is included in ISO 26262's sixth and fourth parts.

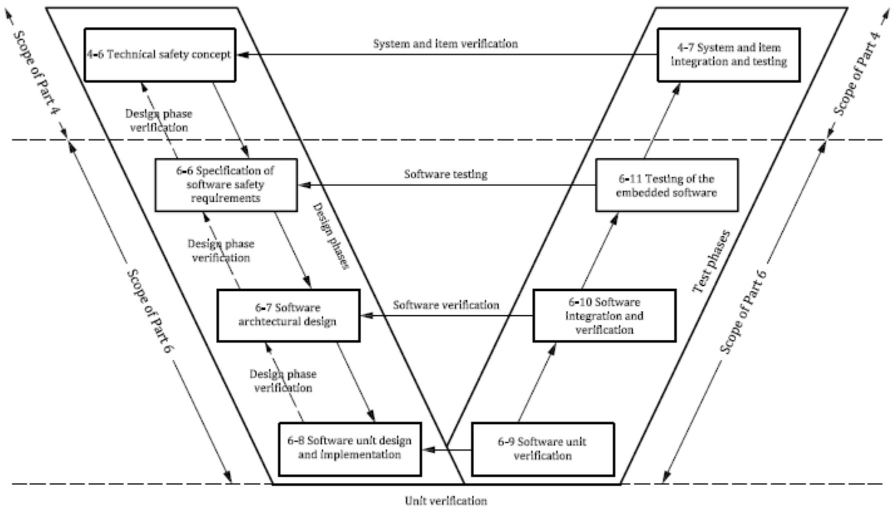

Scope of Part 4
4-6 Technical safety concept
System and item verification
4-7 System and item integration and testing
Scope of Part 4
Design phase verification
6-6 Specification of software safety requirements
Software testing
6-11 Testing of the embedded software
Design phases
Design phase verification
Test phases
6-7 Software archtectural design
Software verification
6-10 Software integration and verification
Scope of Part 6
Scope of Part 6
Design phase verification
6-8 Software unit design and implementation
6-9 Software unit verification
Unit verification

Figure 3.11: The V-model of software development according to ISO 26262-6[2]

3.4 Software Architecture (SA)

SA It is the collection of structures of software components, as well as their interactions and attributes, used to reason about a software system[11], During the design process, functional requirements and quality attributes are taken into consideration while making software architecture design decisions[26]. There are frequent challenges that software engineers face during software design, and to solve these problems, there are design solutions to these common problems that would reduce development costs as well as risks. These solutions are called architectural patterns[41].There are several prevalent architectural patterns, including the ones listed below,and for more details[9]:

- Layers.
- Pipes-Filters.
- Model View Controller (MVC).
- Broker.
- Microkernel.
- Blackboard.

Specific quality attributes (QAs) for each architectural design pattern vary from those of the other patterns based on the nature of the interaction between software components. One of the responsibilities of the software architect is to prioritize some qualities over others, and the process of prioritizing QAs over others is referred to as a trade-off.
In our research, we place more emphasis on patterns that are closely associated with safety qualities, such as the following architectural patterns:

- **Protected Single Channel.** In this pattern, errors are detected by examining the inputs and outputs as well as optional, and usually the application of this pattern is used to provide safety at the lowest cost[23].
- **Homogeneous Redundancy.**When we want to make the system more available even in the event that there is a fault with the primary channel, we will apply this pattern. The purpose behind this design is that in the event that errors are detected that prevent the work of the primary channel from continuing, the work will be switched to an alternate channel in order to assure the system's continued operation[23].

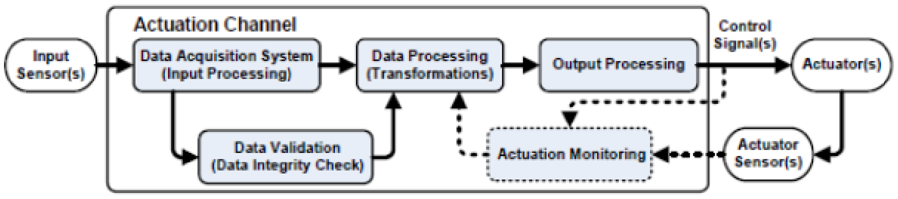


Figure 3.12: protected single channel [46]

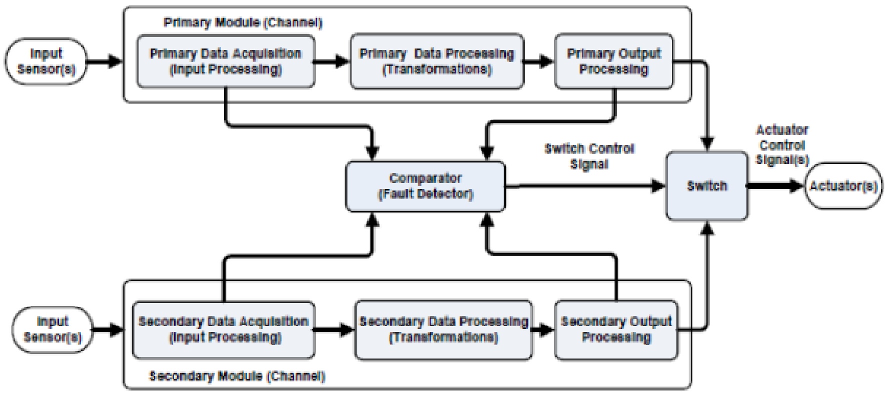

Primary Module (Channel)
Input Sensor(s)
Primary Data Acquisition (Input Processing)
Primary Data Processing (Transformations)
Primary Output Processing
Switch Control Signal
Actuator Control Signal(s)
Comparator (Fault Detector)
Switch
Actuator(s)
Input Sensor(s)
Secondary Data Acquisition (Input Processing)
Secondary Data Processing (Transformations)
Secondary Output Processing
Secondary Module (Channel)

Figure 3.13: Homogeneous/Heterogeneous redundancy [46]

- **Heterogeneous Redundancy.** This pattern is similar to the homogeneous pattern, but differs in that each channel develops independently of the other to ensure that the same mistakes are not repeated because the alternative channel in the homogeneous pattern is an exact copy of the main channel. Although this pattern is more reliable than the previous pattern, it is more expensive[46].

- **Safety Executive.**In some patterns, when a particular failure occurs, they treat it by shutting down the system so that the state of danger does not remain continuous. However, in this pattern, a shut down system may be very dangerous in the event of an error. In the safety executive pattern, there will be a series of complex procedures before reaching a state of fail-safe mode[23].the pattern is shown in figure 3.14.

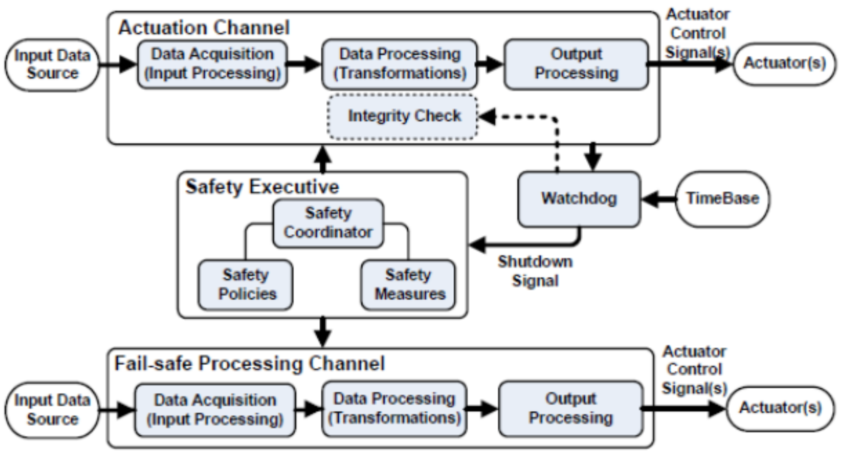


Figure 3.14: safety executive [46]

- **3-level Monitoring** This pattern consists of three levels: the first level monitors the system's internal state; the second level checks the inputs and outputs; and the third level monitors the system's functionality. This pattern provides a safe and economical solution, especially in the automotive industry.[46].

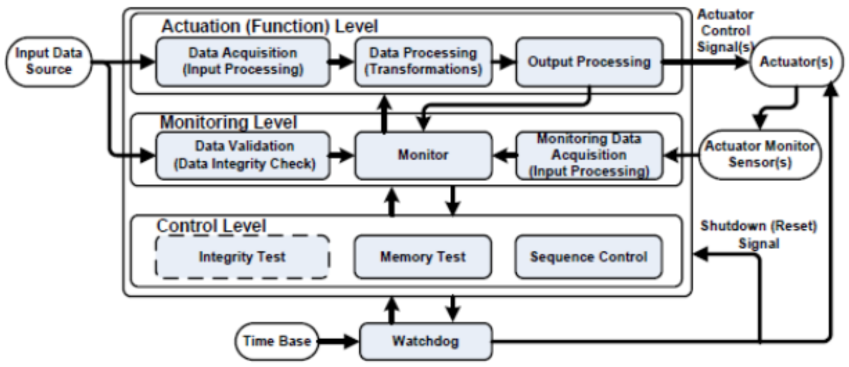

Actuation (Function) Level
Input Data Source
Data Acquisition (Input Processing)
Data Processing (Transformations)
Output Processing
Actuator Control Signal(s)
Actuator(s)
Monitoring Level
Data Validation (Data Integrity Check)
Monitor
Monitoring Data Acquisition (Input Processing)
Actuator Monitor Sensor(s)
Control Level
Integrity Test
Memory Test
Sequence Control
Shutdown (Reset) Signal
Time Base
Watchdog

Figure 3.15: Three level monitoring [46]

Chapter 4

Methodology

4.1 What is a Cruise Control System

A Cruise Control System (CCS) is one of the important systems in vehicles to help drivers control the speed of the vehicle while driving, especially over long distances. In this system, the driver can adjust the speed according to his desire and leave the pressure on the accelerator pedal, and then the system is an alternative to control the required speed without the need to press the accelerator pedal.
in addition to maintaining a constant velocity, this system accelerates or decelerates the vehicle without using the accelerator pedal.

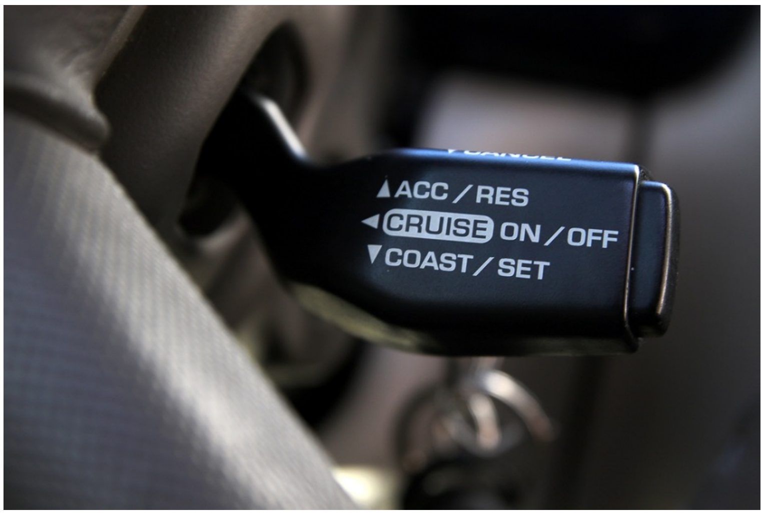


Figure 4.1: Cruise control system in the car

The ways in which the CCS is used vary based on the automobile manufacturer, but they all follow the same basic idea. As shown in the figure above, this system typically includes a number of buttons.

- on/off
  The driver uses it to activate or deactivate the CCS.
- ACC
  It is used to increase vehicle acceleration
- SET
  When you need to maintain the speed, this button is used
- Resume
  The CCS is immediately terminated in the cruise system if the brake pedal is depressed, therefore if the driver wishes to return to the last set speed, he will utilize this button.
- COAST
  It is used to decelerate the vehicle's speed

4.1.1 Automatic vehicle speed control(AVSC) software

A software that simulates the work of the CCS. Since there are issues in implementing the research to the actual cruise system in automobiles, this program will be refactored to be safer utilizing a new agile method proposed.the figure 4.2 represents the simulator software's user interface.
The simulator contains buttons similar to the work of the real system of cruise control in vehicles:

- *Arrancar/Apagar*
  Simulate engine start and shutdown.
- *Acelerar*
  used to boost a speed
- *Pisar Freno*
  When this button is pressed, the brake pedal action is simulated as the speed is slowed down and the speed limit is canceled
- *Parar*
  It is used to decelerate the speed
- *Mantener*
  This button set the speed to a specified level.
- *Reniciar*
  Returns to the previous speed level

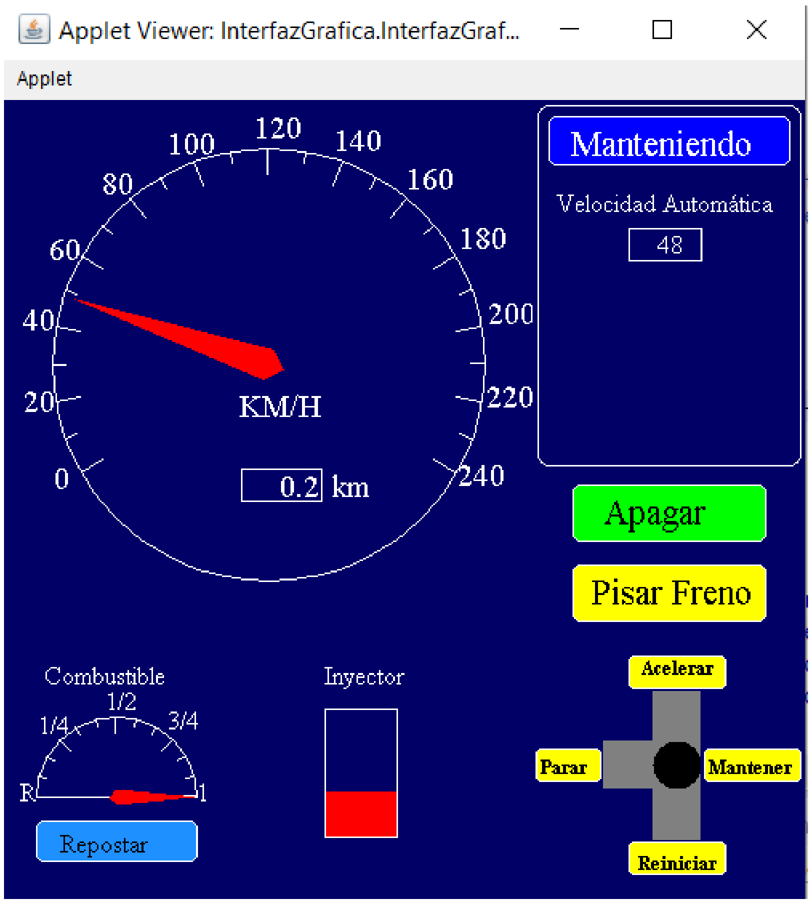


Figure 4.2: AVSC simulator

In addition, the simulator user interface displays the speedometer (gauge), the distance driven in kilometers per hour, the quantity of fuel injection, and the fuel consumption, and when the cruise is set, the selected speed is displayed.

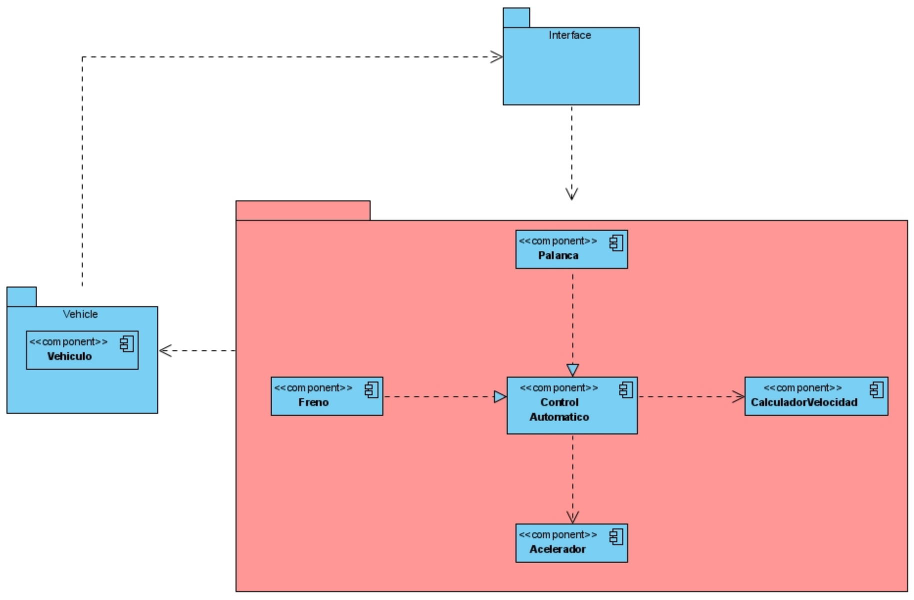


Figure 4.3: Components Diagram for simulator software

The simulator program includes three main packages as shown in the figure 4.3:

1. *ControlAutomatico*
   manages the automatic vehicle's speed control.

2. *InterfazGrafica*
   responsible for creating the applet that will run the application, is composed of four panels each in charge of a part of the vehicle, the speedometer, the lever, the automatic control panel and the fuel monitor and the injector. In addition, it initializes all the objects to be used in the application

3. *SimuladorVehiculo*
   Responsible for simulating the operation of the vehicle and sending vehicle data to the interface

So because automatic control package is in responsible for controlling the speed, the use case 4.4 demonstrates how it performs.

