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 man-

ufacturer, 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

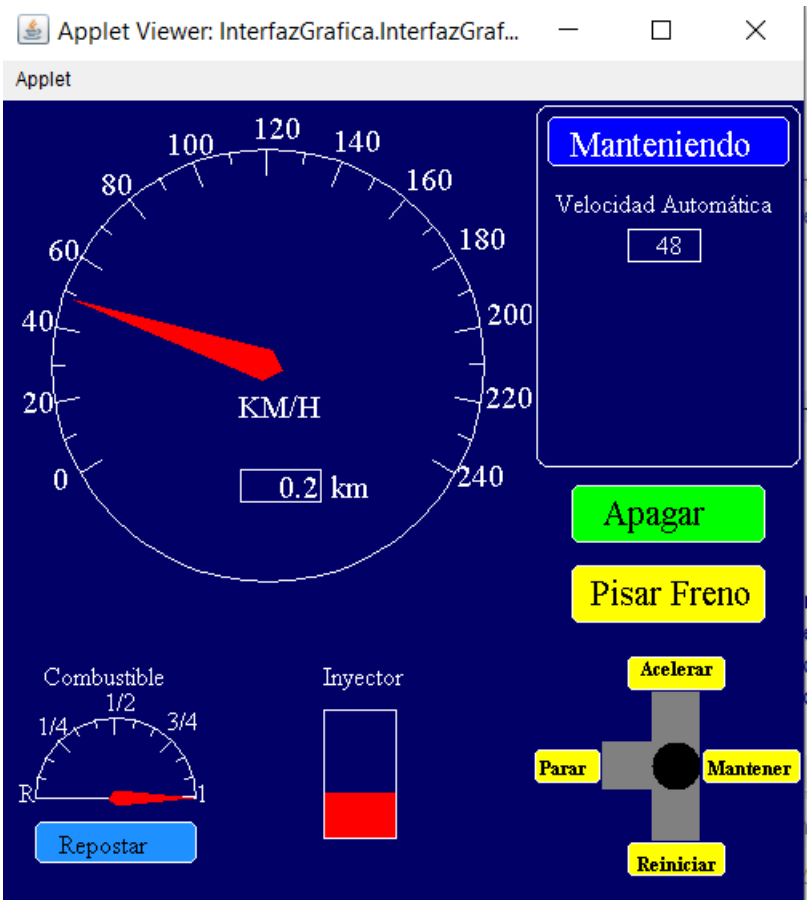

Figure 4.2: AVSC simulator

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.

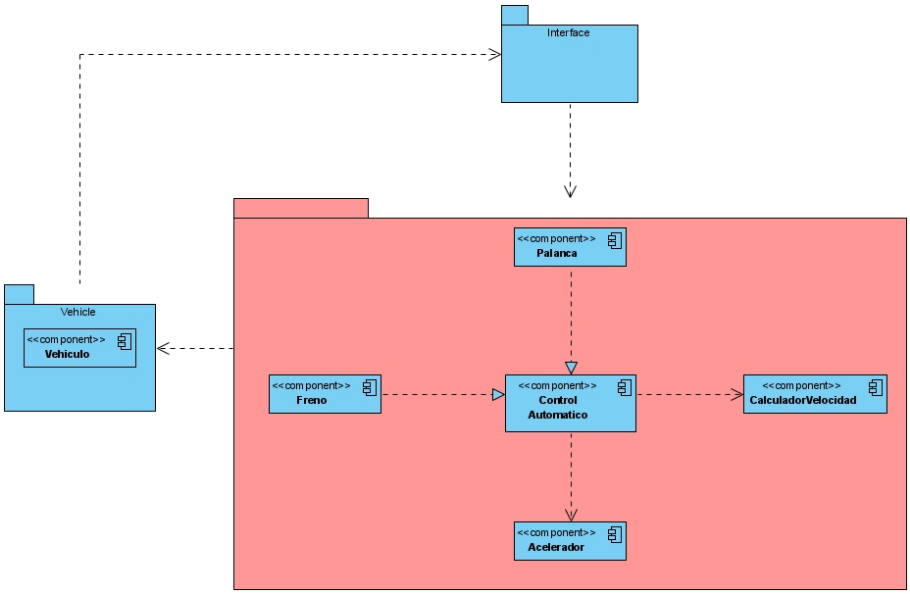

Figure 4.3: Components Diagram for simulator software

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.

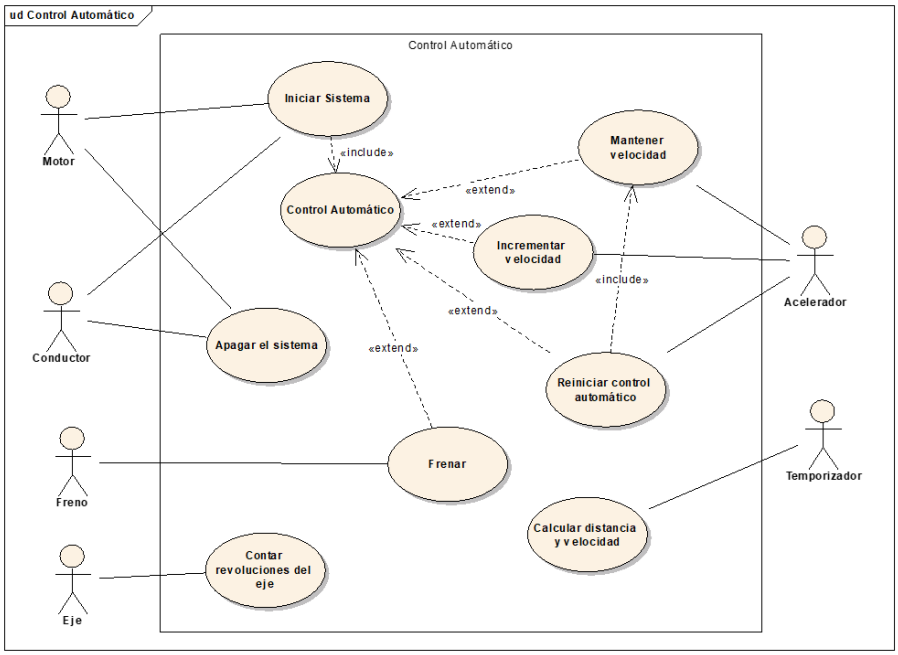

Figure 4.4: Use-case of speed automatic control

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.

