



Instituto Federal do Maranhão
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Robson Luan do Nascimento de Sousa

**Desenvolvimento dirigido por comportamento aplicado em um
sistema simulado de travamento de portas veicular**

Santa Inês - MA
2025

Instituto Federal do Maranhão
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Robson Luan do Nascimento de Sousa

**Desenvolvimento dirigido por comportamento aplicado em um
sistema simulado de travamento de portas veicular**

Trabalho de Conclusão de Curso apresentado
ao Instituto Federal do Maranhão, Campus
Santa Inês, como requisito da obtenção
do título de Bacharel em Engenharia de
Computação.

Orientador: Prof. Dr. Aristoteles de Almeida
Lacerda Neto
Coorientador: Prof. Msc. Emanuel Cleyton
Macedo Lemos

Santa Inês - MA
2025

ESTA SERÁ SUA FICHA CATALOGRÁFICA

ESTA SERÁ SUA FOLHA DE APROVAÇÃO

AGRADECIMENTOS

Agradeço o IFMA e ao curso de Engenharia da Computação, pela formação acadêmica e pelas oportunidades oferecidas ao longo da graduação. Aos professores Msc. Emanuel Cleyton Macedo Lemos e Dr. Aristoteles de Almeida Lacerda Neto, pela orientação e por toda a ajuda durante a minha trajetória de aprendizado. Aos meus pais, minha irmã e toda a minha família pelo suporte e pela ajuda nos momentos difíceis.

RESUMO

O desenvolvimento de sistemas veiculares modernos apresenta elevada complexidade na definição, implementação e validação de produtos — um desafio constantemente enfrentado pelas grandes montadoras. Nesse cenário, este trabalho propõe uma adaptação da metodologia de Desenvolvimento Orientado por Comportamento (Behavior-Driven Development — BDD), incorporando práticas do desenvolvimento ágil para mitigar tais complexidades, ao mesmo tempo em que possibilita a definição estruturada de funcionalidades e a geração de testes de aceitação automatizados, de forma compatível com a Engenharia Automotiva. A prova de conceito será conduzida por meio da aplicação da metodologia no desenvolvimento de um sistema simulado de travamento de portas veicular, cuja implementação é orientada pela validação do comportamento especificado e avaliada com base na aprovação final dos testes.

Palavras-chave: *BDD. Gherkin. teste-automatizado. engenharia-automotiva. desenvolvimento-ágil.*

ABSTRACT

The development of modern automotive systems presents high complexity in the definition, implementation, and validation of products — a challenge constantly faced by major automakers. In this context, this work proposes an adaptation of the Behavior-Driven Development (BDD) methodology, incorporating agile development practices to mitigate such complexities while enabling the structured definition of functionalities and the automated generation of acceptance tests, in a manner compatible with Automotive Engineering. The proof of concept will be conducted through the application of the methodology in the development of a simulated vehicle door locking system, whose implementation is guided by the validation of the specified behavior and evaluated based on the final approval of the tests.

Key-words: *BDD. Gherkin. automated-tests. automotive-engineering. agile-development.*