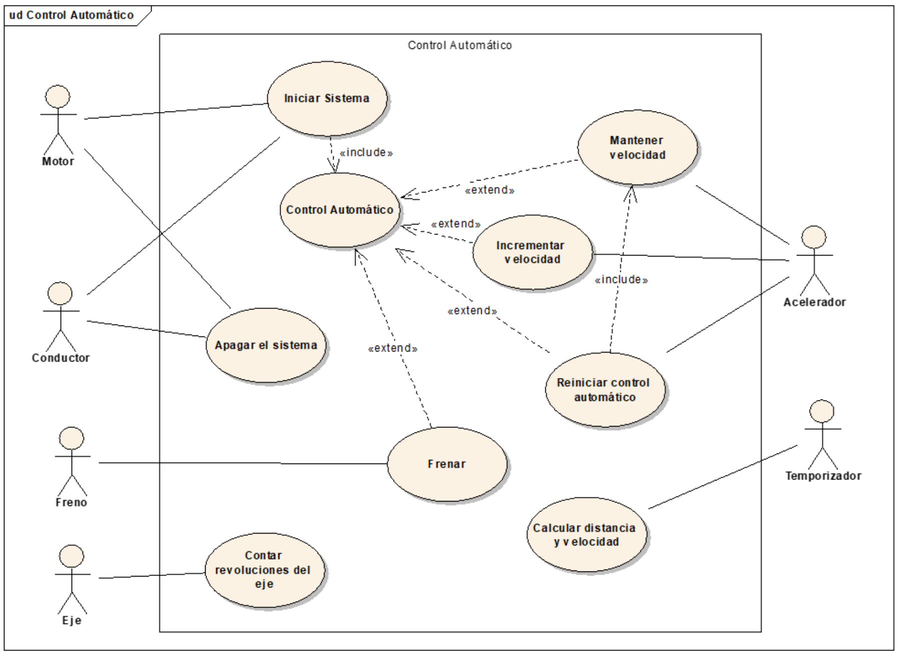


Figure 4.4: Use-case of speed automatic control

4.2 A proposed new agile approach for developing safety-critical systems

To develop SCS using an agile methodology, we have proposed a method that is used for the first time to the best of our knowledge, covering different phases of the development life cycle of this type of system, The following figure presents the general structure of the proposed approach.

This approach is divided into four major sections, which are described below:

1. Product backlog
   It is the first step in our proposed method because, as in the agile methodologies, the requirements to be developed in it are collected. In this step, we collect the functional and safety requirements but separately from each other. The authors[68, 32] advise to make this separation between requirements, safety requirements are usually clearer and more stable compared to functional requirements change over time. This helps us identify and emphasise safety requirements. The following phase will make more sense of the safety analysis of these requirements.

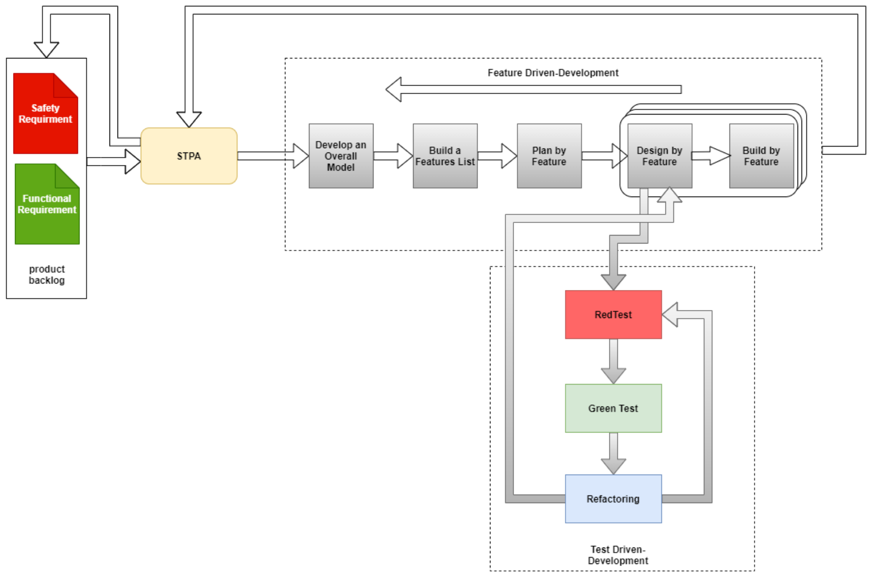


Figure 4.5: A proposed agile method

2. System-Theoretic Process Analysis (STPA)
At this stage, we analyze safety at the system level, which is an important phase in the life cycle of SCS to determine safety requirements. This phase receives system requirements, whether functional or non-functional, and analyzes the relationship between the various components of the system, whether hardware, software, agent, or environment. As a result of the hazard analysis, the product backlog will be provided with more comprehensive and accurate safety requirements. In our research, we preferred to use the (STPA) technology because it is a modern technology, it has been successfully applied in various fields such as automobiles, aviation, and medical systems[74], it gives good results when used in complex systems, and it can be used in various stages of the system development process, either at the beginning or at the end, in addition to using it with agile methodologies successfully as in Safe Scrum[75].

3. Test-Driven Development (TDD)
To boost productivity and enhance the design, we use TDD at this step before proceeding to the FDD's design phase[8, 13, 67]. As a result, it is better to conduct failure tests based on the causal factors that arise as a result of the implementation of STPA. As an outcome, the test-first approach improves the design, and failure in the system is avoided as early as feasible before the developers create the code.

4. Feature-Driven Development (FDD)

This phase is considered the initiation of the development process, during which we picked the FDD methodology as one of the agile approaches, with minor modifications as indicated in the figure 4.5, to address the requirements of developing SCS.
This method is suitable for large and complex systems, which is one of the features of SCS[53, 4]. In addition to the emphasis of this method on the design and building process, we also aim to focus in the development of software and production of a high-quality product[57], accompanied with good documentation.

4.3 Risk assessment test cases

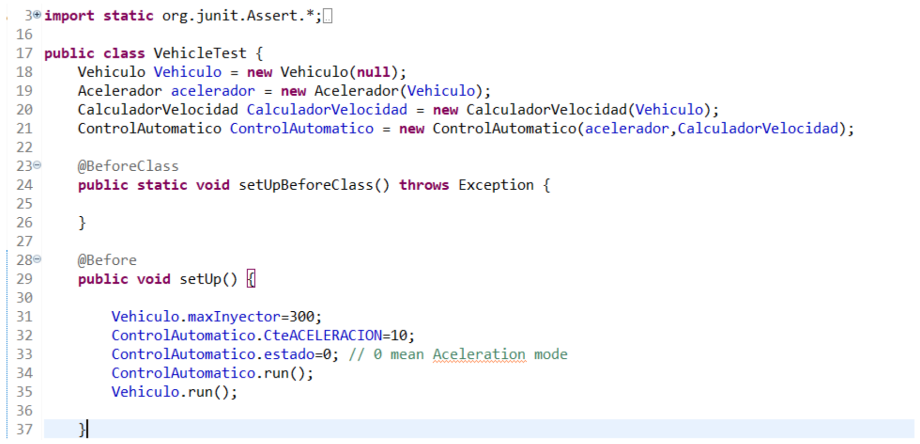
In this section, we will put the AVSC simulator to the test by introducing several potentially hazardous scenarios that could lead to violating safety requirements. Before we begin the tests, we would like to clarify some of the variables that affect speed control, based on their names in the code, as shown in the table below.

Variable name	Description
MaxInyector	The maximum amount of fuel injection is 100, which is also it's a default value.
CteACELERACION	represents the ratio at which we want to increase the maximum fuel input to the motor, and the default value is 5.
rozamientoSuelo	Represents the ground resistance to wheel rolling and its default value is 0.05.
rozamientoAire	Represents the air resistance and its default value is 0.004318.
velocidad	Represents the current vehicle speed.
velocidadAutomatico	Represents the value of the maintained speed.

Table 4.1: Variables that affect vehicle speed

4.3.1 Unsafe acceleration scenario

At first, We made a simple change in the original code so that we could carry out the tests. Results were documented only if entries were within acceptable limits. Then we entered variables and data outside the acceptable range to examine the change in the system's behavior and make sure that unsafe outputs occurred. We assumed the execution time was (10) seconds to check the behavior of the system during this period. Before proceeding with test cases, we will call the classes and methods that are necessary to program test cases and are repeated in most of them, the "Run" method in the four classes (ControlAutomatico,Vehiculo, Acelerador, CalculadorVelocidad) is executed before any test, as shown in the figure 4.6.



```
import static org.junit.Assert.*;

public class VehicleTest {
    Vehiculo Vehiculo = new Vehiculo(null);
    Acelerador acelerador = new Acelerador(Vehiculo);
    CalculadorVelocidad CalculadorVelocidad = new CalculadorVelocidad(Vehiculo);
    ControlAutomatico ControlAutomatico = new ControlAutomatico(acelerador,CalculadorVelocidad);

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {

    }

    @Before
    public void setUp() {

        Vehiculo.maxInyector=300;
        ControlAutomatico.CteACELERACION=10;
        ControlAutomatico.estado=0; // 0 mean Aceleration mode
        ControlAutomatico.run();
        Vehiculo.run();

    }
```

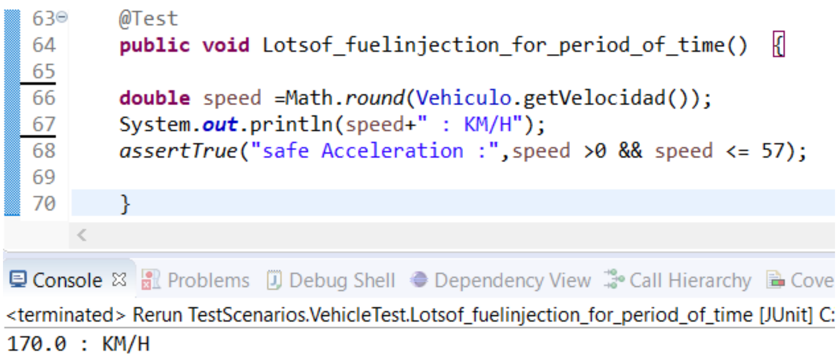
Figure 4.6: illustrate classes, methods, and variables called in the test case

In this scenario, we will have simulated unintended acceleration during the previously specified period (10s), and the vehicle's acceleration behavior was checked when:

1. **The maximum fuel injection is more than the specified value.**

Test case section	Details
Variables	MaxInyctor = 300
	CteACELERACION = 5
Test Description	In this test, we changed the Max value of the injection ("MaxInyctor") while fixing the amount of increase of the injection in a normal value ("CteACELERACION") to illustrate the impact of the maximum injection on the speed, as the normal max value of injection is 100 .
Test Result	We notice a rapid increase in acceleration in 10 s., i.e., the speed reaches 170 km / hour in only 10 s., while in safe mode the speed reached would have been 55 Km/h. A margin of error was imposed on the test case and, therefore, we assumed that if there is an increase of up to 2 units over the normal limit, we would not consider an unsafe state has been reached and safety test has failed.

Table 4.2: Details of Test Case 1



```
@Test
public void Lotsof_fuelinjection_for_period_of_time() {

double speed =Math.round(Vehiculo.getVelocidad());
System.out.println(speed+" : KM/H");
assertTrue("safe Acceleration :",speed >0 && speed <= 57);

}
```

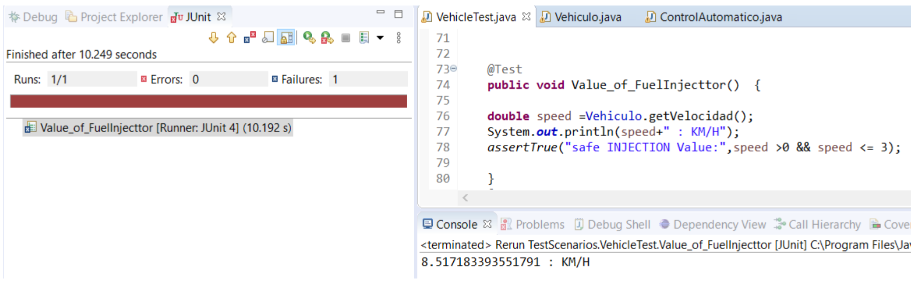
```
<terminated> Rerun TestScenarios.VehicleTest.Lotsof_fuelinjection_for_period_of_time [JUnit] C:
170.0 : KM/H
```

Figure 4.7: Running Test Case 1

2. **Injecting too much fuel in a specified period of time.**

Test case section	Details
Variables	MaxInyctor = 100
	CteACELERACION = 10
Test Description	In this test, we examine the effect of changing the value of the injector ("CteACELERACION"), whose safe value we know to be 5, and in the test we assign the value 10 while keeping the value of the max value of the injection in its normal position, for the purpose of checking the effect.
Test Result	The test failed because the speed exceeded the limit set, which is 3 km/h per second, this value was obtained in the normal situation, which is 5 for ("CteACELERACION")

Table 4.3: Details of Test Case 2



Finished after 10.249 seconds

Runs: 1/1 Errors: 0 Failures: 1

Value_of_FuelInjecttor [Runner: JUnit 4] (10.192 s)

```
    @Test
    public void Value_of_FuelInjecttor()  {

    double speed =Vehiculo.getVelocidad();
    System.out.println(speed+" : KM/H");
    assertTrue("safe INJECTION Value:",speed >0 && speed <= 3);

    }
```

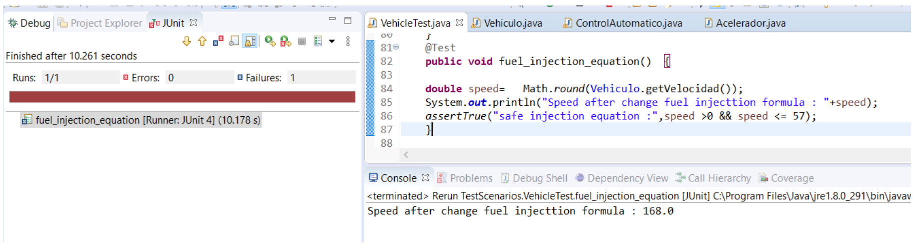
```
<terminated> Rerun TestScenarios.VehicleTest.Value_of_FuelInjecttor [JUnit] C:\Program Files\Ja
8.517183393551791 : KM/H
```

Figure 4.8: Running Test Case 2

3. **Change in the fuel mixing equation.**

Test case section	Details
Variables	MaxInyctor = 100
	CteACELERACION = 5
	Inyector * 0.01 To *0.03
Test Description	Changing the value multiplied by the value of the injector and its effect on the amount of acceleration.
Test Result	The test failed because a very high acceleration was produced, unintentionally. The test was first performed for a normal case and, therefore, a speed of 55 km/h. m. was obtained. By changing the equation, however, a speed value of 168 km/h. m was obtained, which was far from the speed value of the normal case.

Table 4.4: Details of Test Case 3



Finished after 10.261 seconds

Runs: 1/1 Errors: 0 Failures: 1

fuel_injection_equation [Runner: JUnit 4] (10.178 s)

```
    @Test
    public void fuel_injection_equation()  {

    double speed=   Math.round(Vehiculo.getVelocidad());
    System.out.println("Speed after change fuel injecttion formula : "+speed);
    assertTrue("safe injection equation :",speed >0 && speed <= 57);
    }
```

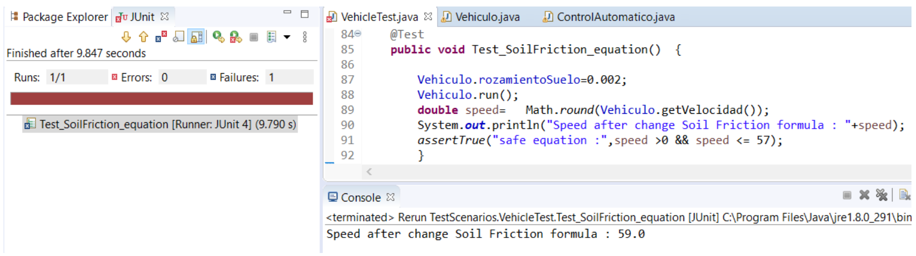
```
<terminated> Rerun TestScenarios.VehicleTest.fuel_injection_equation [JUnit] C:\Program Files\Java\jre1.8.0_291\bin\javav
Speed after change fuel injecttion formula : 168.0
```

Figure 4.9: Running Test Case 3

4. **Change in ground friction and their effect on acceleration.**

Test case section	Details
Variables	MaxInyctor = 100
	CteACELERACION = 5
	rozamientoSuelo = 0.002
Test Description	A decrease in the degree of friction with the ground leads to an increase in acceleration, according to the formula for calculating the velocity.
Test Result	The test failed due to a speed increase of 2 km/h from the normal limit

Table 4.5: Details of Test Case 4



```
Finished after 9.847 seconds
Runs: 1/1    Errors: 0    Failures: 1
Test_SoilFriction_equation [Runner: JUnit 4] (9.790 s)
```

```java
    @Test
    public void Test_SoilFriction_equation()  {

        Vehiculo.rozamientoSuelo=0.002;
        Vehiculo.run();
        double speed=   Math.round(Vehiculo.getVelocidad());
        System.out.println("Speed after change Soil Friction formula : "+speed);
        assertTrue("safe equation :",speed >0 && speed <= 57);
        }
```