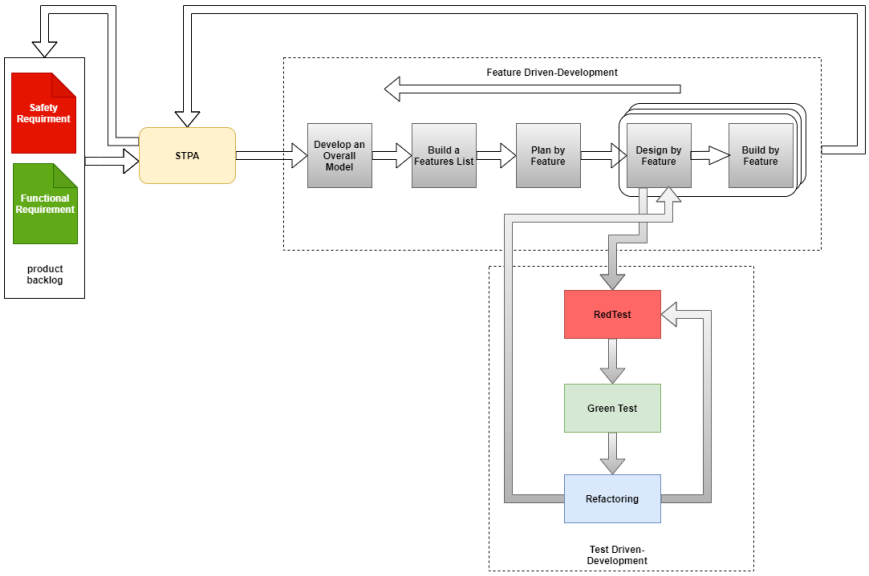

Figure 4.5: A proposed agile method

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.

```
3⊕import static org.junit.Assert.*;
16
17 public class VehicleTest {
18     Vehiculo Vehiculo = new Vehiculo(null);
19     Acelerador acelerador = new Acelerador(Vehiculo);
20     CalculadorVelocidad CalculadorVelocidad = new CalculadorVelocidad(Vehiculo);
21     ControlAutomatico ControlAutomatico = new ControlAutomatico(acelerador,CalculadorVelocidad);
22
23⊖    @BeforeClass
24     public static void setUpBeforeClass() throws Exception {
25
26     }
27
28⊖    @Before
29     public void setUp() {
30
31         Vehiculo.maxInyector=300;
32         ControlAutomatico.CteACELERACION=10;
33         ControlAutomatico.estado=0; // 0 mean Aceleration mode
34         ControlAutomatico.run();
35         Vehiculo.run();
36
37     }
```