LISTA DE ILUSTRAÇÕES

Figura 1 – Interfaces binárias e físicas utilizadas no travamento da porta.	23
Figura 2 – História de Usuário 1: Travamento de todas as portas.	33
Figura 3 – Mecanismo de travamento da porta. Fonte: (REIF, 2015).	34
Figura 4 – Diagrama de abstração do sistema em camadas de aplicação, software básico e componentes físicos.	35
Figura 5 – História de Usuário 2: Destravamento de todas as portas.	37
Figura 6 – História de Usuário 3: Feedback de travamento.	39
Figura 7 – História de Usuário 4: <i>Keyless Access</i>	41
Figura 8 – História de Usuário 5: Travamento automático.	45
Figura 9 – Diagrama de Caixa Preta do sistema de travamento de portas.	46
Figura 10 – Primeiro cenário <i>Gherkin</i> da história de usuário 1.	51
Figura 11 – Segundo cenário <i>Gherkin</i> da história de usuário 1.	52
Figura 12 – Primeiro cenário <i>Gherkin</i> da história de usuário 2.	52
Figura 13 – Segundo cenário <i>Gherkin</i> da história de usuário 2.	53
Figura 14 – Primeiro cenário <i>Gherkin</i> da história de usuário 3.	53
Figura 15 – Segundo cenário <i>Gherkin</i> da história de usuário 3.	54
Figura 16 – Primeiro cenário <i>Gherkin</i> da história de usuário 4.	55
Figura 17 – Segundo cenário <i>Gherkin</i> da história de usuário 4.	56
Figura 18 – Terceiro cenário <i>Gherkin</i> da história de usuário 4.	57
Figura 19 – Primeiro cenário <i>Gherkin</i> da história de usuário 5.	58
Figura 20 – Segundo cenário <i>Gherkin</i> da história de usuário 5.	59
Figura 21 – Terceiro cenário <i>Gherkin</i> da história de usuário 5.	59
Figura 22 – Quarto cenário <i>Gherkin</i> da história de usuário 5.	60
Figura 23 – Modelo feature_model.slx que aplica o conceito de caixa preta. . . .	63
Figura 24 – Relatório de testes - resultado da execução do arquivo .feature	64
Figura 25 – Relatório de testes - resultados por cenário.	64
Figura 26 – Diagrama de estados da lógica do auto travamento.	71
Figura 27 – Relatório de testes - execução final.	73

LISTA DE QUADROS

Quadro 1 – Valores possíveis para as interfaces de entrada e saída	47
Quadro 2 – Falhas de comportamento capturadas durante a modelagem iterativa .	74

SUMÁRIO

1	INTRODUÇÃO	11
1.1	JUSTIFICATIVA	11
1.2	OBJETIVOS	13
1.2.1	Objetivo Geral	13
1.2.2	Objetivos Específicos	13
1.3	ORGANIZAÇÃO DO TRABALHO	13
2	FUNDAMENTAÇÃO TEÓRICA	16
2.1	DESENVOLVIMENTO ORIENTADO A COMPORTAMENTO - BDD	16
2.2	DEFINIÇÃO DAS FUNCIONALIDADES DO SISTEMA POR MEIO DE HISTÓRIAS DE USUÁRIO E MAPEAMENTO DE EXEMPLOS	17
2.3	A LINGUAGEM GHERKIN	19
2.4	DESIGN ORIENTADO POR MODELO	21
2.5	SISTEMAS DE TRAVAMENTO DE PORTA VEICULARES . . .	23
3	METODOLOGIA	25
3.1	PROCEDIMENTOS METODOLÓGICOS	25
3.2	FERRAMENTAS UTILIZADAS	27
3.3	CRITÉRIOS DE AVALIAÇÃO	28
4	IMPLEMENTAÇÃO	29
4.1	ETAPA 1: DESCRIÇÃO DAS HISTÓRIAS DE USUÁRIO	29
4.2	ETAPA 2: MAPEAMENTO DE EXEMPLOS	31
4.2.1	História de usuário 1	31
4.2.2	História de usuário 2	36
4.2.3	História de usuário 3	37
4.2.4	História de usuário 4	40
4.2.5	História de usuário 5	42
4.3	ETAPA 3: DESENHO DO DIAGRAMA DE CAIXA PRETA . . .	45
4.4	ETAPA 4: DESENVOLVIMENTO DOS CENÁRIOS GHERKIN	48
4.4.1	História de usuário 1	50
4.4.2	História de usuário 2	52
4.4.3	História de usuário 3	53
4.4.4	História de usuário 4	55

4.4.5	História de usuário 5	57
5	RESULTADOS	62
5.1	MODELAGEM ITERATIVA DO SISTEMA EM SIMULINK	62
5.1.1	Execução dos testes de aceitação da História de Usuário 1	65
5.1.2	Execução dos testes de aceitação da História de Usuário 2	66
5.1.3	Execução dos testes de aceitação da História de Usuário 3	67
5.1.4	Execução dos testes de aceitação da História de Usuário 4	69
5.1.5	Execução dos testes de aceitação da História de Usuário 5	71
5.2	ANÁLISE DOS RESULTADOS	72
6	CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS	77
6.1	TRABALHOS FUTUROS	78
	REFERÊNCIAS	80

1 INTRODUÇÃO

A indústria automotiva tem experimentado um crescimento constante, acompanhado por um aumento significativo na modernização dos veículos e pela intensa competição entre montadoras para entregar produtos cada vez mais robustos e de alta qualidade. Esse cenário traz consigo um aumento na complexidade do desenvolvimento, demandando soluções mais eficientes e seguras para sistemas cada vez mais sofisticados e que sejam capazes de atender às necessidades específicas de cada cliente.