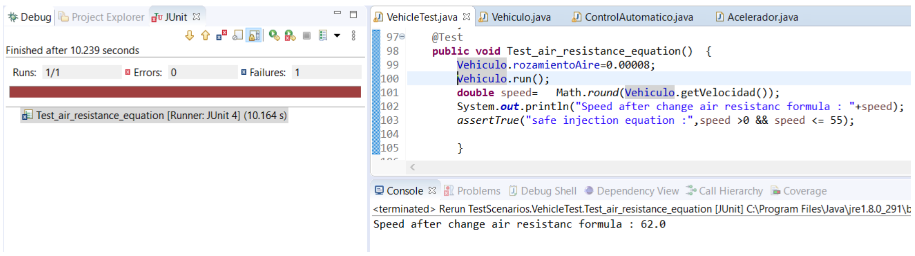
```
<terminated> Rerun TestScenarios.VehicleTest.Test_SoilFriction_equation [JUnit] C:\Program Files\Java\jre1.8.0_291\bin
Speed after change Soil Friction formula : 59.0
```

Figure 4.10: Running Test Case 4

5. **Change in air resistance and their effect on acceleration**

Test case section	Details
Variables	MaxInyctor = 100
	CteACELERACION = 5
	rozamientoAire = 0.00008
Test Description	A decrease in the coefficient of friction with the air causes an increase in acceleration, according to the formula for calculating the vehicle's speed.
Test Result	The test failed due to poor air resistance, resulting in unintended acceleration

Table 4.6: Details of Test Case 5


Finished after 10.239 seconds

Runs: 1/1 Errors: 0 Failures: 1

Test_air_resistance_equation [Runner: JUnit 4] (10.164 s)

```java
    @Test
    public void Test_air_resistance_equation()  {
        Vehiculo.rozamientoAire=0.00008;
        Vehiculo.run();
        double speed=   Math.round(Vehiculo.getVelocidad());
        System.out.println("Speed after change air resistanc formula : "+speed);
        assertTrue("safe injection equation :",speed >0 && speed <= 55);

        }
```

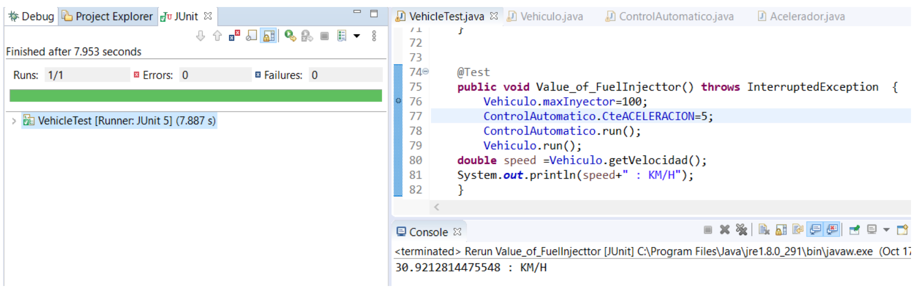
```
<terminated> Rerun TestScenarios.VehicleTest.Test_air_resistance_equation [JUnit] C:\Program Files\Java\jre1.8.0_291\b
Speed after change air resistanc formula : 62.0
```

Figure 4.11: Running Test Case 5

6. **Check X value of Acceleration(CteACELERACION).**

Test case section	Details
Variables	MaxInyctor = 100
	CteACELERACION = 10
Test Description	In this test, we examine the effect of changing the value of the injector ("CteACELERACION"), whose safe value we know to be 5, and in the test we assign the value10 while keeping the value of the max value of the injection in its normal position, for the purpose of checking the effect.
Test Result	Failure of the system behavior occurs when the injector value ("CteACELERACION") is raised to 10, which causes acceleration by two seconds less than allowable limit.

Table 4.7: Details of Test Case 6


Finished after 7.953 seconds

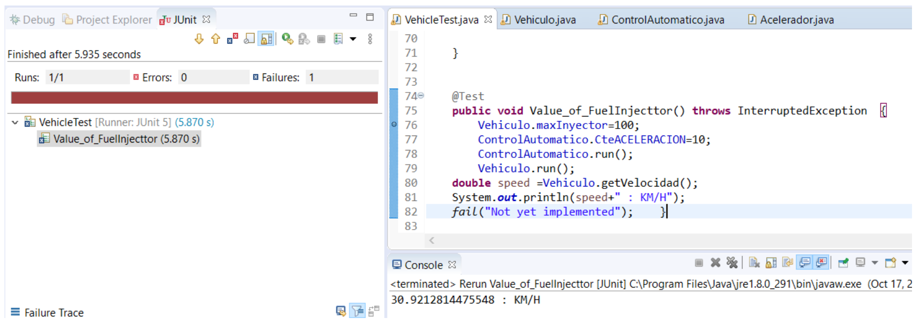
Runs: 1/1 Errors: 0 Failures: 0

VehicleTest [Runner: JUnit 5] (7.887 s)

```java
    }


    @Test
    public void Value_of_FuelInjecttor() throws InterruptedException  {
        Vehiculo.maxInyector=100;
        ControlAutomatico.CteACELERACION=5;
        ControlAutomatico.run();
        Vehiculo.run();
    double speed =Vehiculo.getVelocidad();
    System.out.println(speed+" : KM/H");
    }
```

```
<terminated> Rerun Value_of_FuelInjecttor [JUnit] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe  (Oct 17
30.9212814475548 : KM/H
```

Figure 4.12: Running Test Case 6 in normal state



```
Finished after 5.935 seconds
Runs: 1/1    Errors: 0    Failures: 1
VehicleTest [Runner: JUnit 5] (5.870 s)
    Value_of_FuelInjecttor (5.870 s)
```

```java
    }


    @Test
    public void Value_of_FuelInjecttor() throws InterruptedException {
        Vehiculo.maxInyector=100;
        ControlAutomatico.CteACELERACION=10;
        ControlAutomatico.run();
        Vehiculo.run();
    double speed =Vehiculo.getVelocidad();
    System.out.println(speed+" : KM/H");
    fail("Not yet implemented");    }

```

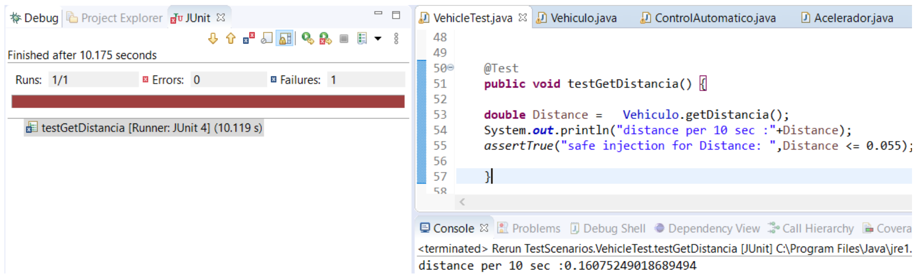
```
<terminated> Rerun Value_of_FuelInjecttor [JUnit] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe  (Oct 17, 2
30.9212814475548 : KM/H
```

Figure 4.13: Execution of the test case 6 in unsafe state

7. **Checking the distance traveled during a given period if we had not set the speed to any specific value.**

Test case section	Details
Variables	MaxInyctor = 300
	CteACELERACION = 15
Test Description	Check the distance if the injector value exceeds the normal limit.
Test Result	The result of this test is certainly an excessive increase in the distance traveled because the acceleration increases, another reason why this test was performed was to make sure that it also affects other functions of the program.

Table 4.8: Details of Test Case 7



```
Finished after 10.175 seconds
Runs: 1/1    Errors: 0    Failures: 1
testGetDistancia [Runner: JUnit 4] (10.119 s)
```

```java


    @Test
    public void testGetDistancia() {

    double Distance =   Vehiculo.getDistancia();
    System.out.println("distance per 10 sec :"+Distance);
    assertTrue("safe injection for Distance: ",Distance <= 0.055);

    }
```

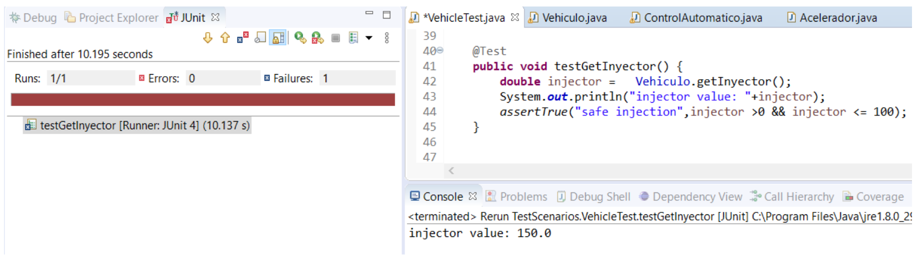
```
<terminated> Rerun TestScenarios.VehicleTest.testGetDistancia [JUnit] C:\Program Files\Java\jre1.
distance per 10 sec :0.16075249018689494
```

Figure 4.14: Running Test Case 7

8. **Check injector value**

Test case section	Details
Variables	MaxInyctor = 300
	CteACELERACION = 5
Test Description	Checking the current value of the injection after 10 seconds of execution.
Test Result	The test has failed because the value of the variable that controls the injection of the vehicle has exceeded the safety limit (100), this value being the upper limit that must not be exceeded.

Table 4.9: Details of Test Case 8



```
Finished after 10.195 seconds
Runs: 1/1    Errors: 0    Failures: 1
testGetInyector [Runner: JUnit 4] (10.137 s)
```

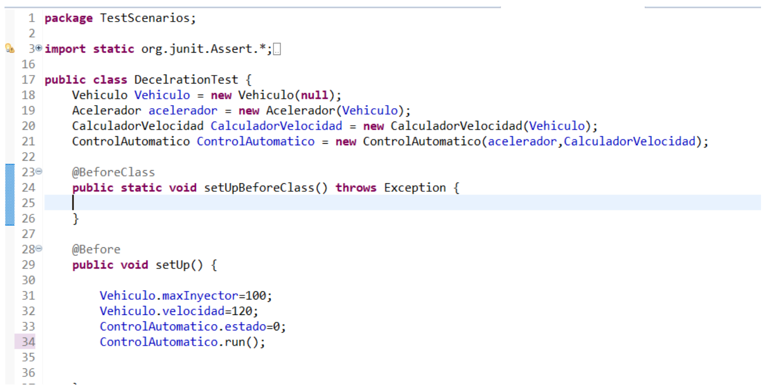
```java
    @Test
    public void testGetInyector() {
        double injector =   Vehiculo.getInyector();
        System.out.println("injector value: "+injector);
        assertTrue("safe injection",injector >0 && injector <= 100);
    }
```

```
<terminated> Rerun TestScenarios.VehicleTest.testGetInyector [JUnit] C:\Program Files\Java\jre1.8.0_2
injector value: 150.0
```

Figure 4.15: Running Test Case 8

4.3.2 Unsafe deceleration scenario

This scenario is similar to the previous one, but we will examine the factors that affect the speed deceleration, and the system is in acceleration mode. We assume that the speed of the vehicle is 180 km/h and tests will be done for a given period of time to show the effect ,
the classes, methods, and variables that were used before each test case was executed for deceleration as in 4.16 ,



```java
package TestScenarios;

import static org.junit.Assert.*;

public class DecelrationTest {
    Vehiculo Vehiculo = new Vehiculo(null);
    Acelerador acelerador = new Acelerador(Vehiculo);
    CalculadorVelocidad CalculadorVelocidad = new CalculadorVelocidad(Vehiculo);
    ControlAutomatico ControlAutomatico = new ControlAutomatico(acelerador,CalculadorVelocidad);

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {

    }

    @Before
    public void setUp() {

        Vehiculo.maxInyector=100;
        Vehiculo.velocidad=120;
        ControlAutomatico.estado=0;
        ControlAutomatico.run();

```

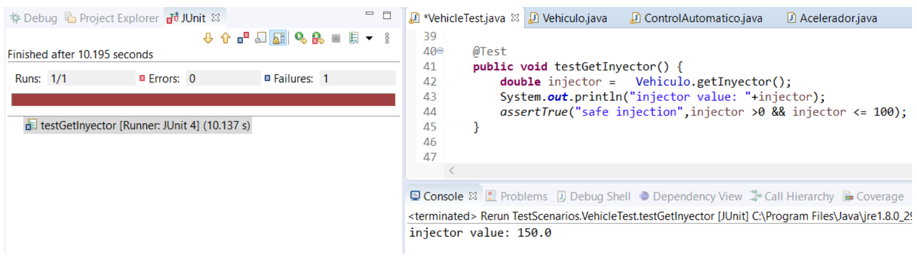
Figure 4.16: illustrate classes, methods, and variables called in the deceleration test scenario

deceleration behavior was checked when:

1. **Check the effect of air resistance**

Test case section	Details
Variables	MaxInyctor = 100
	velocidad = 120
	rozamientoAire = 0.009318 .
Test Description	To examine the effect of wind resistance when calculating the vehicle's speed in 25 s
Test Result	Test failed due to deceleration of more than 3 km

Table 4.10: air resistance Test case 9



```
Finished after 10.195 seconds
Runs: 1/1    Errors: 0    Failures: 1
testGetInyector [Runner: JUnit 4] (10.137 s)
```

```java
    @Test
    public void testGetInyector() {
        double injector =   Vehiculo.getInyector();
        System.out.println("injector value: "+injector);
        assertTrue("safe injection",injector >0 && injector <= 100);
    }
```

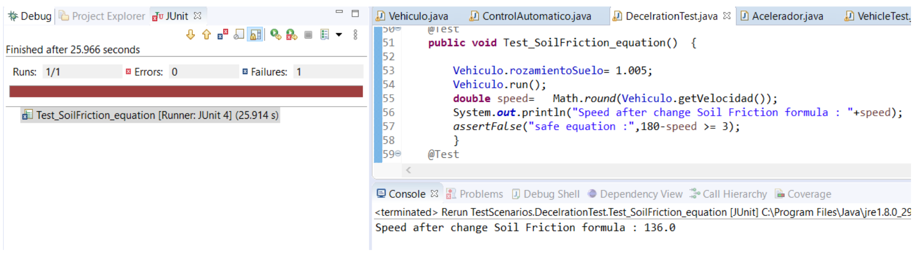
```
<terminated> Rerun TestScenarios.VehicleTest.testGetInyector [JUnit] C:\Program Files\Java\jre1.8.0_2
injector value: 150.0
```

Figure 4.17: Running air resistance Test Case 9

2. **Check the effect of ground resistance**

Test case section	Details
Variables	MaxInyctor = 100
	velocidad = 120
	rozamientoSuelo = 1.005
Test Description	To check the effect of ground resistance on the vehicle's forward speed calculation in 25 sec
Test Result	Test failed due to deceleration of more than 3 km

Table 4.11: Ground resistance Test Case 10



```
Finished after 25.966 seconds
Runs: 1/1    Errors: 0    Failures: 1
Test_SoilFriction_equation [Runner: JUnit 4] (25.914 s)
```

```java
    @Test
    public void Test_SoilFriction_equation()  {

        Vehiculo.rozamientoSuelo= 1.005;
        Vehiculo.run();
        double speed=   Math.round(Vehiculo.getVelocidad());
        System.out.println("Speed after change Soil Friction formula : "+speed);
        assertFalse("safe equation :",180-speed >= 3);
        }
    @Test
```

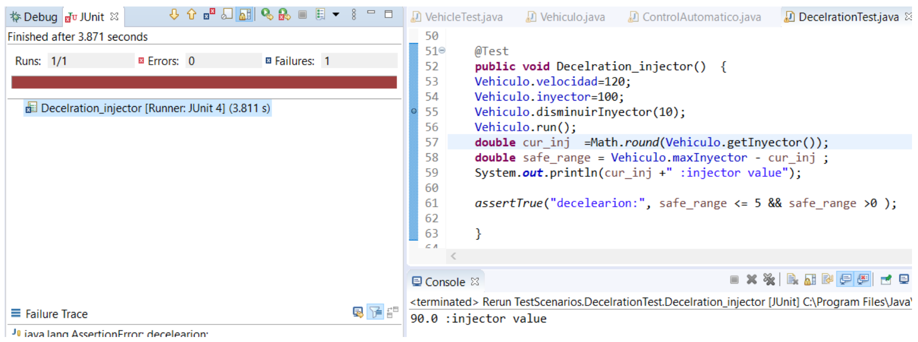
```
<terminated> Rerun TestScenarios.DecelrationTest.Test_SoilFriction_equation [JUnit] C:\Program Files\Java\jre1.8.0_29
Speed after change Soil Friction formula : 136.0
```

Figure 4.18: Running ground resistance Test Case 10

3. **Check if the fuel injection value is too low**

Test case section	Details
Variables	MaxInyctor = 100
	velocidad = 120
	CteACELERACION = 10
Test Description	This test is used to test the effect of the injector variable value being greater than required during deceleration, this test used a Vehiculo.disminuirInyector(CteACELERACION) a method responsible for the slowdown of speed
Test Result	The test failed because the difference between the maximum allowable fuel injection value and the current injection value exceeded 5, which is the normal fuel injection value in the program.

Table 4.12: Fuel injection test case 11 is less than required



```
Finished after 3.871 seconds
Runs: 1/1   Errors: 0   Failures: 1
Decelration_injector [Runner: JUnit 4] (3.811 s)
```

```java
    @Test
    public void Decelration_injector()  {
    Vehiculo.velocidad=120;
    Vehiculo.inyector=100;
    Vehiculo.disminuirInyector(10);
    Vehiculo.run();
    double cur_inj  =Math.round(Vehiculo.getInyector());
    double safe_range = Vehiculo.maxInyector - cur_inj ;
    System.out.println(cur_inj +" :injector value");

    assertTrue("decelearion:", safe_range <= 5 && safe_range >0 );

    }
```

```
<terminated> Rerun TestScenarios.DecelrationTest.Decelration_injector [JUnit] C:\Program Files\Java\
90.0 :injector value
```

```
java.lang.AssertionError: decelearion:
```

Figure 4.19: Execution of an inadequate fuel injection test case 11.