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

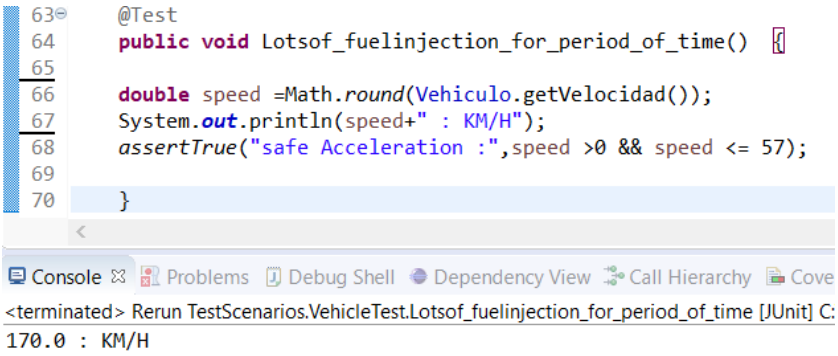

Figure 4.7: Running Test Case 1

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

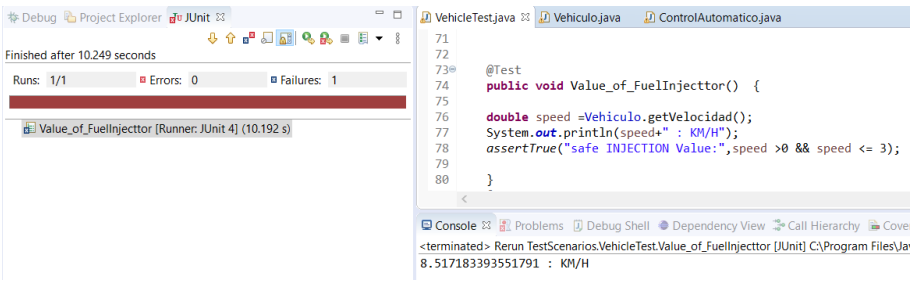

Figure 4.8: Running Test Case 2

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

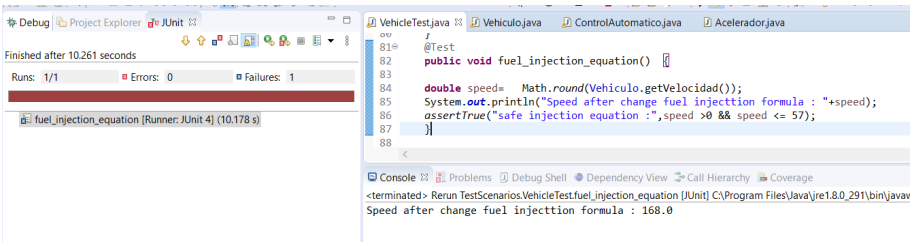

Figure 4.9: Running Test Case 3

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

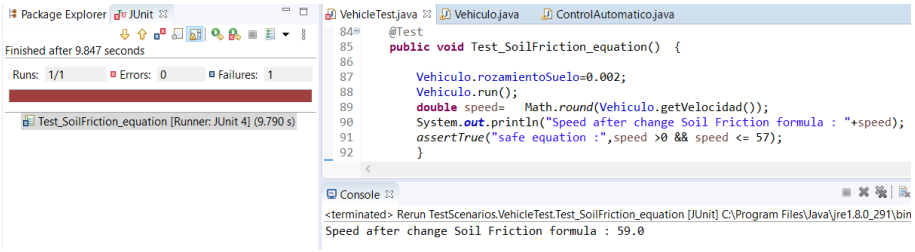

Figure 4.10: Running Test Case 4

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

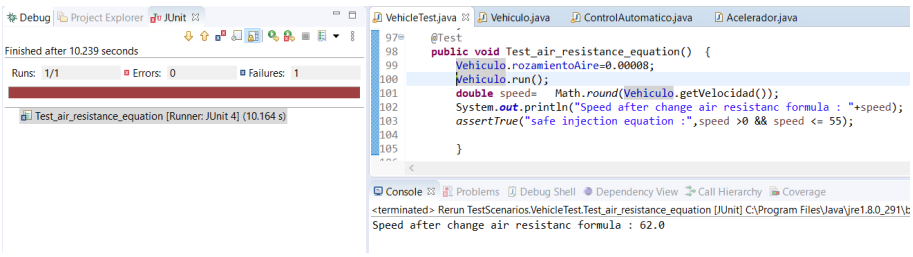

Figure 4.11: Running Test Case 5

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

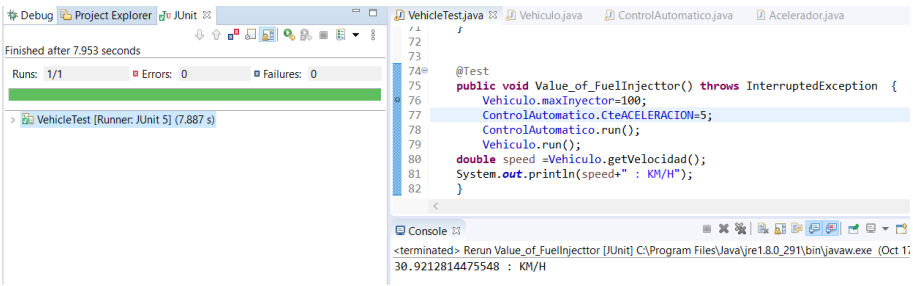

Figure 4.12: Running Test Case 6 in normal state

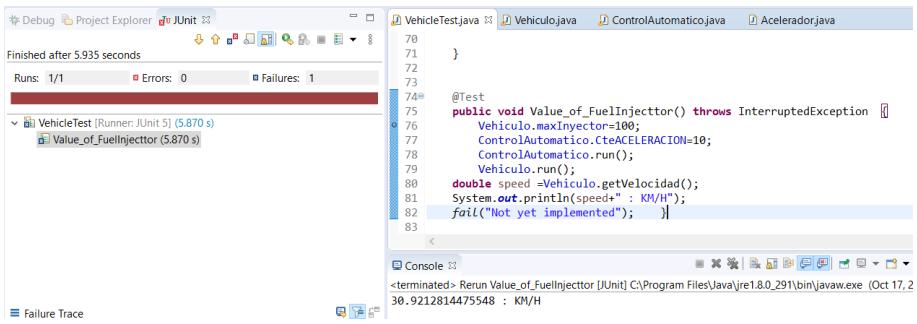

Figure 4.13: Execution of the test case 6 in unsafe state

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

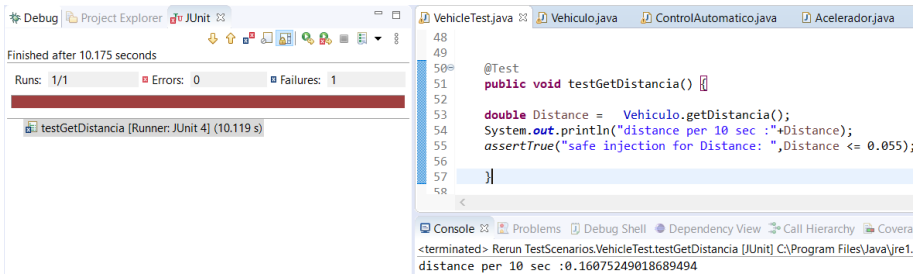

Figure 4.14: Running Test Case 7

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

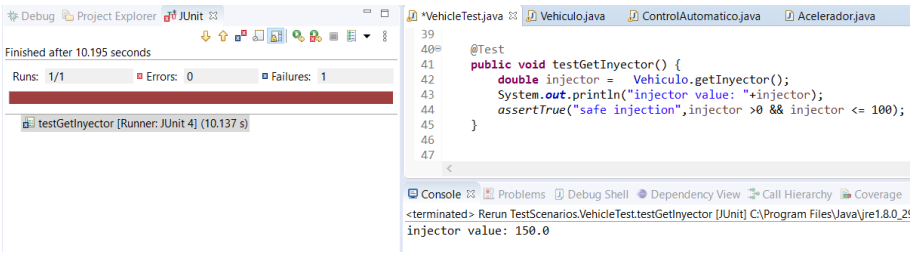

Figure 4.15: Running Test Case 8

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

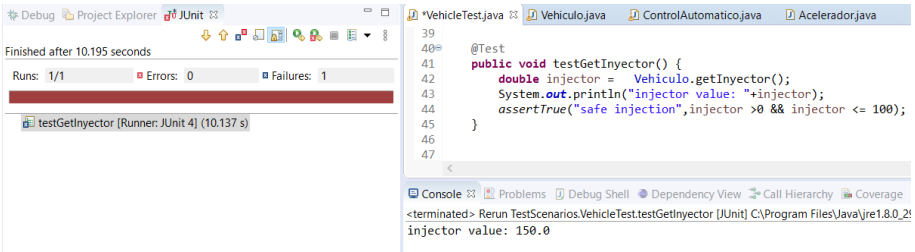

Figure 4.17: Running air resistance Test Case 9

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

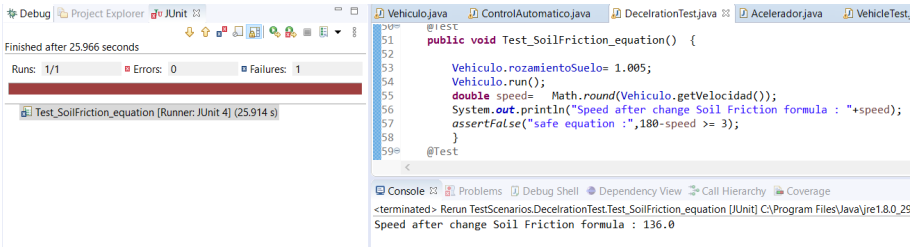

Figure 4.18: Running ground resistance Test Case 10

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

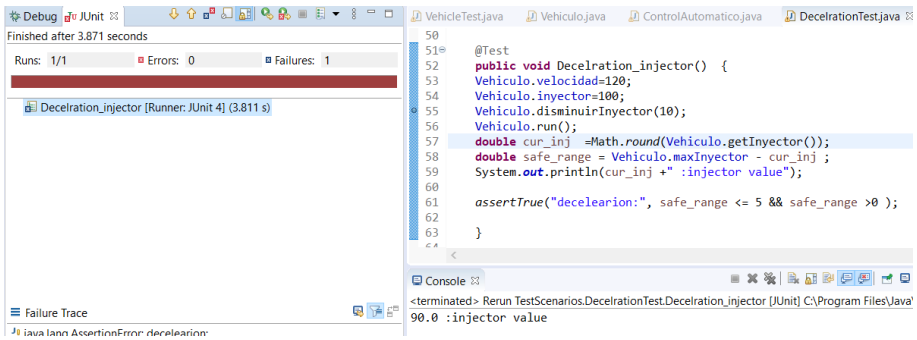

Figure 4.19: Execution of an inadequate fuel injection test case 11.