A modernização do setor automotivo tem conduzido ao próximo grande avanço: os veículos definidos por software (*Software-Defined Vehicles*) (FINIO; DOWNIE, 2025). Essa abordagem possibilita a customização contínua e a evolução do produto mesmo após a aquisição, permitindo a entrega de novas funcionalidades ao veículo por meio de atualizações de software feitas remotamente *Over-the-Air (OTA)*, suportadas pelo hardware já embarcado.

Diante desse contexto, a Engenharia Automotiva enfrenta o desafio de projetar e desenvolver produtos com um nível altíssimo de qualidade, ao mesmo tempo em que busca reduzir erros e retrabalho. A aplicação de metodologias ágeis, como o *Behavior Driven Development (BDD)* (NORTH, 2006), oferece uma abordagem promissora, pois foca na definição do comportamento esperado do sistema a partir da perspectiva do usuário. Essa abordagem permite uma colaboração mais efetiva entre os membros de uma equipe ágil (Atlassian, n.d.), facilita a descoberta de funcionalidades e possibilita que o desenvolvimento seja guiado por testes de aceitação automatizados, promovendo entregas de valor de forma mais rápida e com maior confiabilidade.

1.1 JUSTIFICATIVA

O problema central deste trabalho é: “como é possível aplicar o processo de BDD para facilitar a definição do sistema e guiar seu desenvolvimento no contexto da Engenharia de Sistemas Automotivos?”. A solução proposta consiste na adaptação do processo, integrando as técnicas já consolidadas do BDD com metodologias amplamente utilizadas na indústria automotiva, de forma a lidar com as complexidades inerentes ao desenvolvimento de sistemas veiculares.

Este trabalho insere-se no contexto atual da indústria automotiva, adotando o sistema de travamento de portas como prova de conceito para a geração do produto. Durante seu desenvolvimento, serão consideradas como referência implementações reais de sistemas já presentes em veículos, como os da Volkswagen (VOLKSWAGEN, 2025) e da Bosch (GSMH, 2022; REIF, 2015). Nesse sentido, a adaptação do processo de BDD será

aplicada ao desenvolvimento de um produto compatível com as especificações industriais, ao mesmo tempo em que busca minimizar as complexidades enfrentadas em sua execução.

O BDD incorpora diversas premissas que surgem das lições aprendidas na aplicação de outras metodologias ágeis, definindo o desenvolvimento do produto a partir do comportamento que gera valor ao usuário. Além disso, ele busca promover a colaboração entre as diferentes etapas do desenvolvimento, garantindo que toda a equipe compreenda o produto e consiga assumir a perspectiva do usuário.

Por outro lado, a Engenharia Automotiva apresenta uma complexidade significativa, o que torna desafiadora a entrega de produtos de alta qualidade. A definição de um sistema veicular envolve muito mais do que software ou eletrônica, sendo necessário integrar componentes mecânicos, considerar falhas potenciais, garantir a rastreabilidade de requisitos e contemplar casos de uso extremos.

O principal desafio deste trabalho consiste em identificar técnicas capazes de capturar as complexidades envolvidas no desenvolvimento de sistemas automotivos e adaptá-las às premissas do BDD. Em casos como o do sistema de travamento de portas, isso é traduzido na necessidade de atender a funcionalidades essenciais, como (GSMH, 2022):

- Permitir o acesso ao interior do veículo para todas as pessoas autorizadas;
- Gerenciar a abertura das portas a partir das maçanetas internas e externas;
- Proteger os ocupantes da abertura indevida das portas;
- Proteger o veículo contra furtos enquanto ele está estacionado.