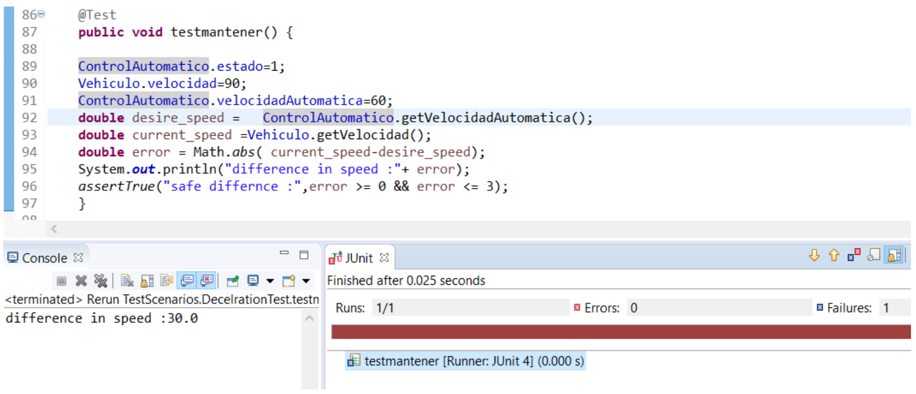
4.3.3 Unsafe scenario when cruise control

In this section, we'll consider that the system is in cruise control mode and that there has been an unintentional acceleration or deceleration of the vehicle's speed.

1. **Unsafe acceleration**

Test case section	Details
Variables	velocidad = 90
	velocidadAutomatico = 60
Test Description	This test reveals any unintentional acceleration that occurs while using cruise control.
Test Result	The test case failed because there is a significant difference between the set speed and the vehicle's actual speed.

Table 4.13: Test case 12 illustrate unintended acceleration during maintained speed



```java
@Test
public void testmantener() {

ControlAutomatico.estado=1;
Vehiculo.velocidad=90;
ControlAutomatico.velocidadAutomatica=60;
double desire_speed =  ControlAutomatico.getVelocidadAutomatica();
double current_speed =Vehiculo.getVelocidad();
double error = Math.abs( current_speed-desire_speed);
System.out.println("difference in speed :"+ error);
assertTrue("safe differnce :",error >= 0 && error <= 3);
}
```

```
<terminated> Rerun TestScenarios.DecelrationTest.testn
difference in speed :30.0
```

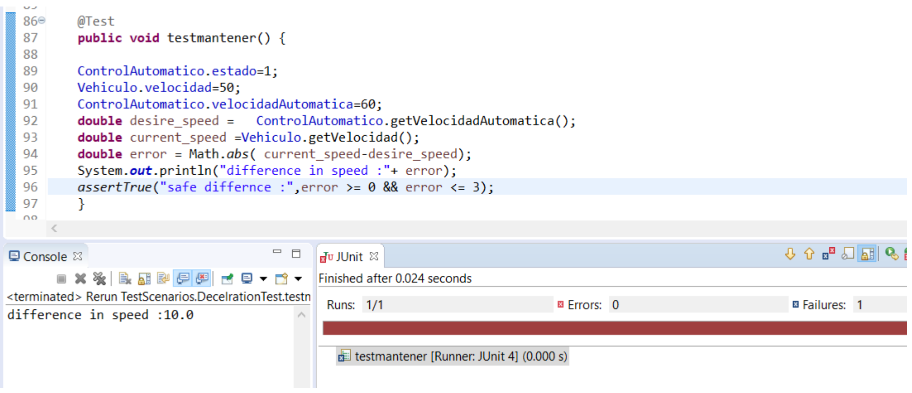
```
Finished after 0.025 seconds
Runs: 1/1   Errors: 0   Failures: 1
testmantener [Runner: JUnit 4] (0.000 s)
```

Figure 4.20: running the test case 12 for unexpected acceleration with the cruise control engaged.

2. **Unsafe deceleration**

Test case section	Details
Variables	velocidad = 50
	velocidadAutomatico = 60
Test Description	This test reveals any unintentional deceleration that occurs while using cruise control.
Test Result	The test case failed because there is a significant difference between the set speed and the vehicle's actual speed.

Table 4.14: Test case 13 illustrates unintended deceleration during maintained speed



```java
@Test
public void testmantener() {

ControlAutomatico.estado=1;
Vehiculo.velocidad=50;
ControlAutomatico.velocidadAutomatica=60;
double desire_speed =  ControlAutomatico.getVelocidadAutomatica();
double current_speed =Vehiculo.getVelocidad();
double error = Math.abs( current_speed-desire_speed);
System.out.println("difference in speed :"+ error);
assertTrue("safe differnce :",error >= 0 && error <= 3);
}
```

```
<terminated> Rerun TestScenarios.DecelrationTest.testn
difference in speed :10.0
```

```
Finished after 0.024 seconds
Runs: 1/1   Errors: 0   Failures: 1
testmantener [Runner: JUnit 4] (0.000 s)
```

Figure 4.21: running the test case 13 for unexpected deceleration with the cruise control engaged.

4.4 Implementation of the Proposed Method

In this part, we will implement our method, taking into consideration the ISO-26262 requirements.

4.4.1 Product backlog

In this backlog, we will collect functional requirements and safety requirements independently to simplify the process of tracing safety requirements and conducting a hazard analysis. The requirements will be defined using the Easy Approach to Requirements Syntax (EARS) standard. see[47]For details, The requirements are collected according to what is available or obtained from the customer, and it is not necessary to specify which of the requirements will be worked on in advance or to determine the team that will accomplish them because when applying the section of the FDD there is another backlog responsible for this thing.

Functional Requirements

The purpose of functional requirements is to identify what tasks must be completed by the system and its behavior in order to correctly perform the specified functions. These requirements are frequently modified by the customer. In the following table 4.15, we will list a few of the system's most important essential needs.

Functional Requirements No	Description
FR-1	FR-1.1 When <press> the <accelerator> the vehicle shall be <have an acceleration >
	FR-1.2 While <acceleration> the accelerator shall < keep the acceleration > until the mode is changed by the driver
	FR-1.3 The <accelerator >shall <stop accelerating> when the <brakes are applied>
FR-2	FR-2.1 The <maintainer> shall <keep the speed >at the same average
	FR-2.2 When <maintainer is pressed>, the <current speed> shall be saved
	FR-2.3 The <maintainer >shall <stop keep cruising> when the <brakes> are applied
FR-3	FR-3.1 The <resume> shall <return >to last saved speed
	FR-3.3 when The <accelerator pressed> the <resume>shall <canceling>

Table 4.15: Functional Requirements table

Safety Requirements

Generally, safety goals are more constant, although the specifications of these requirements might be challenging. According to ISO-26262, technical safety requirements, the system design architecture, and the hardware-software interface (HSI) should be addressed as prerequisites, As a result, the method of safety analysis (STPA) will assist us in identifying these requirements, and as depicted in the figure 4.5, In the following table, we specify the initial safety goals and relate the most relevant safety requirements with safety objectives to facilitate comprehension and make it simpler to trace and verify compliance with the requirements.

Safety Goals	How to achieve the safety goal
SG-1 Ensure that there is no unintended acceleration or deceleration	Monitor the system in case of unintended acceleration or deceleration.
	maintain the safe speed cruising
	notify the driver when fault occurs

Table 4.16: Safety Goal table

Safety Requirements (SR) for SG1	description
SR1	The <actuator> shall <run> from one program
SR2	The <actuator> signal shall have a<range check>
SR3	while a <signal delayed> shall <send >error code
SR4	The <speed> value shall <send> from one program
SR5	while a <speed value> < delayed> shall <send >error code
SR6	The <current speed> value shall have a <Compared> with a set speed
SR7	While the <current speed> is <delayed>, an error signal should be <sent>

Table 4.17: safety requirments table

ASIL Determination

According to the ISO-26262 standard, risks are classified into four categories, beginning with the lowest degree of risk (A) and progressing to the highest degree (D) Hazards that are identified as QM do not dictate any safety requirements". There are three specific variables to consider when determining the classification for the safety goal and safety requirements, namely:

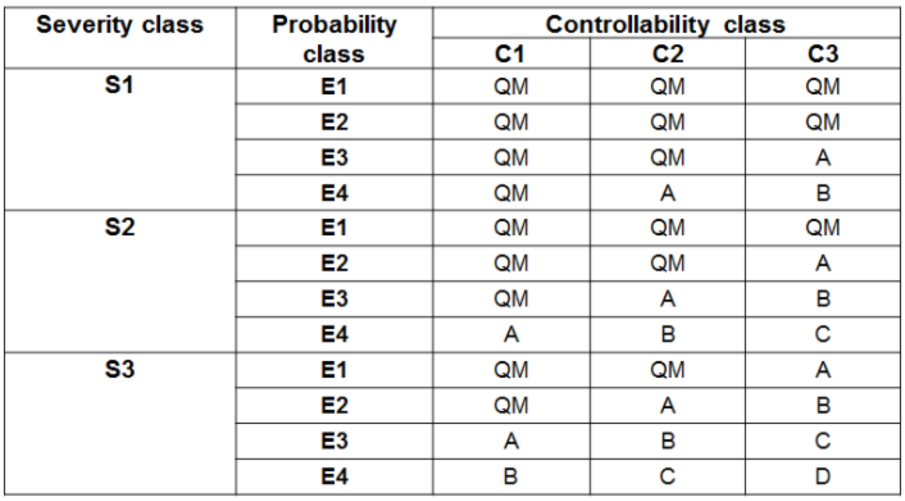
- Severity: Indicates the level of survival in the event of a system failure. It is classified into four levels:
  1. S0: no injury .
  2. S1: mild and moderate injuries .
  3. S2: severe and potentially fatal injuries, with a good chance of survival
  4. S3:life-threatening injuries with a low chance of survival, as well as fatal injuries .
- Exposure: It reflects the probability that a hazard will occur.
  1. E0: It is extremely unlikely to occur.
  2. E1 : Very little chance to occur.
  3. E2: Low probability.
  4. E3: Medium probability.

5. E 4: High probability.

- Controllability : The ability to avoid risk by controlling failure, whether through human intervention or system control.

1. C0: controllable in general.
2. C1: simply controllable.
3. C2: normally controllable.
4. C3 : Uncontrollable .

The table below demonstrates how to determine the ASIL level in accordance with ISO 26262.



Severity class	Probability class	Controllability class		
		C1	C2	C3
S1	E1	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	E1	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	E1	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Figure 4.22: ASIL determination table [51]

Referring to the previous table, we will determine the level of ASIL for each of the safety goal and safety requirements.

Safety Goals No	Severity	Exposure	Controllability	ASIL
SG-1	S2	E3	C3	ASIL B

Table 4.18: Safety goal ASIL determination

Safety requirments No	Exposure	Sever	Controlability	ASIL
SR1	E2	S1	C1	QM
SR2	E3	S2	C1	A
SR3	E3	S2	C3	B
SR4	E2	S1	C1	QM
SR5	E3	S2	C3	B
SR6	E4	S2	C2	B
SR7	E3	S2	C3	B

Table 4.19: Safety requirments ASIL determenation

4.5 Implementation of STPA

To initiate the analysis process utilizing STPA technology, with a general description of the system and an understanding of the components and their interactions, including agents. The system is typically described by displaying documents or explanations by specialists, resulting in the creation of a control structure diagram that simulates how the control of system components performs, The following figure shows a simplified diagram of the control structure of the cruise system

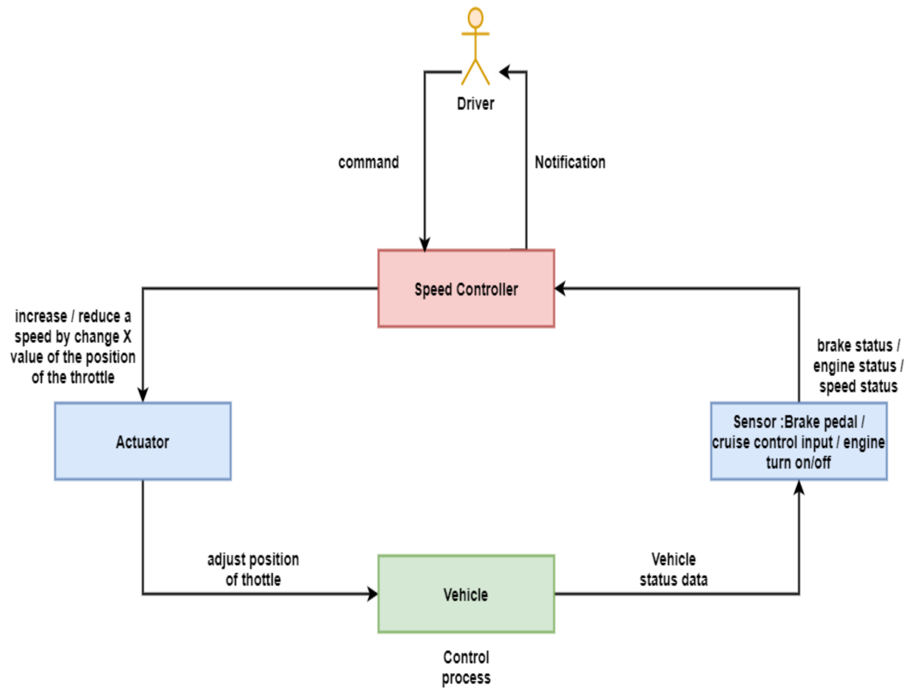

Driver
command
Notification
Speed Controller
increase / reduce a speed by change X value of the position of the throttle
brake status / engine status / speed status
Actuator
Sensor :Brake pedal / cruise control input / engine turn on/off
adjust position of thottle
Vehicle
Vehicle status data
Control process

Figure 4.23: ccs control structure diagram

In this control structure, to comprehend how to control the speed, the driver sends commands to set the speed, the cruise control unit sends the appropriate signal to the actuator, who adjusts its position based on the inputs to it, the vehicle control unit handles the changes and sends the updated data to the speed control unit to check if the current speed equal to desired speed , and then notifies the driver of the current state.

4.5.1 Determine unsafe control action(UCA) and associated reasons.

In this section, we identify the actions that could lead to the hazards and the accidents that result from these actions.

Control actions

Referring to the figure 4.23, we will determine which control action (CA) lead to the risks:
CA1:Sending an X-value signal to the actuator from the automatic speed control unit
CA2:Send the amount of current speed
CA3:Compared the actual speed to the desired speed

Possible accidents as a result of unsafe actions

AC1:vehicle can collide with a vehicle in front of it, a barrier or even a pedestrian.
AC2:The possibility of colliding with another car from behind

Hazards predicted as a result of control actions

H1:Unintended acceleration of the vehicle when the speed is set, will lead to AC1.
H2:Unintended deceleration of the vehicle when the speed is set,will lead to AC2

Unsafe Control Action(UCA)				
control action	provided	not provided	Provided too(early/late)	provided too(long/soon)
CA1	UCA1:not required providing, lead to H1.	UCA2: required not providing, lead to H2.	UCA3: too late lead to H2 | too early :N/A	UCA4: too long lead to H1 Especially if signal it is out of range
CA2	UCA5:not required providing, lead to H1 or H2	UCA6: required not providing, lead to H2	UCA7: too late lead to H2 | too early :N/A	UCA8: too long lead to H1 especially if a value it is out of range
CA3	UCA9:Available but there is a big difference between the current speed and the desired speed, will lead to H1 or H2	UCA10: required not providing, lead to H1,H2 Depending on the current speed	UCA11: too late lead to H2 , H1 Depending on the current speed | too early :N/A	UCA12: too long:N/A | stooped to soon could lead to H1,H2

Table 4.20: STPA first step

4.5.2 Occasional factors

The technique in the next step evaluates the possible reasons and related causal scenario, as illustrated in table 4.21:

UCA No	crucial factors (CF)
UCA1	CF1 Send a signal to the same Actuator from another program
UCA2	CF2 Send a value equal to zero, or well below the range
UCA3	CF3 A hardware malfunction occurs in the signal transmission
UCA4	CF4 The signal is out of range
UCA5	CF5 sending data from another program
UCA6	CF6 data equal to 0 or not readable
UCA7	CF7 A hardware malfunction occurs in the signal transmission
UCA8	CF8 data out of range
UCA9	CF9 The influence of external factors that lead to significant acceleration or deceleration (resistance of ground or air, malfunction outside the speed control unit)
UCA10	CF10 A hardware issue with the data transfer or a significant delay transfer current speed
UCA11	CF11 significant delay transfer current speed
UCA12	CF12 significant delay transfer current speed

Table 4.21: STPA second step

Constraints of safety

It is considered the last step of the STPA technology, in which we define the constraints that avoid or mitigate unsafe control actions from occurring and from which we can derive the safety requirements,The safety constraints in the table4.22 will be considered for transformation into safety requirements in Section 4.4.1 .

crucial factors NO	Safety constraints
CF1	Sending the control signal must be from one program
CF2	The signal range must be checked
CF3	monitor the timing of sending the signal
CF4	check signal range
CF5	The data must be sent from one program
CF6	check information range
CF7	monitor the timing of sending the signal
CF8	check information range
CF9	Monitor current speed with required speed every specified time period
CF10	Monitor data over time
CF11	Monitor data over time
CF12	Monitor data over time

Table 4.22: safety constraints for UCA

4.6 Application of the FDD

In this part, we will implement the FDD method, taking into account the outputs of the previous stages. We will use the ETVX(Entry,Tasks,Verification,Exit) template to formalize and clarify the implementation of tasks.