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

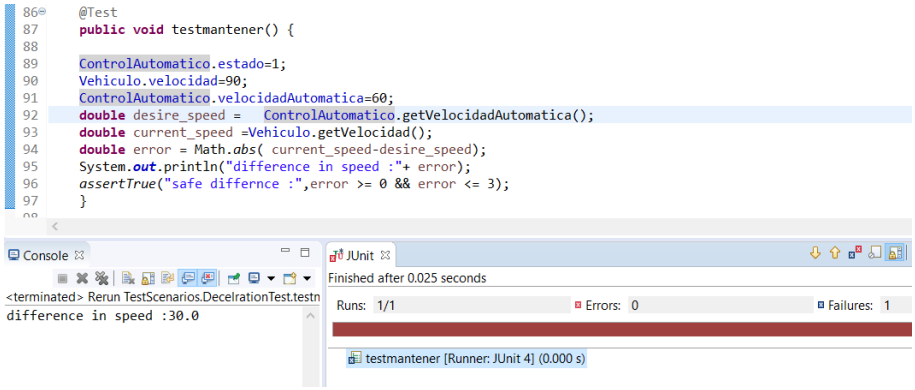

Figure 4.20: running the test case 12 for unexpected acceleration with the cruise control engaged.

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

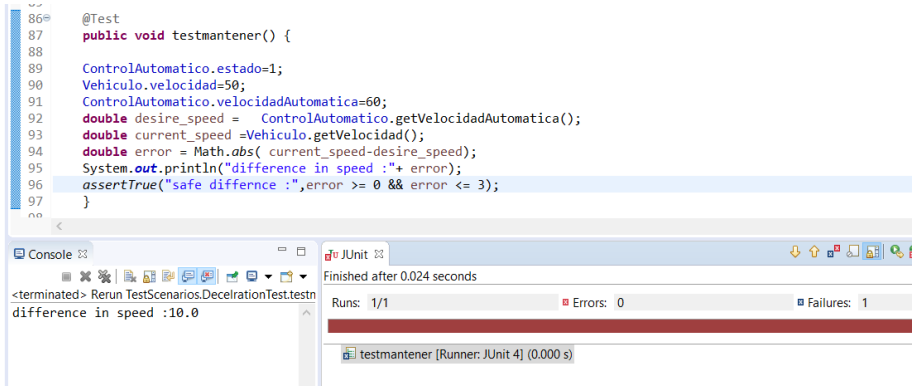

Figure 4.21: running the test case 13 for unexpected deceleration with the cruise control engaged.

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

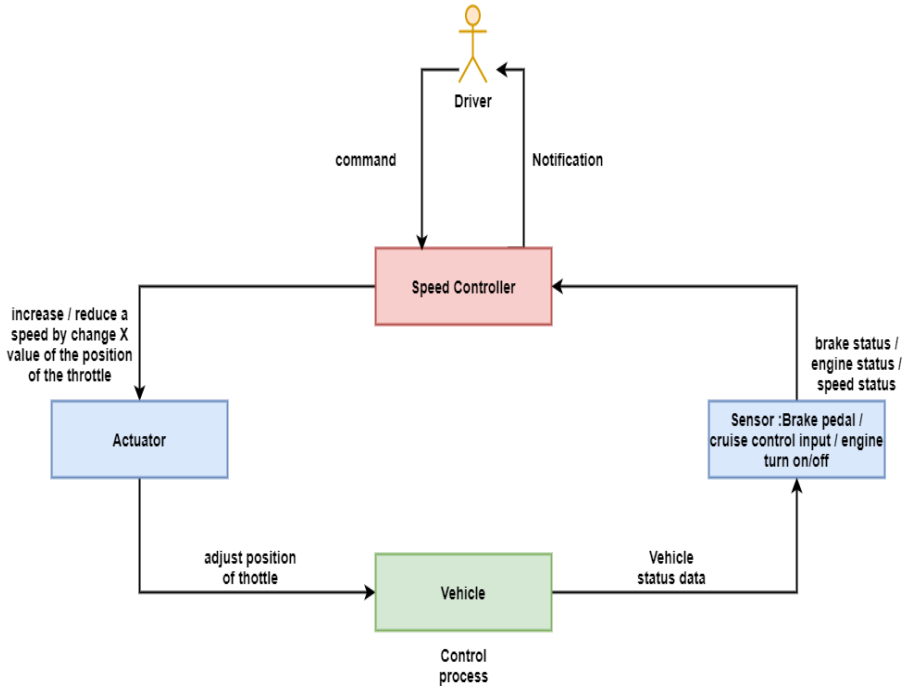

Figure 4.23: ccs control structure diagram

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.

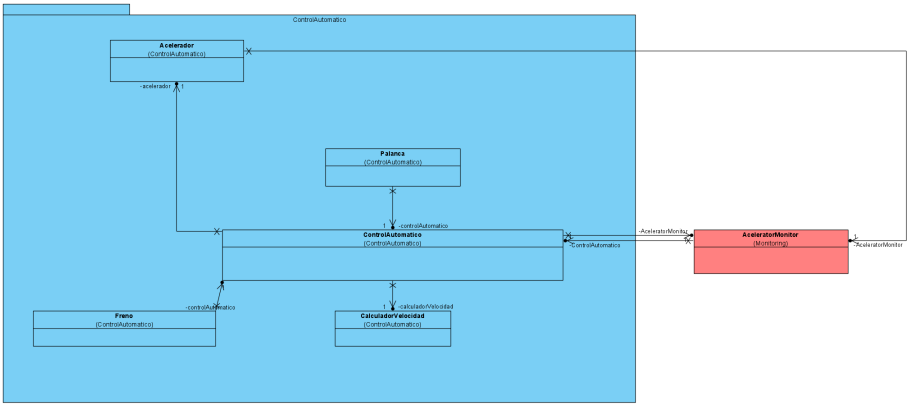

Figure 4.24: monitor acceleration domain

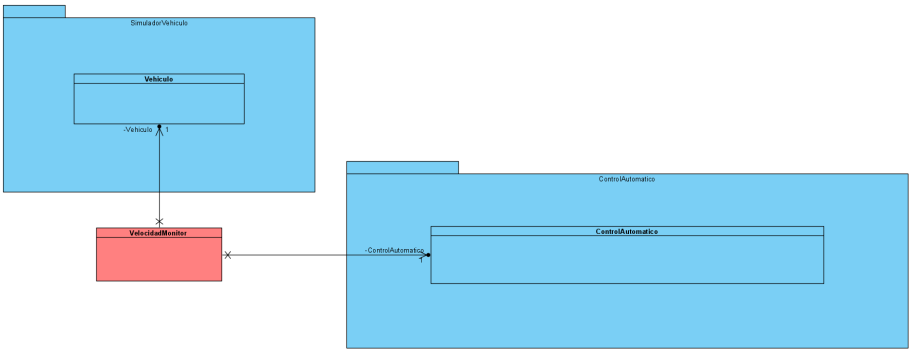

Figure 4.25: speed monitor domain

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.