Uma das principais complexidades desse tipo de sistema está na definição do mecanismo empregado no travamento das portas. Conforme descrito por Reif (2015), podem ser utilizadas travas mecânicas ou eletrônicas, cada um com conjuntos específicos de componentes e lógicas de software distintas que influenciam diretamente o desenvolvimento. Esse desafio é recorrente na indústria automotiva, na qual diversos sistemas dependem de decisões técnicas de hardware ou soluções mecânicas para viabilizar o início do processo de design.

Neste trabalho, será apresentada uma metodologia que demonstra como a aplicação do Desenvolvimento Orientado a Comportamento (BDD) possibilita projetar o sistema de forma independente dos detalhes técnicos, facilitando tanto o desenvolvimento das funcionalidades quanto a execução dos testes. Isso ocorre porque a abordagem do BDD prioriza a perspectiva do usuário e a especificação do comportamento esperado do sistema, os quais não estão diretamente vinculados à implementação.

1.2 OBJETIVOS

1.2.1 Objetivo Geral

Aplicar o levantamento e definição de funcionalidades na análise de um sistema de travamento de portas veicular, seguido do desenvolvimento guiado por testes de aceitação automatizados, demonstrando como essas práticas podem contribuir para o desenvolvimento de software automotivo de forma mais organizada e eficiente.

1.2.2 Objetivos Específicos

- Realizar o levantamento e definição de funcionalidades considerando possíveis limitações decorrentes da dependência de hardware e de sistemas mecânicos;
- Analisar as complexidades dos sistemas automotivos de maneira compatível com o processo de BDD;
- Empregar técnicas de desenvolvimento como padrão de escrita de software AUTOSAR (AUTOSAR, n.d.) e Design Orientado por Modelos (*Model-Based Design - MBD*) (MathWorks, 2020), que são amplamente utilizadas na indústria automotiva;
- Definir um escopo de desenvolvimento que possibilite a realização do design do sistema forma clara e compreensível;
- Empregar testes de aceitação automatizados para guiar o desenvolvimento do sistema, consistente com o design definido;
- Realizar a implementação do sistema de maneira iterativa, possibilitando à descoberta de melhorias surgidas durante a aplicação do processo;
- Aplicar a metodologia desenvolvida na produção de um sistema de travamento de portas veicular simulado.

1.3 ORGANIZAÇÃO DO TRABALHO

No Capítulo 2, serão apresentados os conceitos que compõem a fundamentação teórica deste trabalho, abrangendo:

- **O processo de BDD** - histórico do desenvolvimento do processo e seus pilares fundamentais do processo;
- **Mapeamento de exemplos** - metodologia aplicada para realizar a análise do sistema e definição de funcionalidades;

- **A linguagem *Gherkin*** - linguagem usada para a escrita de cenários, que atuam como definição de comportamentos e como testes de aceitação;
- **Design orientado por modelo** - metodologia adotada para a implementação do sistema;
- **Sistemas de travamento de porta veiculares** - definição e caracterização dos sistemas de segurança voltados ao travamento de portas em veículos.

Em seguida, no Capítulo 3, será apresentada a metodologia adotada, descrevendo o procedimento de desenvolvimento do produto, estruturado em seis etapas. Serão também detalhadas as ferramentas utilizadas neste trabalho, bem como os critérios de avaliação, definidos a partir da execução dos testes de aceitação de cada funcionalidade do sistema.

No Capítulo 4, a implementação é conduzida por meio da execução das quatro primeiras etapas do processo. Essa fase tem início com a definição das histórias de usuário — que representam as funcionalidades do sistema — conforme descrito na Seção 4.1. As histórias orientam todas as etapas subsequentes, desde a elaboração dos testes de aceitação até a modelagem do sistema.

Para cada história de usuário, o mapeamento de exemplos, descrito na Seção 4.2, será empregado para orientar a definição dos detalhes relacionados ao comportamento esperado. Em seguida, na Seção 4.3, o sistema será modelado como uma abstração que explicita as interações com o usuário, ao mesmo tempo em que oculta os aspectos técnicos de sua implementação. A partir das histórias de usuário e do modelo do sistema, será elaborada, na Seção 4.4, uma série de cenários que representam tanto a especificação do comportamento esperado quanto os testes de aceitação correspondentes.

No Capítulo 5, serão executados os testes de validação definidos pelos cenários das histórias de usuário. Esse processo assegura que a modelagem do sistema seja orientada pela execução dos testes e que o comportamento especificado seja atendido. Ela também possibilita a identificação e a correção de falhas detectadas durante a execução dos testes.