Section	Description
Entry	A quick description of the procedure and a list of requirements that must be satisfied before the phase can begin.
Tasks	A task list must be performed as part of this procedure
Verification	The method used to verify the outputs and whether the criteria have been achieved.
Exit	The outputs obtained

Table 4.23: ETVX template

4.6.1 Develop an overall model

Section	Description
Entry	The inputs for this stage will be the requirements in the product backlog. It is acceptable to follow the same traditional steps to build the overall model. However, it is preferable that the requirements be subjected to a safety analysis to help us determine more accurate safety requirements so that the work team takes these requirements into account while developing the overall models. The constraints imposed by ISO 26262 must be taken into account.Those responsible for this step are domain experts, chief programmers, and chief architects.
Tasks	1. Reviewing the proposed system. 2. Separate the system into domains. 3. Mapping each domain in accordance with the proposed safety requirements. 4. Safety requirements are subject to ISO 26262 regulations. 5. To create a simplified model for each domain. 6. Then assemble these subsections into an overall model.
Verification	Verify that the model complies with safety requirements and ISO 26262 regulations.
Exit	1. Overall model. 2. Basic class diagram. 3. Notes on why the current overall model was chosen.

Table 4.24: First phase template

When reviewing how the cruise control system works, we can divide the system into two main domains. The first domain is where the actuator controls the vehicle speed 4.24, and the second domain is where the value of the current speed is sent to the automatic speed control unit4.25, where it is compared to the speed at which the acceleration is required to be set.

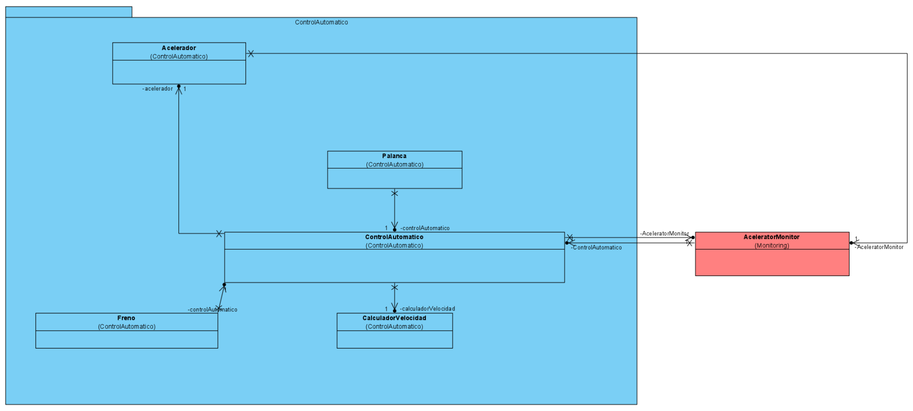


Figure 4.24: monitor acceleration domain

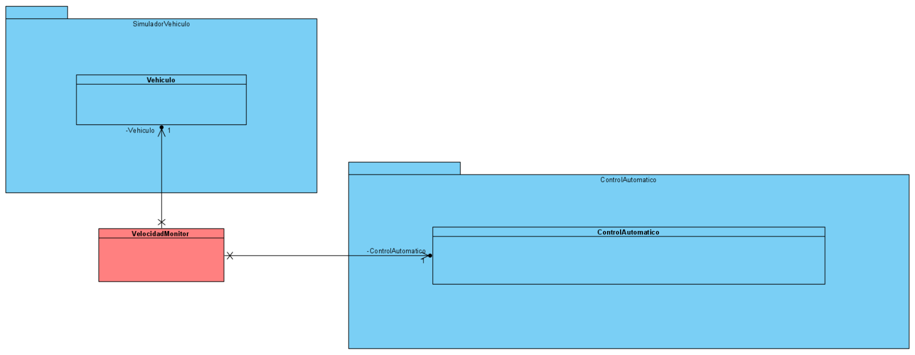


Figure 4.25: speed monitor domain

In this simplified class diagrams, we have added the components for monitoring the signal that may be out of the safe range, and these components are highlighted in red to distinguish them from the functional components,The safety requirements that we have obtained are in compliance with ISO 26262-6 7.4.12 safety mechanisms recommendations through error detection and error handling. The next step is to unify the previous two domains into a single, comprehensive model4.26.

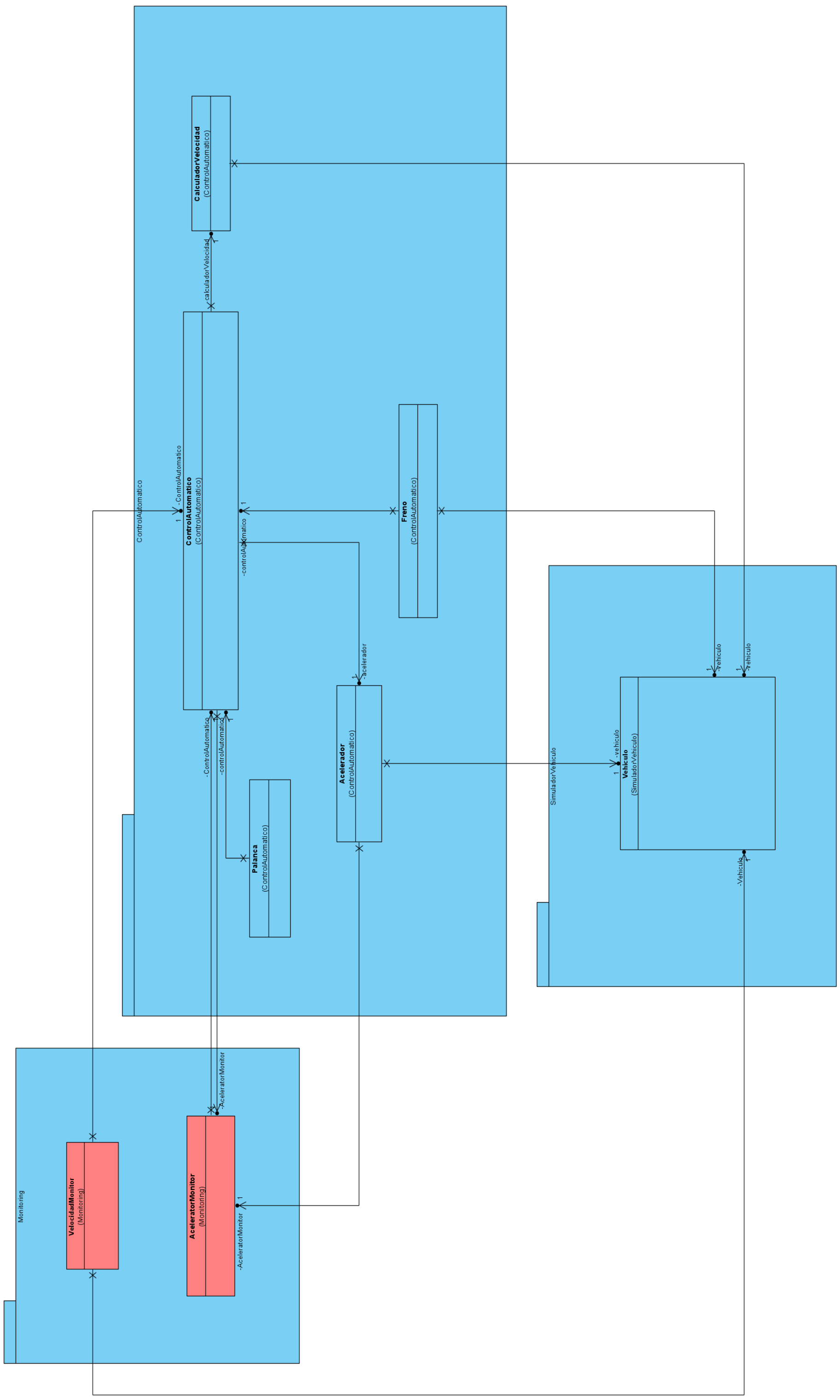


Figure 4.26: overall modeling

The previous model represents the general outline of the system, and we note that the software components responsible for safety are isolated and do not interfere with the work of the functional system; this is what is recommended in ISO 26262; To verify that the model meets safety requirements, Table 4.25 illustrates the matter. through this overall model, the features will be derived.

Safety requirements No	model validation notes
SR1	According to the model, "Acelerador" will be only responsible for the actuator
SR2	acelerator Monitor has a range check
SR3	acelerator Monitor has a range check and sends an error code if there is a signal delayed
SR4	According to the model, "Vehiculo" will be only responsible for sending speed value
SR5	According to the model, "VelocidiadMonitor" will check the range and send an error code if there is a delayed signal.
SR6	According to the model, "VelocidiadMonitor" will compare a current speed with a maintained speed.
SR7	According to the model, "VelocidiadMonitor" will compare a current speed with a maintained speed and send an error code if there is a delayed signal.

Table 4.25: Verification of overall model

4.6.2 Build a Features list

Section	Description
Entry	an Overall Model
Tasks	A group of Chief Programmers typically forms a team to analyze the functional decomposition of the domain based on the partitioning previously done by Domain Experts. The team divides the domain into several major areas, which are then divided into smaller sets of activities. Each activity is further divided into individual features. This process results in a hierarchical list of features,Emphasis will be placed on areas of concern in the domain of safety.
Verification	The team evaluates the list of features that has been produced, either internally or through external assessment, considering the approval of the domain expert on these list of features.
Exit	The team generates a list of features organized into categories, beginning with major feature sets (areas) and proceeding to feature sets (activities) and individual features within those activities

Table 4.26: second phase template

Feature No	Major Feature set	Feature set	Feature Name
F-1	Monitoring	Acceleration monitoring	Check the safe range of acceleration
F-2	Monitoring	Acceleration monitoring	Sending an error code to the speed control unit in case of failure
F-3	Monitoring	velocity monitoring	Compare current speed with the set speed value
F-4	Monitoring	velocity monitoring	Sending an error code to the speed control unit in case of failure

Table 4.27: features List table

4.6.3 Plan by Feature

Section	Description
Entry	The features list
Tasks	1. The project planning group includes the Project Manager, the Development Manager, and the Chief Programmers. 2. Task the Chief Programmers with specific feature sets. 3. Assign developers to specific classes.
Verification	The Project Manager, Development Manager, and Chief Programmers can assess the progress and effectiveness of the project through their active participation in the planning process. By engaging and using their expertise, these individuals can conduct a self-assessment and make informed decisions.
Exit	1. The chief programmers are responsible for specific feature sets. 2. List of class owners (developers). 3. A schedule indicating the target completion dates for major features (by month and year), feature sets (by month and year), and features (by week).

Table 4.28: third phase template

Monitoring as (Major Feature)					
Feature No	Feature set	Feature name	from date	to date	class name / developer
F-1	Acceleration Monitoring	Check the safe range of acceleration	1/7	7/7	AceleratorMonitor / Zain
F-2	Acceleration Monitoring	Sending an error code to the speed control unit in case of failure	8/7	9/7	AceleratorMonitor / Zain
F-3	Velocity Monitoring	Compare current speed with the set speed value	10/7	17/7	VelocidadMonitor /Zain
F-4	Velocity Monitoring	Sending an error code to the speed control unit in case of failure	18/7	25/7	VelocidadMonitor /Zain

Table 4.29: timetable of the plan by feature

4.6.4 Design by Feature

In the design stage, we must take into account the safety goal that must be achieved. From the above safety requirements, the following GSN diagram 4.27 facilitates an understanding of how to achieve that. In this phase, the Chief Programmer is in charge of organizing the development of features by selecting them from a group of assigned features and dividing them into smaller groups. They also assemble teams of developers to work on specific features, and the teams create in-depth diagrams for the chosen features. The chief programmer then reviews and updates the object model based on the diagrams, and the developers create class and method summaries. A verification of the design is also conducted.

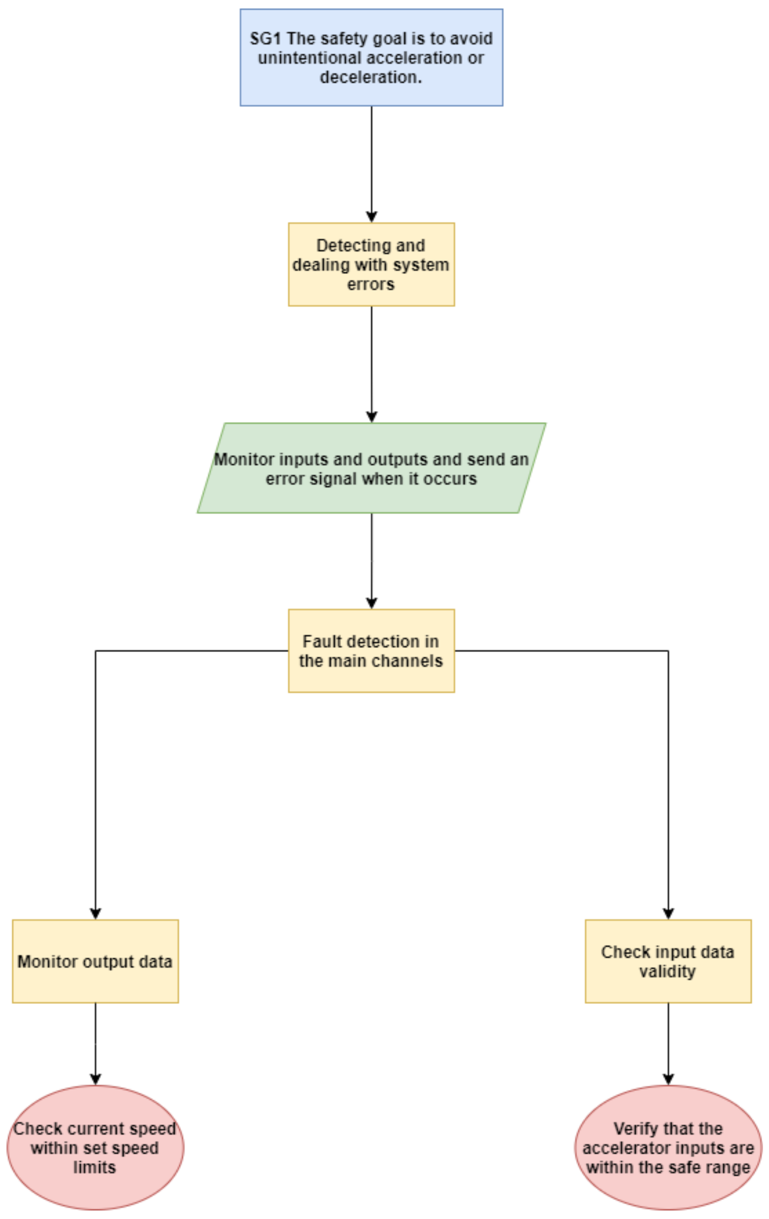

SG1 The safety goal is to avoid unintentional acceleration or deceleration.
Detecting and dealing with system errors
Monitor inputs and outputs and send an error signal when it occurs
Fault detection in the main channels
Monitor output data
Check input data validity
Check current speed within set speed limits
Verify that the accelerator inputs are within the safe range

Figure 4.27: GSN diagram of safety goal

Section	Description
Entry	The FDD phase three has been successfully finished by the planning team.
Tasks	1. create sequence diagrams for features. 2. create a design according to ISO 26262-6 7.4.3 recommendation. 3. update an overall model
Verification	Verify the design according to ISO 26262-6 7.4.14
Exit	1. sequence diagrams . 2. Alternative design according to safety and ISO 26262 requirements. 3. An updated overall object model with classes, methods.

Table 4.30: design phase template

Table 4.29 will be employed to generate a sequence diagram for each feature:

- The following sequence diagram shows how to monitor unintended acceleration for feature (F-1).

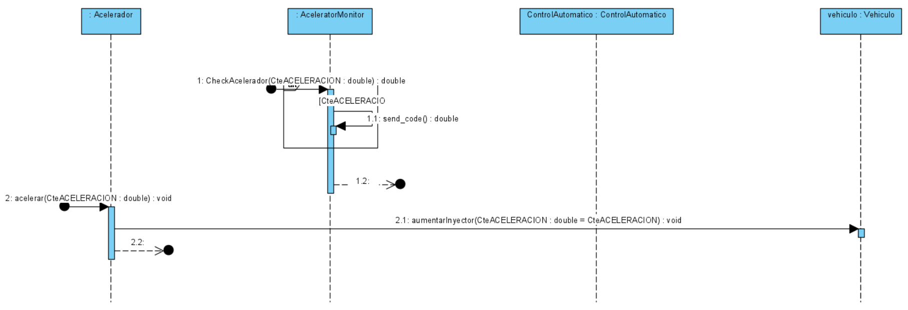

: Acelerador
: AceleratorMonitor
ControlAutomatico : ControlAutomatico
vehiculo : Vehiculo
1: CheckAcelerador(CteACELERACION : double) : double
[CteACELERACIO
1.1: send_code() : double
1.2:
2: acelerar(CteACELERACION : double) : void
2.1: aumentarInyector(CteACELERACION : double = CteACELERACION) : void
2.2:

Figure 4.28: sequence diagram of acceleration monitoring

- The following sequence diagram illustrates the classes and methods responsible for comparing the current speed with the specified speed for feature F-3.