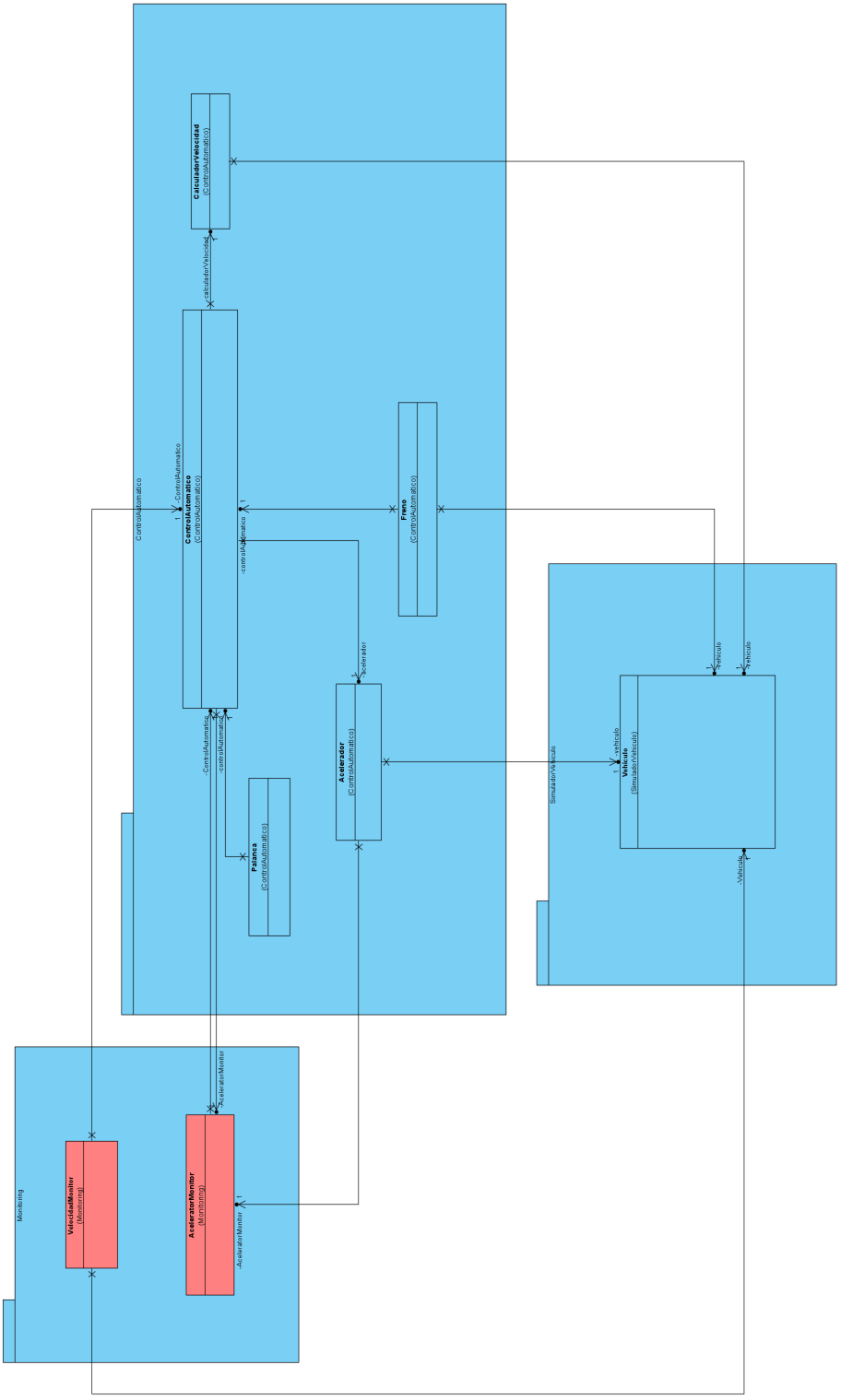

Figure 4.26: overall modeling

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.

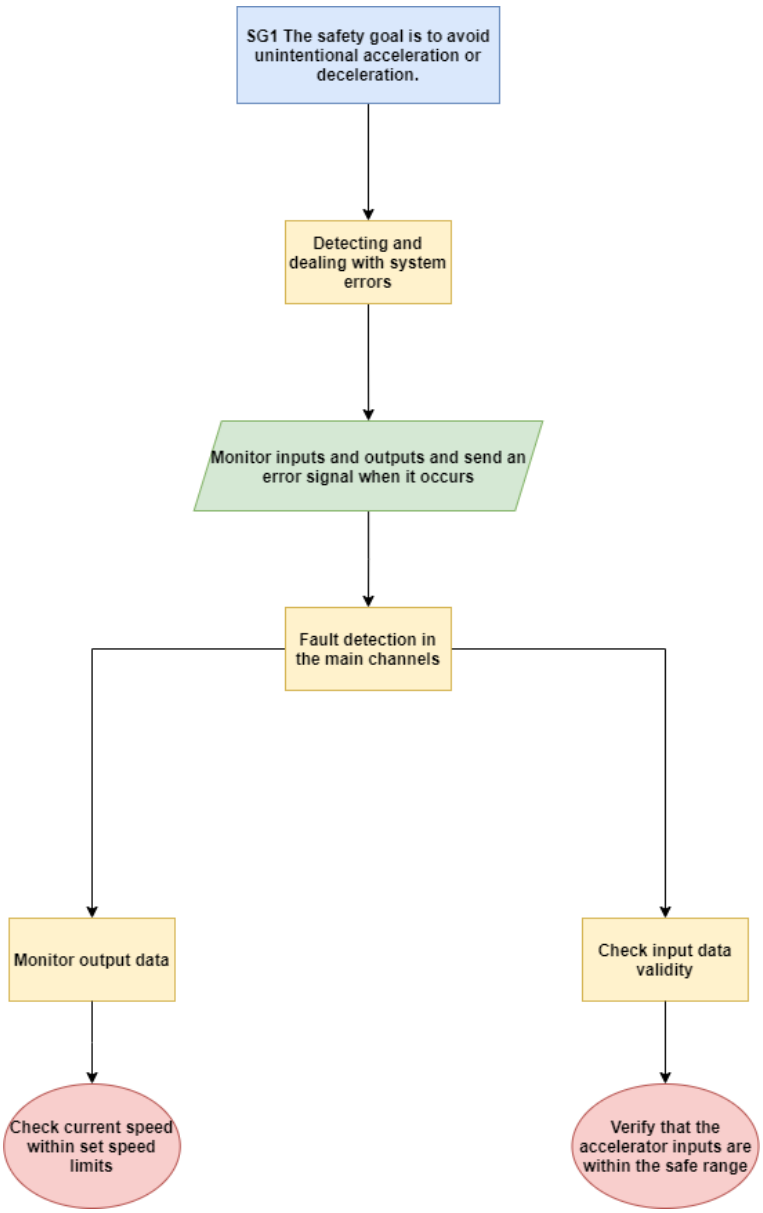

Figure 4.27: GSN diagram of safety goal

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).

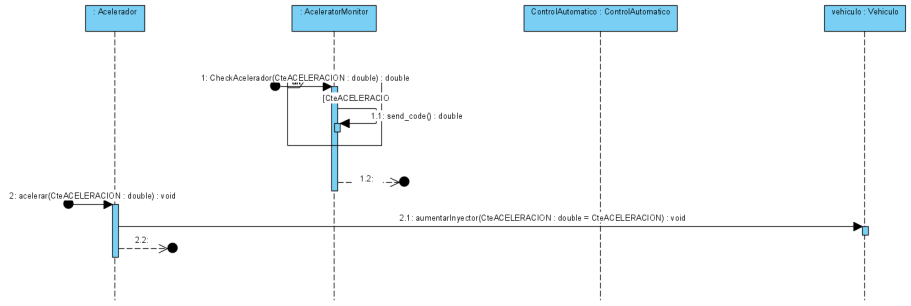

Figure 4.28: sequence diagram of acceleration monitoring

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

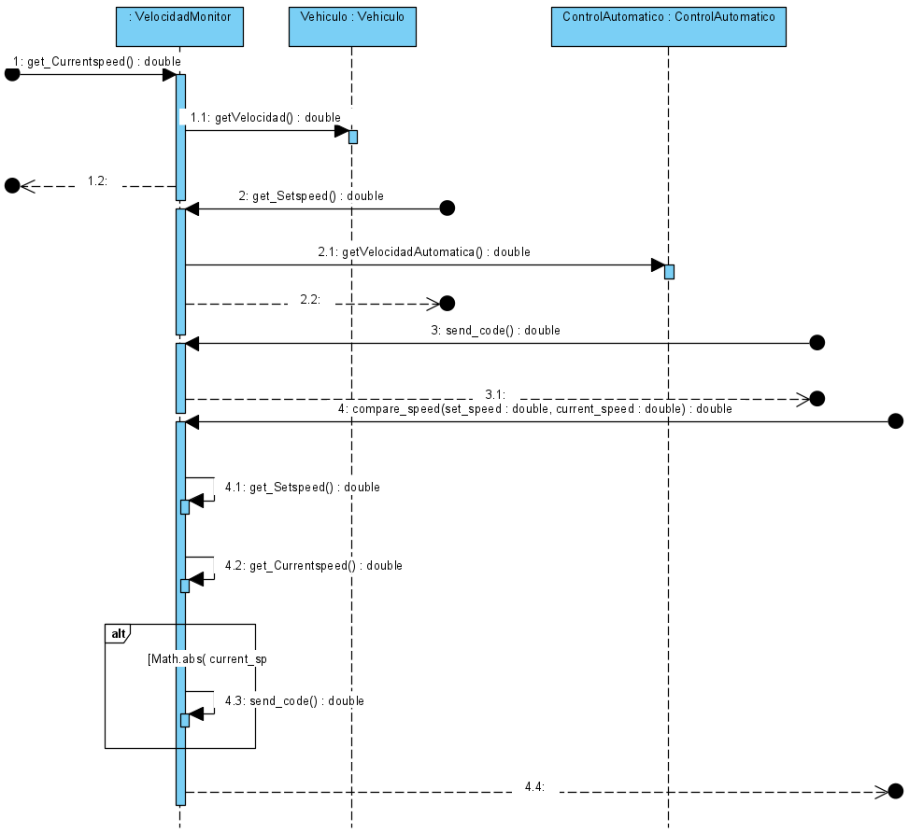

Figure 4.29: sequence diagram of monitoring current speed with set one

- 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.

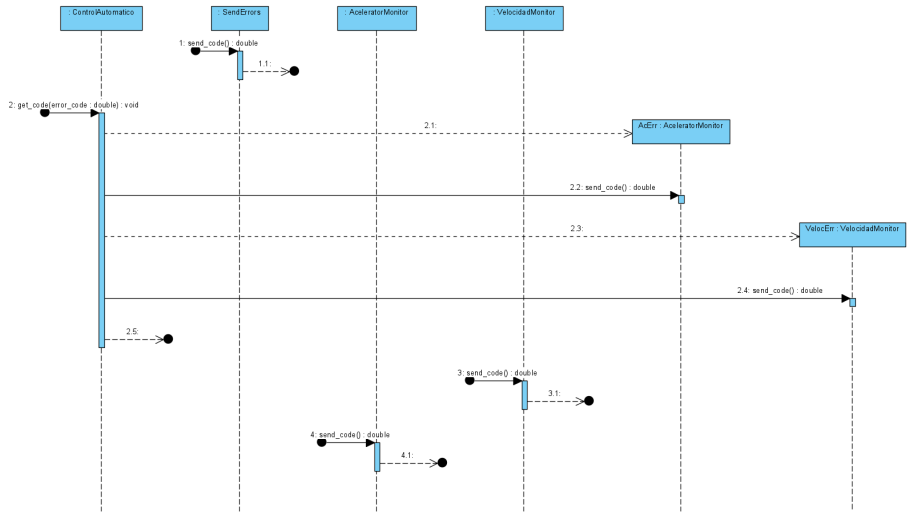

Figure 4.30: sequence diagram of sending errors code

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.

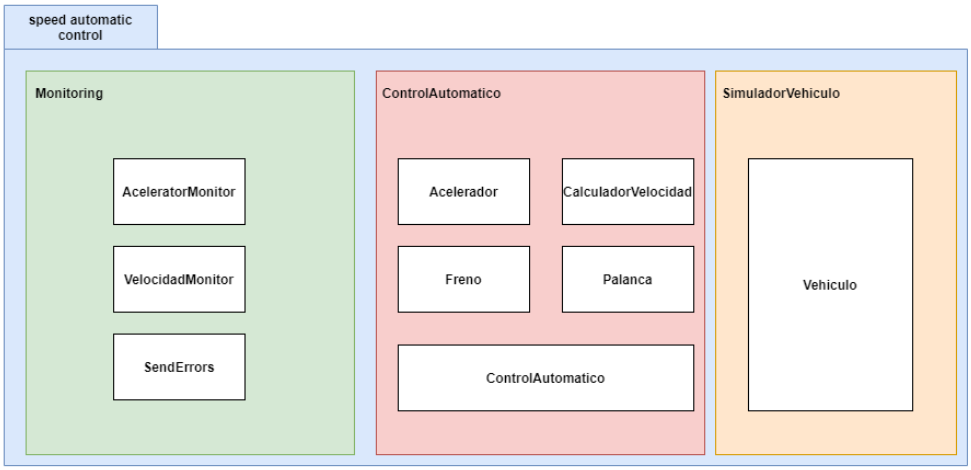

Figure 4.31: Basic hierarchical structure

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.