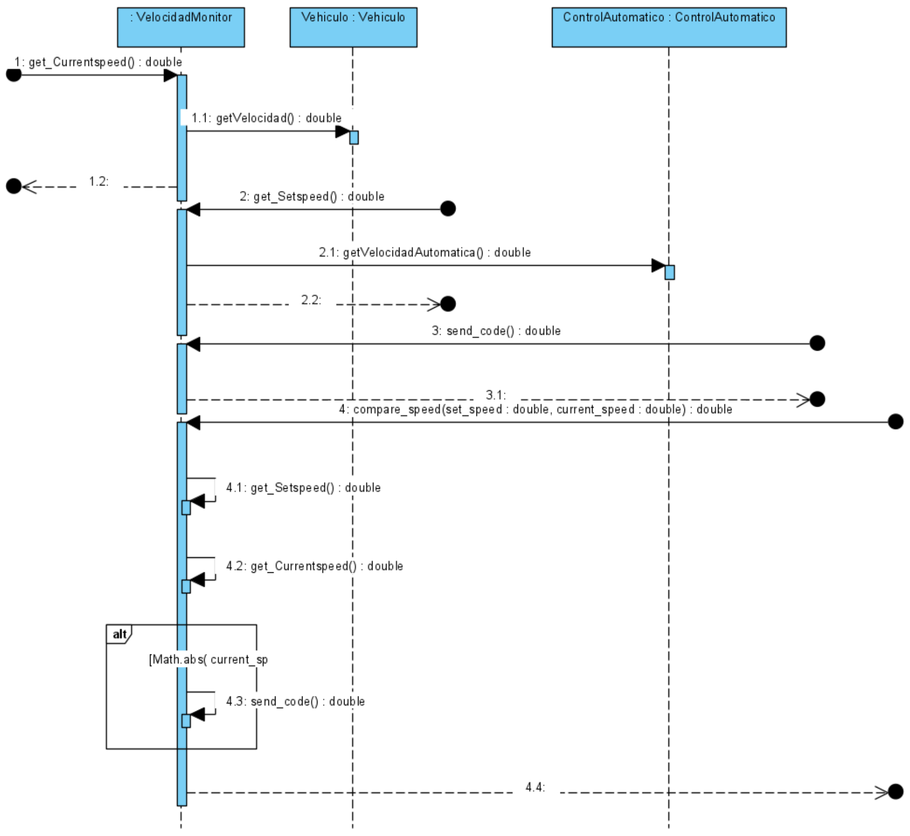

: VelocidadMonitor
Vehiculo : Vehiculo
ControlAutomatico : ControlAutomatico
1: get_Currentspeed() : double
1.1: getVelocidad() : double
1.2:
2: get_Setspeed() : double
2.1: getVelocidadAutomatica() : double
2.2:
3: send_code() : double
3.1:
4: compare_speed(set_speed : double, current_speed : double) : double
4.1: get_Setspeed() : double
4.2: get_Currentspeed() : double
alt
[Math.abs(current_sp
4.3: send_code() : double
4.4:

Figure 4.29: sequence diagram of monitoring current speed with set one

- The interface that will be utilized to create loose coupling will be used to deliver error codes to the automatic control unit, as illustrated by the sequence diagram for features F-2 and F-4 below.

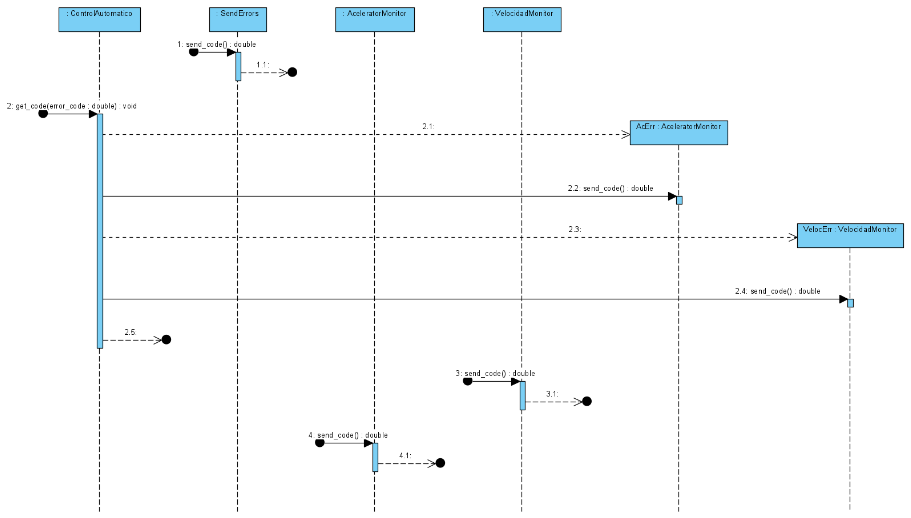


Figure 4.30: sequence diagram of sending errors code

According to ISO 26262 standards, software architecture design is subject to general principles based on ASIL classification, as shown in the table below, (+) indicates that it is recommended; (++) that it is highly recommended.

Principles	ASIL			
	A	B	C	D
"Appropriate hierarchical structure of the software components"	++	++	++	++
"Restricted size and complexity of software components"	++	++	++	++
"Restricted size of interfaces "	+	+	+	++
"Strong cohesion within each software component"	+	++	++	++
"Loose coupling between software components "	+	++	++	++
"Appropriate scheduling properties "	++	++	++	++
"Restricted use of interrupts "	+	+	+	++
"Appropriate spatial isolation of the software components "	+	+	+	++
"Appropriate management of shared resources "	++	++	++	++

Table 4.31: Software architectural design principles according to iso 26262-6 [2]

Usually, the system is divided into subsystems that perform a specific function or achieve a safety goal in accordance with ISO 26262. The safety requirements and objectives of these systems are classified according to ASIL. From the preceding, the part related to system monitoring and error detection has been classified as ASIL B, based on the safety goal and its primary function. From here, we will apply design principles according to their classification, as well as how to verify it.
To understand how the design will be, below is a description of the contents of the table 4.31.

- "Hierarchical structure of software components"
  The objective is attained by partitioning the large design blocks into smaller units. This is done by proceeding from System Level Software Elements to software components, and then to Software Sub-Components.

- "Restricted size and complexity of software components"
  The software component should be chosen based on its unique characteristics and functionalities.

- "Restricted size of interfaces "
  Restrict the amount of data transmitted between software components.

- "Strong cohesion within each software component"
  Based on the functionality that will be used, the software component with high cohesion is selected.

- "Loose coupling between software components"
  A system with loose coupling has components that can function independently of each other, and can be replaced or modified without affecting the overall system, Interfaces can be utilized to implement this.

- "Appropriate scheduling "
  consideration of the sequence of data transfer between software components.

- "Restricted use of interrupts"
  Using interrupt-based processing can cause the program to constantly switch to a different mode and thus affect execution scheduling.

- "Appropriate spatial isolation of the software components"
  Isolate software components in a separate memory that cannot be modified by other components.

- "Appropriate management of shared resources"
  Manage access to resources and ensure no conflict between components to obtain resources.

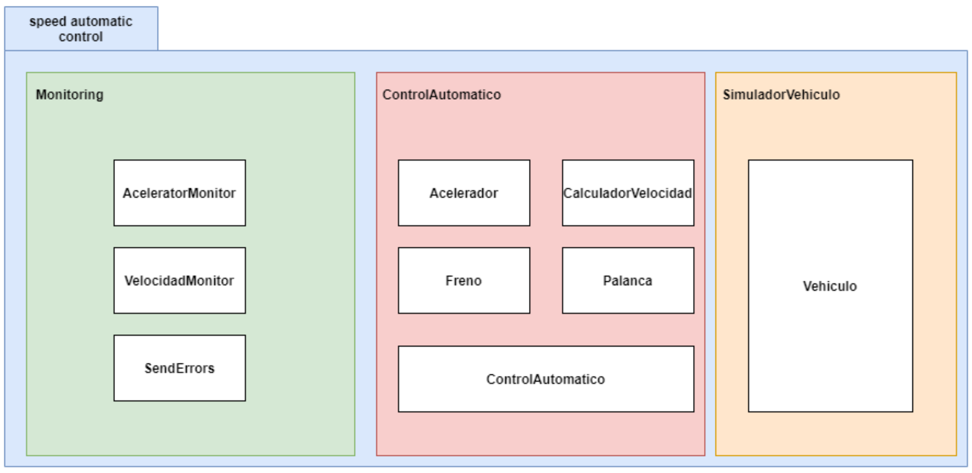


Figure 4.31: Basic hierarchical structure

Adhering to the design principles mentioned in table 4.31 will provide quality assurance and avoid potential failures in the system being developed. Therefore, building a software architectural design on a hierarchical structure based on the previous design principles enables us to comprehend how to code and how software components interact with one another. Each component of the system must be responsible for a specific function. one, as in the following figure 4.31, where the components that perform the monitoring function are grouped together, and thus we achieve strong cohesion among the components and reduce the unwanted complexity and interference. At the same time, a trade-off should be taken into account between increasing the complexity of the system and reducing the dependence between the components. Furthermore, isolating the components from one another makes it easier to manage shared resources while also reducing the use of interrupts and interest in special scheduling. By implementing a specific function for each component in the simulation environment, there is no possibility to physically isolate the components or use interrupts as in embedded systems, but threads are used to make the main components. It operates independently and has its own, albeit limited, memory. Figure 4.32 shows how each component contains its own sub-components that are isolated from each other as much as possible.

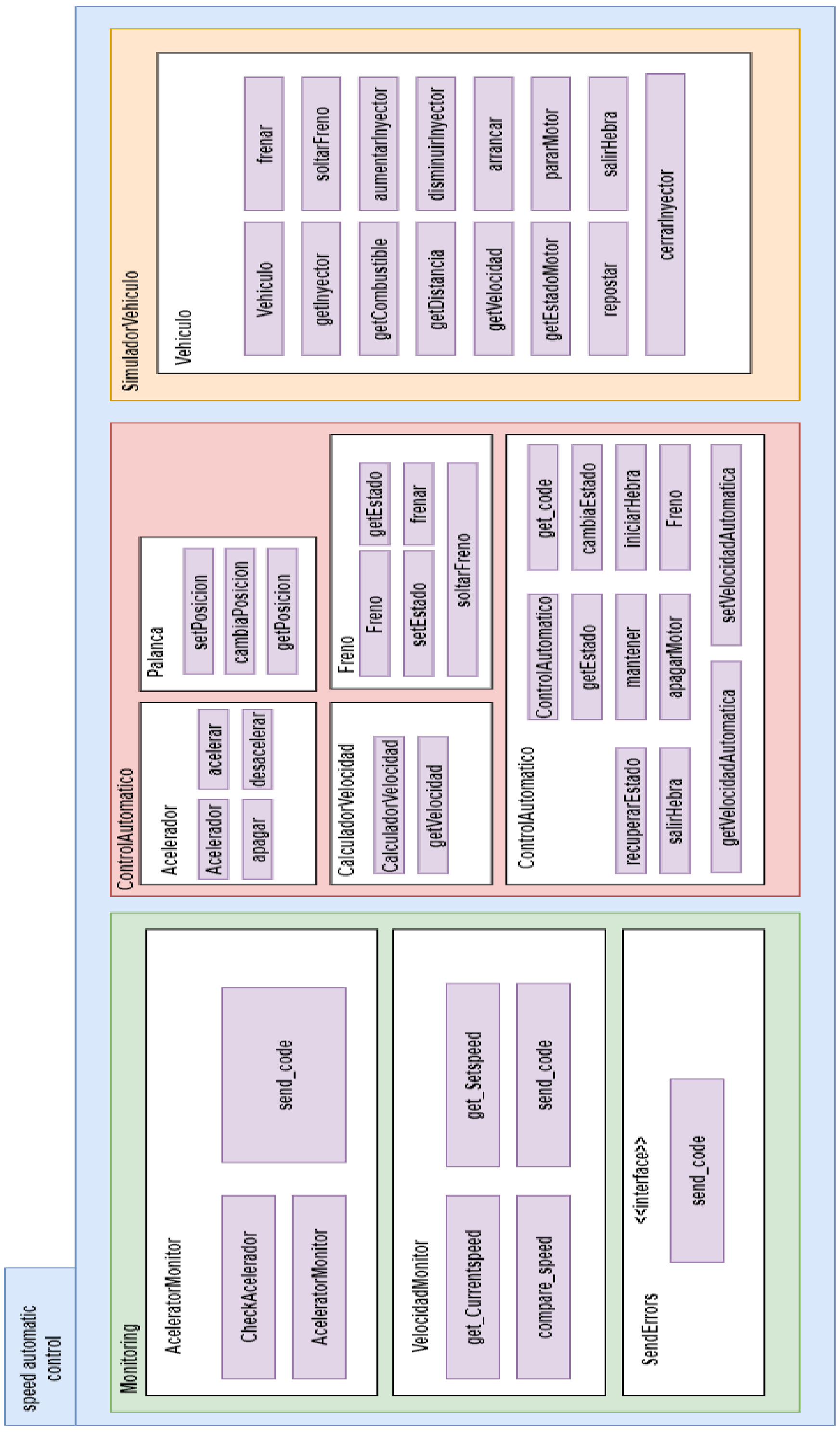


Figure 4.32: hierarchical of software components

To validate the design, ISO 26262 introduced several methods, as displayed in the table 4.32. The selection of the method is based on its capability to verify the identified safety requirements and its compatibility with the software's target environment. The "inspection of the design" method is suitable to validate our design by ensuring that the system complies with predefined safety requirements and that there are no outstanding safety requirements that have not been fulfilled.

Methods	ASIL			
	A	B	C	D
"Walk-through of the design"	++	+	o	o
"Inspection of the design"	+	++	++	++
"Simulation of dynamic behaviour of the design "	+	+	+	++
"Prototype generation"	o	o	+	++
"Formal verification"	o	o	+	+
"Control flow analysis "	+	+	++	++
"Data flow analysis "	+	+	++	++
" Scheduling analysis"	+	+	++	++

Table 4.32: Verification methods for software architectural designs according to iso 26262-6 [2]

We can trace the features and verify that they were achieved in a simple manner using the table below.

Feature No	component name	check status
F-1	aceleratorMonitor	checked
F-2	sendcode interface	checked
F-3	VelocidadMonitor	checked
F-4	sendcode interface	checked

Table 4.33: Design checklist

Through the design requirements, an interface(send error) that was not present in the overall model has been added, and for this, the overall model will be updated as shown in figure 4.33 with the details of the classes and methods used.

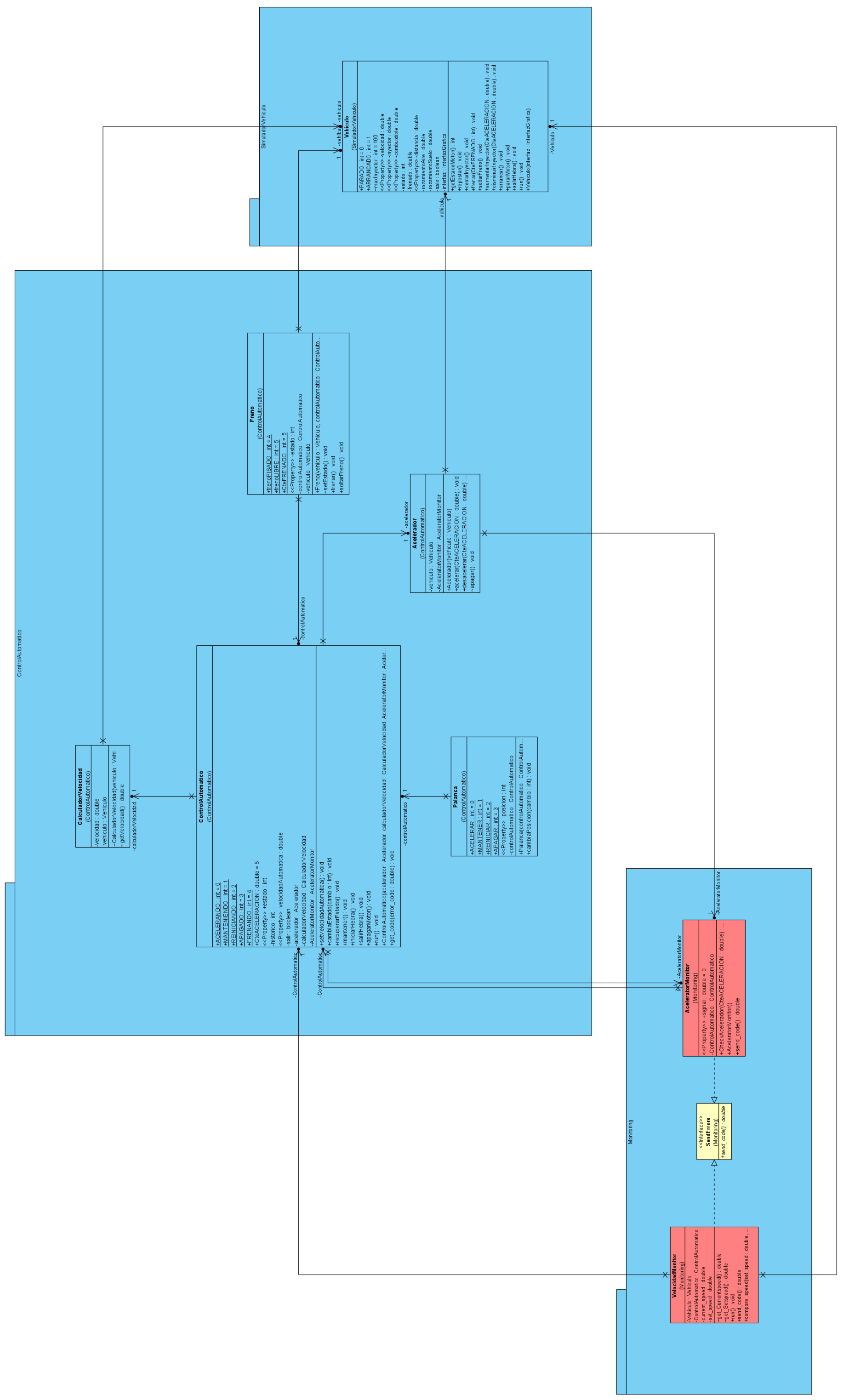


Figure 4.33: updated overall model

4.6.5 TDD

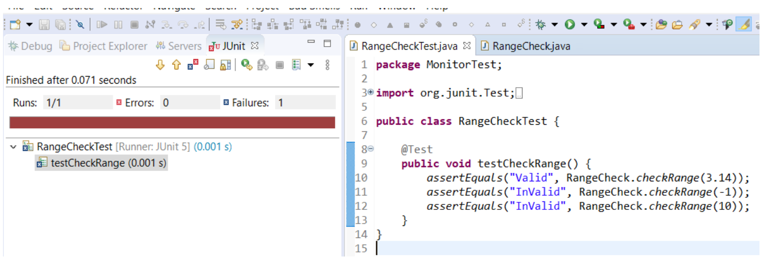
Applying this section to the components that have been added, the purpose of this is to avoid errors that face the design before the code is actually written. In the next stage, simplified tests will be written for the monitoring components.

Acceleration Monitor

Acceleration monitoring will be done by checking the safe range of the inputs, in general. To implement TDD, the test is written before writing the code by creating a "Rangecheck" function, the following tables show test cases for the method we want to code later.

Input	Actual output	Expected output	Test Result
3.14	invalid	valid	fail
-1	valid	invalid	fail
10	valid	invalid	fail

Table 4.34: check range red test



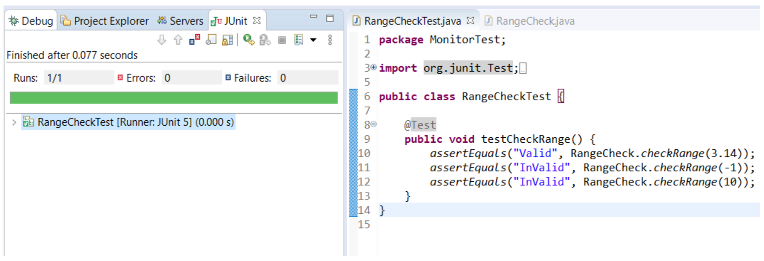
```
Finished after 0.071 seconds
Runs: 1/1    Errors: 0    Failures: 1
RangeCheckTest [Runner: JUnit 5] (0.001 s)
    testCheckRange (0.001 s)
```

```java
package MonitorTest;

import org.junit.Test;

public class RangeCheckTest {

    @Test
    public void testCheckRange() {
        assertEquals("Valid", RangeCheck.checkRange(3.14));
        assertEquals("InValid", RangeCheck.checkRange(-1));
        assertEquals("InValid", RangeCheck.checkRange(10));
    }
}
```

Figure 4.34: red test Running check range test case

Input	Actual output	Expected output	Test Result
3.14	valid	valid	pass
-1	invalid	invalid	pass
10	invalid	invalid	pass

Table 4.35: check range green test



```
Finished after 0.077 seconds
Runs: 1/1    Errors: 0    Failures: 0
> RangeCheckTest [Runner: JUnit 5] (0.000 s)
```

```java
package MonitorTest;

import org.junit.Test;

public class RangeCheckTest {

    @Test
    public void testCheckRange() {
        assertEquals("Valid", RangeCheck.checkRange(3.14));
        assertEquals("InValid", RangeCheck.checkRange(-1));
        assertEquals("InValid", RangeCheck.checkRange(10));
    }
}
```

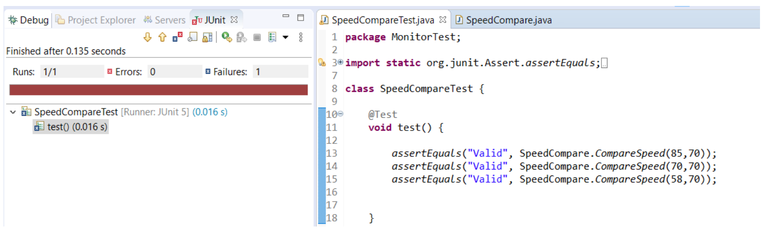
Figure 4.35: Running check range green test

Speed Monitoring

In the cruise control system, it is critical to monitor whether the current speed is equal to the speed set by the driver. A function that compares the current speed to the set speed will be tested with a margin of error of 10 as a safe difference between the two speeds. Before writing the main code for this method, a test case will be written, indicating that this method performs a function, comparing the two speeds without going into more complex details, such as how to read the speeds or make them run on a separate thread because these details will be in the function's main code.

Input (current speed, set speed)	Actual output	Expected output	Test Result
(85,70)	invalid	invalid	pass
(70,70)	valid	valid	pass
(58,70)	valid	invalid	fail

Table 4.36: Monitroring speed red test



```
Finished after 0.135 seconds
Runs: 1/1    Errors: 0    Failures: 1
v SpeedCompareTest [Runner: JUnit 5] (0.016 s)
    test() (0.016 s)
```

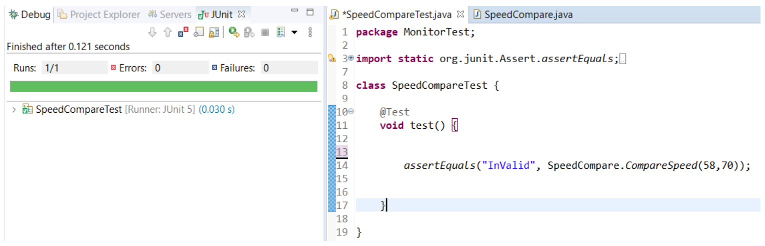
```java
package MonitorTest;

import static org.junit.Assert.assertEquals;

class SpeedCompareTest {

    @Test
    void test() {

        assertEquals("Valid", SpeedCompare.CompareSpeed(85,70));
        assertEquals("Valid", SpeedCompare.CompareSpeed(70,70));
        assertEquals("Valid", SpeedCompare.CompareSpeed(58,70));


    }
```

Figure 4.36: Running monitoring speed test case

There is a failure in the test that was executed, and in the event of deceleration at the current speed, the function does not detect the error, and therefore a refactor will be made for the method in order for the test to pass. Errors that appear before the start of writing the actual code will be discovered before starting the next step, and this will reduce the effort and time on the work team.

Input (current speed, set speed)	Actual output	Expected output	Test Result
(85,70)	invalid	invalid	pass
(70,70)	valid	valid	pass
(58,70)	invalid	invalid	pass

Table 4.37: Monitoring speed green test



```
Finished after 0.121 seconds
Runs: 1/1    Errors: 0    Failures: 0
> SpeedCompareTest [Runner: JUnit 5] (0.030 s)
```

```java
package MonitorTest;

import static org.junit.Assert.assertEquals;

class SpeedCompareTest {

    @Test
    void test() {


        assertEquals("InValid", SpeedCompare.CompareSpeed(58,70));


    }

}
```

Figure 4.37: Running monitoring speed test case after refactoring

4.6.6 Build by Feature

Section	Description
Entry	phase 4 design by feature completed, with design inspected list.
Tasks	1. Class owners implement the requirements of the features that were previously defined, particularly the safety requirements. 2. Each class owner will test the code to verify if it fits the feature requirements; these tests are often defined and detailed by the chief programmer. 3. Inspection of the code, whether before or after testing or even during coding, is one of the chief programmer's tasks, and he or she must make a decision on it. 4. After testing and code inspection are complete, the build of the class is released.
Verification	1. code inspection and unit test. 2. unit design code inspection according to ISO 26262-6 8.4. 3. Methods for verifying software units according to ISO 26262-6 9.4.2.
Exit	The completion of each feature and the classes associated with it after its promotion to the build via the completion of the inspection and testing on the code marks the end of this process.

Table 4.38: build by feature phase template

In addition to the requirements of the identified features, ISO 26262-6 specifies a number of principles that must be followed during implementation to avoid failure, and these principles must be available when the code inspection is performed.

Principles	ASIL			
	A	B	C	D
"One entry and one exit point in sub-programmes and functions"	++	++	++	++
"No dynamic objects or variables, or else online test during their creation"	+	++	++	++
"Initialization of variables "	++	++	++	++
"No multiple use of variable names"	++	++	++	++
"Avoid global variables or else justify their usage"	+	+	++	++
"Restricted use of pointers"	+	++	++	++
"No implicit type conversions"	+	++	++	++
" No hidden data flow or control flow"	+	++	++	++
" No unconditional jumps"	++	++	++	++
" No recursions"	+	+	++	++

Table 4.39: Design and implementation principles for a software unit in accordance with ISO 26262-6 [2]

As stated in the table below, ISO 26262 specifies different methods for testing software units, and we can use one or more of them. The fault-tolerance method is suitable for ensuring that the software unit detects an error.

Methods	ASIL			
	A	B	C	D
"Walk-through"	++	+	o	o
"Pair-programming"	+	+	+	+
"Inspection"	+	++	++	++
"Semi-formal verification"	+	+	++	++
"Formal verification"	o	o	+	+
"Control flow analysis"	+	+	++	++
"Data flow analysis"	+	+	++	++
"Static code analysis"	++	++	++	++
" Static analyses based on abstract interpretation"	+	+	+	+
"Requirements-based test"	++	++	++	++
"Interface test"	++	++	++	++
"Fault injection test"	+	+	+	++
"Resource usage evaluation"	+	+	+	++
"Back-to-back comparison test between model and code, if applicable"	+	+	++	++

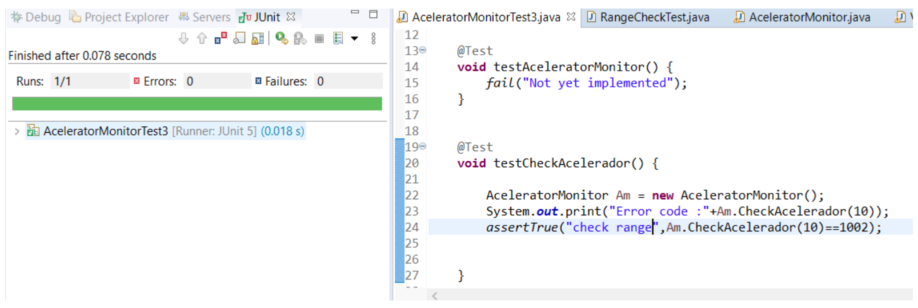
Table 4.40: Methods for verifying software units in accordance with ISO 26262-6 [2]

Now we are testing implemented units that fulfil the features requirements :

- Accelerator monitor : The main idea of this function is to monitor the accelerator outputs to check if they are within the safe range, which is (0, 5), and if they are outside this range, an error code will be sent to the automatic speed control unit. We assumed that the error code that is sent is 1002 because there are no standardized fault codes among vehicle manufacturers.

Input	Actual output	Expected output	Test Result
3.14	0	0	pass
-1	1002	1002	pass
10	1002	1002	pass

Table 4.41: Accelerator monitoring test case 1



Finished after 0.078 seconds

Runs: 1/1 Errors: 0 Failures: 0

AceleratorMonitorTest3 [Runner: JUnit 5] (0.018 s)

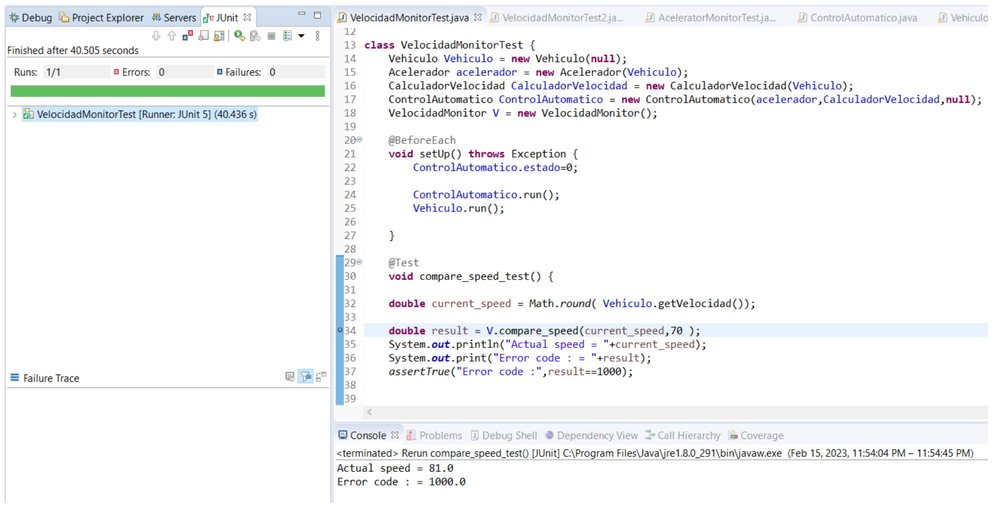
```
    @Test
    void testAceleratorMonitor() {
        fail("Not yet implemented");
    }


    @Test
    void testCheckAcelerador() {

        AceleratorMonitor Am = new AceleratorMonitor();
        System.out.print("Error code :"+Am.CheckAcelerador(10));
        assertTrue("check range",Am.CheckAcelerador(10)==1002);


    }
```

Figure 4.38: Running Accelerator monitoring test

- velocity monitor : In this software unit, the cruise control system will be monitored when the cruise is set, and whether the current speed is not accelerating or decelerating by 10 from the set speed. In case of an error, an error code will be sent to the automatic control unit.

Input (current speed, set speed)	Actual output	Expected output	Test Result
(81,70)	1000	1000	pass
(70,70)	0	0	pass
(58,70)	1000	1000	pass

Table 4.42: Velocity Monitoring test case 2



Finished after 40.505 seconds

Runs: 1/1 Errors: 0 Failures: 0

VelocidadMonitorTest [Runner: JUnit 5] (40.436 s)

```
class VelocidadMonitorTest {
    Vehiculo Vehiculo = new Vehiculo(null);
    Acelerador acelerador = new Acelerador(Vehiculo);
    CalculadorVelocidad CalculadorVelocidad = new CalculadorVelocidad(Vehiculo);
    ControlAutomatico ControlAutomatico = new ControlAutomatico(acelerador,CalculadorVelocidad,null);
    VelocidadMonitor V = new VelocidadMonitor();

    @BeforeEach
    void setUp() throws Exception {
        ControlAutomatico.estado=0;

        ControlAutomatico.run();
        Vehiculo.run();

    }

    @Test
    void compare_speed_test() {

    double current_speed = Math.round( Vehiculo.getVelocidad());

    double result = V.compare_speed(current_speed,70 );
    System.out.println("Actual speed = "+current_speed);
    System.out.print("Error code : = "+result);
    assertTrue("Error code :",result==1000);
```

```
<terminated> Rerun compare_speed_test() [JUnit] C:\Program Files\Java\jre1.8.0_291\bin\javaw.exe  (Feb 15, 2023, 11:54:04 PM – 11:54:45 PM)
Actual speed = 81.0
Error code : = 1000.0
```

Figure 4.39: Running Velocity monitoring test

Backward tracing will be used in this phase The purpose of backward traceability is to ensure that all parts of the software development process are linked and traceable, including testing and maintenance, Each feature is upgraded to build after testing, code inspection, and client acceptance.

Feature No	Requirement Description	Implementation Component	Test Case Identifier	Test Results
F-1	Monitor the output of the accelerator in case the output is outside the safe range	aceleratorMonitor.java	Test case 1	pass
F-2	Send error code 1002 to the automatic control in case detect error	SendErrors.java	Test case 1	pass
F-3	Make a comparison of the current speed of the vehicle with the speed set by the driver, and if the difference is more than 10 km/h, an error code is sent to the control unit.	VelocidadMonitor.java	Test case 2	pass
F-4	Send error code 1000 to the automatic control in case detect error	SendErrors.java	Test case 2	pass

Table 4.43: Tracing backward features

Chapter 5

Results and Discussion

5.1 Results

List of results:

I. Methodological:

   1. New methodological approach for SCS development
   2. Application of STPA to AVSC validation
   3. TDD-based validation and refactoring of AVSC
   4. FDD-based development (initiated)

II. Design:

   1. AVSC potential hazards identification through the definition of tests cases
   2. Elicitation of AVSC functional and non-functional requirements with EARS notation
   3. Risks categorization
   4. Determination of AVSC's ASIL severity levels

III. Building:

   1. STPA implementation for the AVSC
   2. Determination of the set of constraints that propitiate system safety and prevent from system's unsafe control action (UCA)
   3. FDD-based construction/refactoring of the AVSC
   4. Identification of the AVSC features list (monitoring acceleration, speed control, ...)

5. Development of a suite of tests for AVCS validation

IV. Safety:

1. The refactored software demonstrated improved safety features, as it adhered to the safety requirements.

5.2 Discussion

I.1 A new approach for developing SCS has been proposed, starting from a previously obtained backlog, which includes F and NF requirements for the AVCS study case given and studied in this project.

I.2 By using STPA, we have obtained and detailed the safety requirements of the study case through hazard analysis.

I.3 To avoid possible hidden errors in the system design, we used test-driven development before starting to refactor the case study software.

I.4 In this phase, an FDD strategy has been outlined and, to some extent, initiated for the AVSC study.

II.1 A re-evaluation of the AVCS case study has been carried out by deriving a set of test cases, which test possible hazardous scenarios in the deployment and operation of the system.

II.2 The set of functional requirements has been obtained, with the help of EARS structural notation, and the specific set of AVSC safety requirements according to ISO-26262.

II.3 The set of AVCS software operation risks have been categorized according to different levels of risk attributes, i.e. severity (S0-S3), exposure (E0-E3) and controllability (C0-C3).

II.4 A correspondence has been established between the AVCS safety requirements and the severity levels (A–D) proposed by the ASIL standard.

III.1 The STPA has been implemented as a global reference control structure diagram for the refactoring of the AVCS software, and from it the unsafe control actions have been identified, as well as the causes that trigger them.

III.2 The set of constraints that guarantee the satisfaction of safety requirements has been fully identified and, therefore, the constraints that may prevent an AVCS operation leading to a UCA have been elicited.