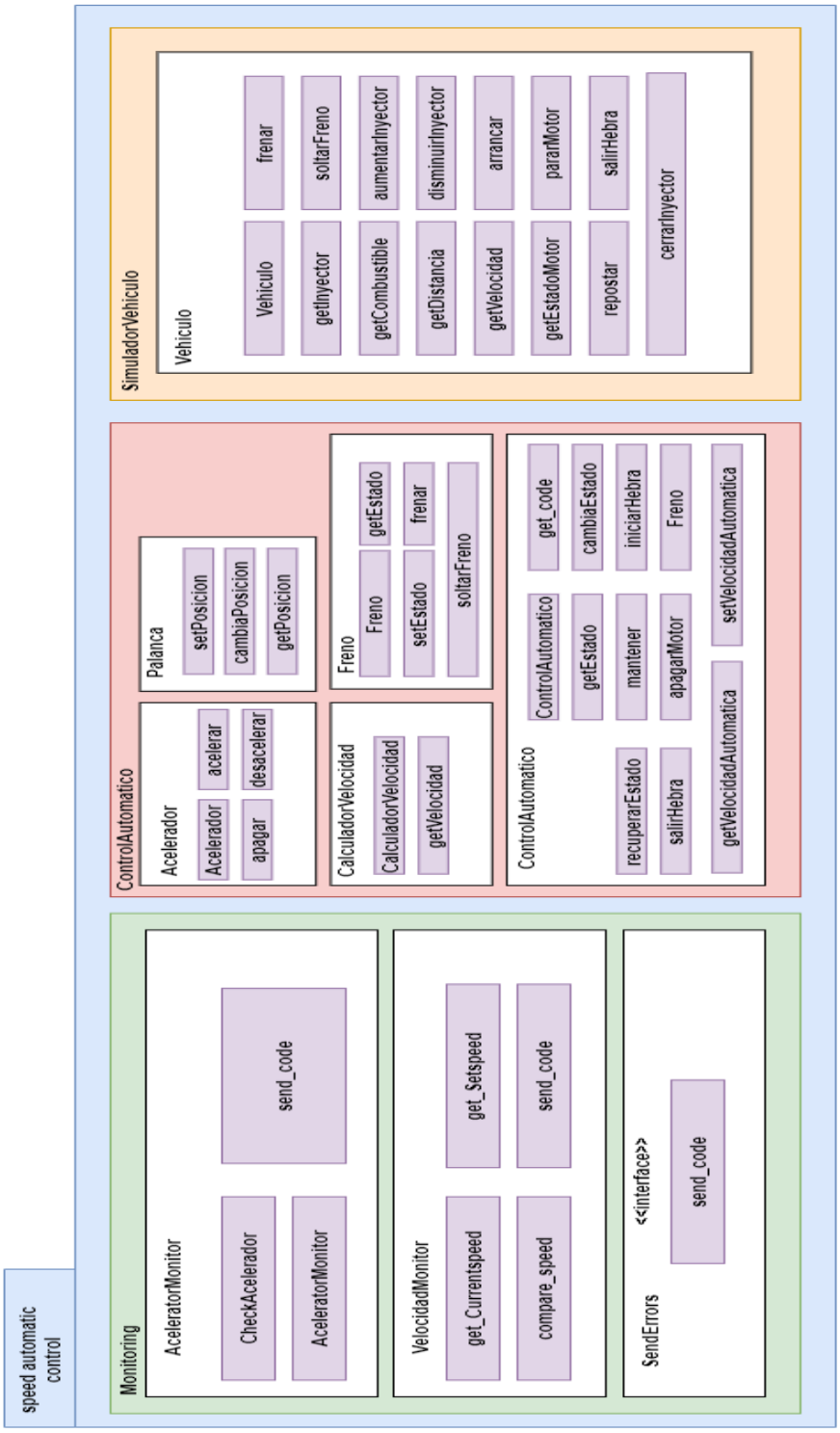

Figure 4.32: hierarchical of software components

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.