III.3 The FDD has been applied by developing a new architectural model that has been obtained and that separates the AVCS software architecture into different sub-domains. By accessing the system requirements in the product backlog, according to the FDD methodology, the software components responsible for safety have been isolated according to the iSO 26262 recommendations.

III.4 A list of the product features (case study software), in terms of acceleration/deceleration monitoring, speed control, has been elaborated so that a feature-by-feature plan can be applied, according to the steps of the FDD methodology, to validate and eventually refactoring the AVCS software.

III.5 The development of a test suite for the AVCS software to monitor acceleration and speed control has been proposed and schematically introduced. A complete set of features has been identified to track backwards to validate the AVCS from software implementation to requirements, which were obtained in the first step of the FDD method.

IV.1 Finally,The refactored software included a safety mechanism for detecting and handling errors, as part of the obligations that must be considered according to ISO 26262.

Chapter 6

Conclusions and future work

6.1 Conclusions

The main objective of our research is to define agile software development (ASD) methods for developing a complex system with critical safety characteristics. By applying ASD to this type of system, we aim to leverage the advantages of the Agile development process, such as flexibility, adaptability to changing requirements, incremental delivery, customer collaboration, continuous improvement, early risk identification and mitigation, budget and cost estimation, and improvement of quality and customer satisfaction. However, developing safety-critical systems (SCS) involves strict standards that emphasize heavy documentation and resist changes in requirements or development plans. On the other hand, implementing Agile with SCS faces barriers due to its focus on the development process rather than documentation, the flexibility to accommodate changing requirements, reliance on loosely written user stories, and iterative and incremental development processes.
To address this issue, we proposed a new methodology that integrates Feature-Driven Development (FDD), System-Theoretic Process Analysis (STPA), and Test-Driven Development (TDD), with a backlog containing initial requirements separated into safety and functional requirements, In order to notate the requirements, an Easy Approach to Requirement Syntax (EARS) is used. We expected this methodology to provide an acceptable compromise between the advantages of Agile, such as flexibility, adaptability, and customer collaboration, and the SCS development specification, such as comprehensive documentation and fulfilling safety regulations corresponding with SCS development.

Through a case study of the Automatic Vehicle Speed Control (AVSC) system, we demonstrate the effectiveness of our methodology. By refactoring the AVSC software according to ISO-26262 standards and conducting rigorous testing, we improve the safety features and verify that it complies with safety requirements. The results of our study indicated the possibility of developing software with safety features compatible with standards using agile methodologies in a practical way, offering a balanced approach that effectively addresses safety considerations while leveraging the advantages of agile practices. Through the case study of the AVSC system, we observed improved safety features, compliance with safety requirements, and enhanced software reliability. this research contributes to bridging the gap between Agile development and SCS, providing valuable insights for industries seeking to develop complex software systems with safety-critical characteristics.

The implications of our findings extend beyond the specific AVSC case study. The proposed methodology can be applied to a wide range of SCS in various industries. Software development teams can effectively address safety considerations, mitigate risks, and enhance overall software reliability. This has implications for industries such as healthcare, aerospace, and energy, where software safety is paramount.

While our research has made knowledge contributions, it is important to acknowledge certain limitations. The proposed methodology has been validated through the AVSC case study, and further research is needed to assess its applicability in different contexts and industries. Additionally, careful consideration is required when generalizing the findings to all types of SCS. These limitations provide opportunities for future studies to explore these aspects in greater depth.

6.2 Future work

In this area should focus on several important aspects. First, further research is needed to validate the proposed methodology in different contexts and industries beyond the AVSC case study. This will help assess its applicability and effectiveness in a variety of SCS. Additionally, conducting comparative studies that analyze the performance and outcomes of the proposed methodology against other existing approaches would provide valuable insights for practitioners. Finally, investigating the scalability of the methodology for larger and more complex software systems would be beneficial to ensure its practicality in real-world scenarios.

Chapter 7

References

Bibliography

[1] URL: `http://agilemanifesto.org/principles.html`.

[2] URL: `https://www.iso.org/standard/68388.html`.

[3] Pekka Abrahamsson et al. "Agile software development methods: Review and analysis". In: *arXiv preprint arXiv:1709.08439* (2017).

[4] Shabib Aftab et al. "Comparative Analysis of FDD and SFDD". In: *International Journal of Computer Science and Network Security* 18.1 (2018), pp. 63–70.

[5] Shabib Aftab et al. "Using FDD for small project: An empirical case study". In: *International Journal of Advanced Computer Science and Applications* 10.3 (2019), pp. 151–158.

[6] Jayaweera A Lakshika Anjalee, Victoria Rutter, and Nithushi R Samaranayake. "Application of Failure Mode and Effect Analysis (FMEA) to improve medication safety: a systematic review". In: *Postgraduate Medical Journal* 97.1145 (2021), pp. 168–174.

[7] Faiza Anwer et al. "Comparative analysis of two popular agile process models: extreme programming and scrum". In: *International Journal of Computer Science and Telecommunications* 8.2 (2017), pp. 1–7.

[8] Dave Astels. *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference, 2003.

[9] Paris Avgeriou and Uwe Zdun. "Architectural patterns revisited-a pattern language". In: (2005).

[10] Douglas Augusto Barcelos Bica and Carlos Alexandre Gouvea da Silva. "Learning Process of Agile Scrum Methodology With Lego Blocks in Interactive Academic Games: Viewpoint of Students". In: *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje* 15.2 (2020), pp. 95–104. DOI: `10.1109/RITA.2020.2987721`.

[11] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* 2013.

[12] Kent Beck. *Extreme programming explained: embrace change.* addison-wesley professional, 2000.

[13] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[14] Herbert D. Benington. "Production of Large Computer Programs". In: *Annals of the History of Computing* 5.4 (1983), pp. 350–361. DOI: `10.1109/MAHC.1983.10102`.

[15] Konstantin Beznosov. "Extreme security engineering: On employing XP practices to achieve'good enough security'without defining it". In: *First ACM Workshop on Business Driven Security Engineering (BizSec), Fairfax, VA.* Vol. 31. 2003.

[16] Konstantin Beznosov and Philippe Kruchten. "Towards agile security assurance". In: *Proceedings of the 2004 workshop on New security paradigms.* 2004, pp. 47–54.

[17] B. Boehm. "Get ready for agile methods, with care". In: *Computer* 35.1 (2002), pp. 64–69. DOI: `10.1109/2.976920`.

[18] Jonathan Bowen. "Formal methods in safety-critical standards". In: *Proceedings 1993 Software Engineering Standards Symposium.* IEEE. 1993, pp. 168–177.

[19] Riccardo Carbone et al. "Scrum for safety: Agile development in safety-critical software systems". In: *International Conference on the Quality of Information and Communications Technology.* Springer. 2021, pp. 127–140.

[20] H Frank Cervone. "Understanding agile project management methods using Scrum". In: *OCLC Systems & Services: International digital library perspectives* (2011).

[21] John O Clark. "System of systems engineering and family of systems engineering from a standards, V-model, and dual-V model perspective". In: *2009 3rd annual IEEE systems conference.* IEEE. 2009, pp. 381–387.

[22] Jane Cleland-Huang and Michael Vierhauser. "Discovering, analyzing, and managing safety stories in agile projects". In: *2018 IEEE 26th International Requirements Engineering Conference (RE).* IEEE. 2018, pp. 262–273.

[23] Bruce Powel Douglass. *Real-time design patterns: robust scalable architecture for real-time systems.* Addison-Wesley Professional, 2003.

[24] Tore Dybå and Torgeir Dingsøyr. "Empirical studies of agile software development: A systematic review". In: *Information and software technology* 50.9-10 (2008), pp. 833–859.

[25] A Ericson Clifton. *Hazard analysis techniques for system safety, Hoboken.* 2005.

[26] Siamak Farshidi, Slinger Jansen, and Jan Martijn van der Werf. "Capturing software architecture knowledge for pattern-driven design". In: *Journal of Systems and Software* 169 (2020), p. 110714.

[27] Brian Fitzgerald et al. "Scaling agile methods to regulated environments: An industry case study". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 863–872. DOI: 10.1109/ICSE.2013.6606635.

[28] Kevin Forsberg and Harold Mooz. "The relationship of system engineering to the project cycle". In: *INCOSE international symposium*. Vol. 1. 1. Wiley Online Library. 1991, pp. 57–65.

[29] Xiaocheng Ge, Richard F Paige, and John A McDermid. "An iterative approach for development of safety-critical software and safety arguments". In: *2010 Agile Conference*. IEEE. 2010, pp. 35–43.

[30] A Hajou, RS Batenburg, and S Jansen. "Method æ, the agile software development method tailored for the pharmaceutical industry". In: *Lecture Notes on Software Engineering* 3.4 (2015), p. 251.

[31] Lise Tordrup Heeager. "Introducing agile practices in a documentation-driven software development practice: a case study". In: *Journal of Information Technology Case and Application Research* 14.1 (2012), pp. 3–24.

[32] Lise Tordrup Heeager and Peter Axel Nielsen. "A conceptual model of agile software development in a safety-critical context: A systematic literature review". In: *Information and Software Technology* 103 (2018), pp. 22–39.

[33] Lise Tordrup Heeager and Peter Axel Nielsen. "Meshing agile and plan-driven development in safety-critical software: a case study". In: *Empirical Software Engineering* 25.2 (2020), pp. 1035–1062.

[34] J Highsmith. "What is Agile Software Development? STSC Crosstalk". In: *Journal of Defense Software Engineering* (2002).

[35] Michael Hirsch. "Moving from a plan driven culture to agile development". In: *International Conference on Software Engineering*. Vol. 27. 2005, p. 38.

[36] Takuto Ishimatsu et al. "Modeling and hazard analysis using STPA". In: (2010).

[37] Ron Jeffries et al. *Extreme programming installed*. Addison-Wesley Professional, 2001.

[38] Henrik Jonsson, Stig Larsson, and Sasikumar Punnekkat. "Agile Practices in Regulated Railway Software Development". In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. 2012, pp. 355–360. DOI: 10.1109/ISSREW.2012.80.

[39] Henrik Jonsson, Stig Larsson, and Sasikumar Punnekkat. “Agile practices in regulated railway software development”. In: *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops*. IEEE. 2012, pp. 355–360.

[40] Rashidah Kasauli et al. “Safety-critical systems and agile development: a mapping study”. In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2018, pp. 470–477.

[41] Mohamad Kassab, Ghizlane El-Boussaidi, and Hafedh Mili. “A quantitative evaluation of the impact of architectural patterns on quality requirements”. In: *Software Engineering Research, Management and Applications 2011*. Springer, 2012, pp. 173–184.

[42] John C Knight. “Safety critical systems: challenges and directions”. In: *Proceedings of the 24th international conference on software engineering*. 2002, pp. 547–550.

[43] Nancy G Leveson. *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016.

[44] Adriana Libosvárová and Peter Schreiber. “Fault tree analysis optimized by genetic algorithms”. In: *Vedecké Práce Materiálovotechnologickej Fakulty Slovenskej Technickej Univerzity v Bratislave so Sıdlom v Trnave* 21.Special Issue (2013), p. 78.

[45] Weiguo Lin and Xiaomin Fan. “Software development practice for FDA-compliant medical devices”. In: *2009 International Joint Conference on Computational Sciences and Optimization*. Vol. 2. IEEE. 2009, pp. 388–390.

[46] Yaping Luo et al. “An architecture pattern for safety critical automated driving applications: Design and analysis”. In: *2017 Annual IEEE International Systems Conference (SysCon)*. IEEE. 2017, pp. 1–7.

[47] Alistair Mavin et al. “Easy approach to requirements syntax (EARS)”. In: *2009 17th IEEE International Requirements Engineering Conference*. IEEE. 2009, pp. 317–322.

[48] Tom McBride and Marion Lepmets. “Quality Assurance in Agile Safety-Critical Systems Development”. In: *2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 2016, pp. 44–51. DOI: `10.1109/QUATIC.2016.016`.

[49] Martin McHugh, Fergal McCaffery, and Valentine Casey. “Barriers to adopting agile practices when developing medical device software”. In: *International Conference on Software Process Improvement and Capability Determination*. Springer. 2012, pp. 141–147.

[50] Martin McHugh, Fergal McCaffery, and Garret Coady. "An agile implementation within a medical device software organisation". In: *International Conference on Software Process Improvement and Capability Determination*. Springer. 2014, pp. 190–201.

[51] Richard Messnarz et al. "Implementing functional safety standards–experiences from the trials about required knowledge and competencies (SafEUr)". In: *European Conference on Software Process Improvement*. Springer. 2013, pp. 323–332.

[52] Thor Myklebust and Tor Stålhane. *The agile safety case*. Springer, 2018.

[53] Zahid Nawaz, Shabib Aftab, and Faiza Anwer. "Simplified FDD Process Model." In: *International Journal of Modern Education & Computer Science* 9.9 (2017).

[54] J. Newkirk. "Introduction to agile processes and extreme programming". In: *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 2002, pp. 695–696. DOI: 10.1145/581441.581450.

[55] Jesper Pedersen Notander, Per Runeson, and Martin Höst. "A model-based framework for flexible safety-critical software development: a design study". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. 2013, pp. 1137–1144.

[56] Richard F Paige et al. "Towards agile engineering of high-integrity systems". In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2008, pp. 30–43.

[57] Steve R Palmer and Mac Felsing. *A practical guide to feature-driven development*. Pearson Education, 2001.

[58] Raman Ramsin and Richard F Paige. "Process-centered review of object oriented software development methodologies". In: *ACM Computing Surveys (CSUR)* 40.1 (2008), pp. 1–89.

[59] Adam Roman and Michal Mnich. "Test-driven development with mutation testing–an experimental study". In: *Software Quality Journal* 29.1 (2021), pp. 1–38.

[60] Pieter Adriaan Rottier and Victor Rodrigues. "Agile development in a medical device company". In: *Agile 2008 Conference*. IEEE. 2008, pp. 218–223.

[61] Winston W Royce. "Managing the development of large software systems: concepts and techniques". In: *Proceedings of the 9th international conference on Software Engineering*. 1987, pp. 328–338.

[62] Alejandra Ruiz et al. “Reuse of safety certification artefacts across standards and domains: A systematic approach”. In: *Reliability Engineering & System Safety* 158 (2017), pp. 153–171.

[63] Nayan B Ruparelia. “Software development lifecycle models”. In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13.

[64] Ken Schwaber and Mike Beedle. *Agile software development with scrum. Series in agile software development.* Vol. 1. Prentice Hall Upper Saddle River, 2002.

[65] Lubna Siddique and Bassam A. Hussein. “Practical insight about choice of methodology in large complex software projects in Norway”. In: *2014 IEEE International Technology Management Conference.* 2014, pp. 1–4. DOI: `10.1109/ITMC.2014.6918615`.

[66] Lalit Kumar Singh and Hitesh Rajput. “Dependability analysis of safety critical real-time systems by using Petri nets”. In: *IEEE Transactions on Control Systems Technology* 26.2 (2017), pp. 415–426.

[67] Maria Siniaalto and Pekka Abrahamsson. “A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage”. In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007).* 2007, pp. 275–284. DOI: `10.1109/ESEM.2007.35`.

[68] Tor Stålhane, Thor Myklebust, and GK Hanssen. “The application of Safe Scrum to IEC 61508 certifiable software”. In: *11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference.* Vol. 8. 2012, pp. 6052–6061.

[69] Ernst Stelzmann. “Contextualizing agile systems engineering”. In: *IEEE Aerospace and Electronic Systems Magazine* 27.5 (2012), pp. 17–22. DOI: `10.1109/MAES.2012.6226690`.

[70] Sardar Muhammad Sulaman et al. “Comparison of the FMEA and STPA safety analysis methods–a case study”. In: *Software quality journal* 27.1 (2019), pp. 349–387.

[71] Mohsan Tanveer. “Agile for large scale projects — A hybrid approach”. In: *2015 National Software Engineering Conference (NSEC).* 2015, pp. 14–18. DOI: `10.1109/NSEC.2015.7396338`.

[72] Christopher M Thomas. “An Overview of the Current State of the Test-First vs. Test-Last Debate”. In: *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal* 1.2 (2014), p. 2.

[73] Xiaofeng Wang, Kieran Conboy, and Minna Pikkarainen. “Assimilation of agile practices in use”. In: *Information Systems Journal* 22.6 (2012), pp. 435–455.

[74] Yang Wang, Jasmin Ramadani, and Stefan Wagner. “An exploratory study on applying a scrum development process for safety-critical systems”. In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2017, pp. 324–340.

[75] Yang Wang and Stefan Wagner. “Toward Integrating a System Theoretic Safety Analysis in an Agile Development Process.” In: *Software Engineering (Workshops)*. 2016, pp. 156–159.

[76] Andrew Wils et al. “Agility in the avionics software world”. In: *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer. 2006, pp. 123–132.

[77] Maryam Zahabi and David Kaber. “A fuzzy system hazard analysis approach for human-in-the-loop systems”. In: *Safety Science* 120 (2019), pp. 922–931.

Chapter 8

acronyms

Acronyms

ASD Agile software development. 11

ASIL Automotive Safety Integrity Levels. 50

AVSC automatic vehicle speed control. 16, 18

CCS Cruise Control System. 57

FDD Feature-Driven Development. 16, 40, 62

FMEA Failure Modes and Effects Analysis. 49

FTA Fault Tree Analysis. 48

QA Quality assurance. 16

RUP Rational Unified Process. 31

SCS Safety critical system. 11

STAMP Systems-Theoretic Accident Model and Processes. 46

STPA System-Theoretic Process Analysis. 16, 29, 46, 62

TDD Test-Driven Development. 16, 29, 45, 62

Xp eXtreme Programming. 37

Chapter 9

Figuers

List of Figures

Chapter 10

Tables

List of Tables