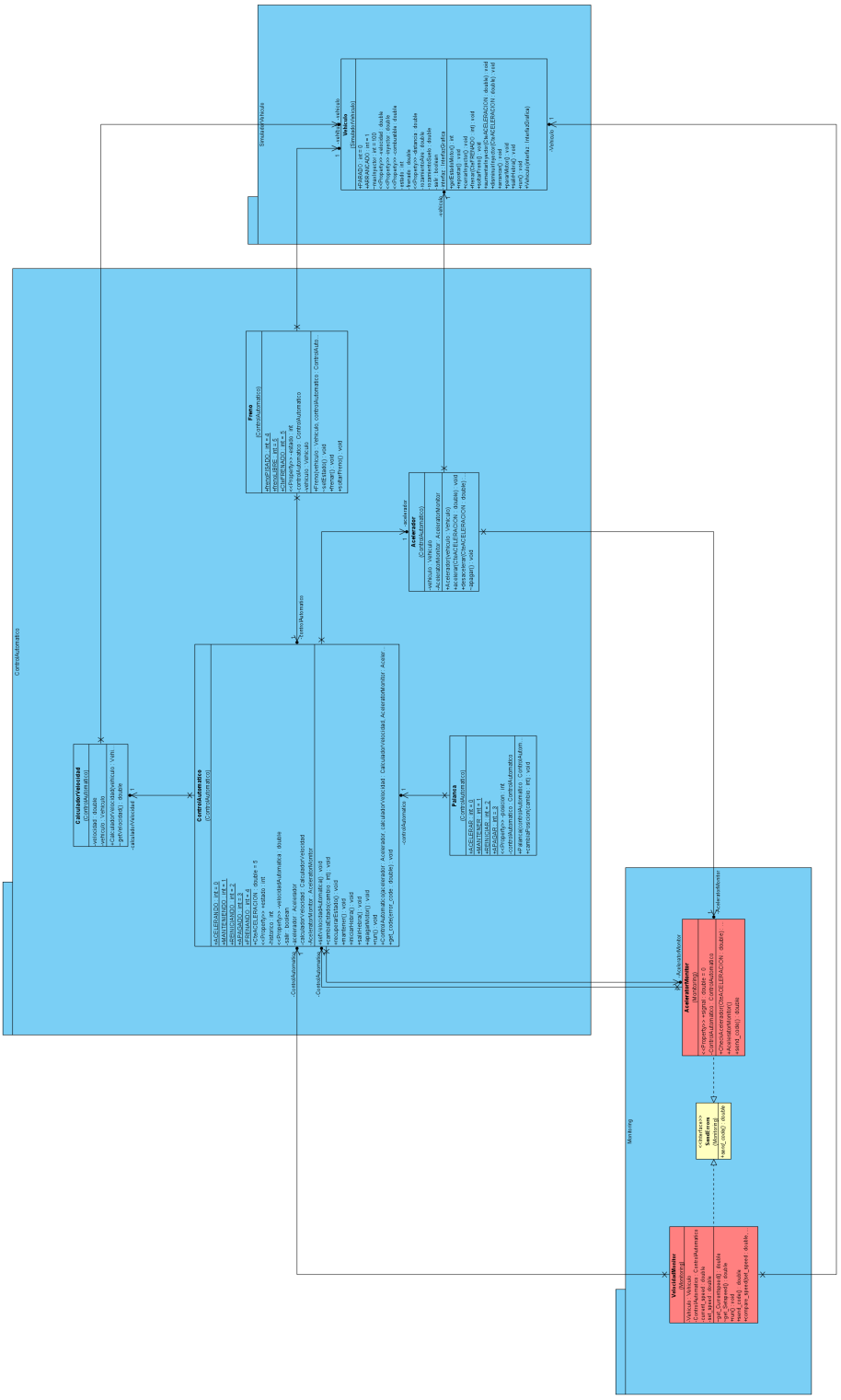

Figure 4.33: updated overall model

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

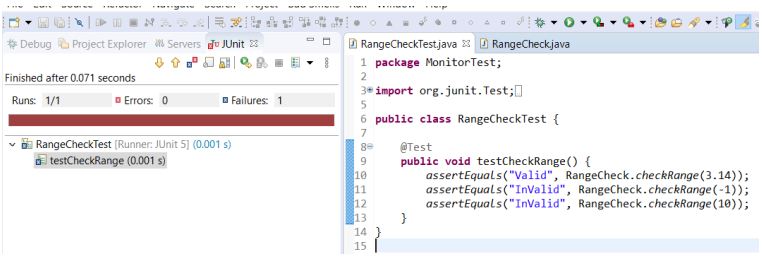

Figure 4.34: red test Running check range test case

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

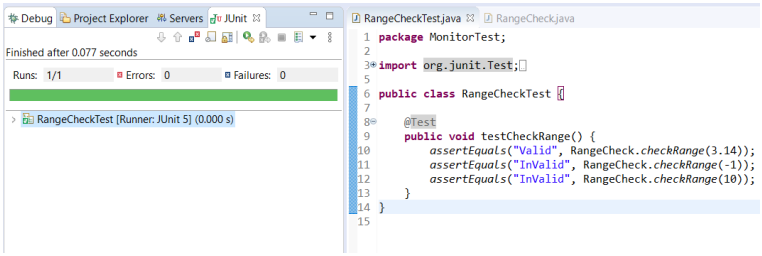

Figure 4.35: Running check range green test

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

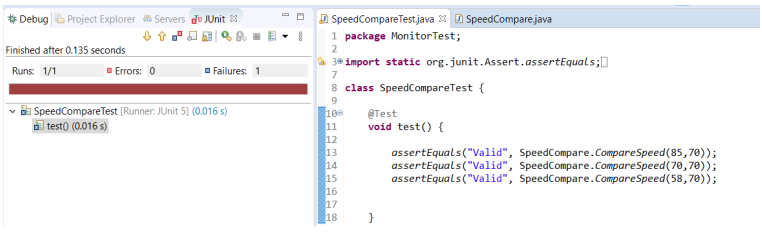

Figure 4.36: Running monitoring speed test case

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

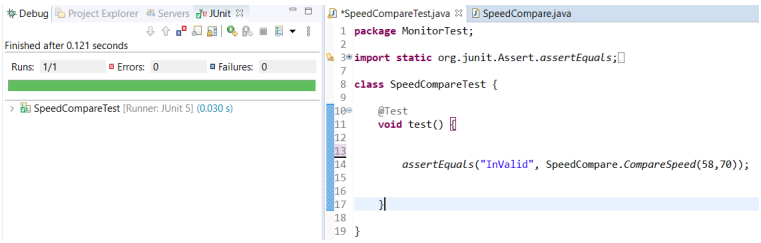

Figure 4.37: Running monitoring speed test case after refactoring

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

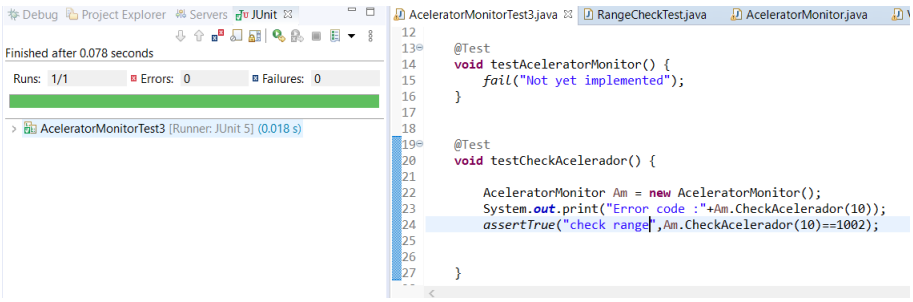

Figure 4.38: Running Accelerator monitoring test

- 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

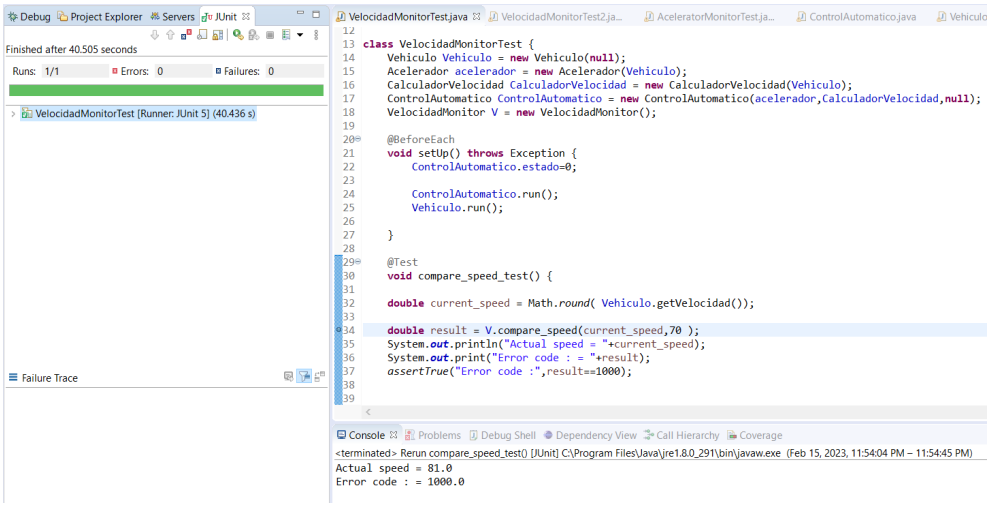

Figure 4.39: Running Velocity monitoring test

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