A aprovação das histórias de usuário será realizada a partir da verificação de que:

- Os testes foram executados e confirmaram a aprovação de todos os cenários relacionados à história;
- A execução dos cenários é repetida ao longo do desenvolvimento das demais histórias, de modo a assegurar que o comportamento previamente validado não seja comprometido.

Por fim, no Capítulo 6, será apresentada uma síntese dos principais aspectos observados durante a aplicação da metodologia proposta. Adicionalmente, serão discutidas

algumas limitações do escopo deste trabalho, acompanhadas dos passos futuros necessários para a continuidade do desenvolvimento e geração do produto físico.

2 FUNDAMENTAÇÃO TEÓRICA

Este trabalho tem sua fundamentação no Desenvolvimento Orientado a Comportamento (BDD), com foco em sua adaptação para aplicação em desenvolvimento de sistemas de software automotivo. Esse processo será aplicado na definição, implementação e validação de um sistema de travamento de portas veicular.

2.1 DESENVOLVIMENTO ORIENTADO A COMPORTAMENTO - BDD

A metodologia de desenvolvimento orientado a comportamento foi inicialmente proposta por Dan North (NORTH, 2006), que compartilhou em seu blog as lições aprendidas ao aplicar e ensinar o processo de desenvolvimento orientado a testes (TDD). Segundo o autor, o BDD evoluiu além de práticas ágeis e é designado para fazê-las mais acessíveis e efetivas para times que são novos no desenvolvimento ágil.

A evolução do processo de BDD foi impulsionada por diversas descobertas práticas que surgiram em resposta às dificuldades enfrentadas na aplicação do TDD. A mudança da nomenclatura de “teste” para “comportamento”, por exemplo, é aparentemente simples, mas gerou um impacto profundo. Essa nova terminologia ajudou a consolidar a ideia de que os testes devem sempre validar o comportamento esperado do sistema, o que, por sua vez, reduziu significativamente dúvidas comuns entre desenvolvedores iniciantes.

Outra mudança importante nesse processo foi a adoção do termo *should* (deveria) em vez de *shall* (deve) ou *will* (vai) na formulação dos nomes dos testes. Desta forma, o teste se torna uma sentença natural e provoca o questionamento se o comportamento demonstrado deveria ser realizado pelo sistema. Esse tipo de reflexão é especialmente valioso em equipes ágeis (Atlassian, n.d.), pois estimula o diálogo - principalmente em discussões que envolvem profissionais com diferentes níveis de conhecimento técnico ou de domínio do negócio.

O processo consolidado por Dan North tem foco no desenvolvimento robusto e traz diversas práticas voltadas à entregar valor ao usuário de forma mais rápida. Esse amadurecimento é evidente nas características apontadas por Solis e Wang (2011) sobre o processo de BDD em sua forma atual, que consistem de:

1. **Linguagem Ubíqua** - uma linguagem comum compreensível tanto por desenvolvedores quanto por especialistas do domínio ou do negócio. Deve ser utilizada ao longo de todo o processo de desenvolvimento do produto, assegurando uma comunicação clara e evitando ambiguidades entre as partes envolvidas;

2. **Processo de decomposição iterativa** - facilita a priorização do desenvolvimento de funcionalidades para o sistema ao delinear o valor gerado para o cliente final;
3. **Descrições em texto simples com histórias de usuário e templates de cenários** - emprega templates que contém uma mistura de palavras chaves e linguagem natural na criação de histórias de usuário e de cenários, garantindo sua acessibilidade e facilidade de entendimento;
4. **Testes de aceitação automatizados com regras de mapeamento** - aplica especificações executáveis na forma de cenários compostos de passos que são traduzidos em testes, os quais orientam o desenvolvimento.

A validação das funcionalidades definidas durante o processo de BDD, consiste na criação de especificações executáveis na forma de cenários que servem de critérios de aceitação das histórias de usuário. Os cenários utilizam um padrão de escrita que incorpora uma junção de palavras-chave com linguagem natural e podem ter interface com o código desenvolvido por meio de diversas ferramentas como *JBehave*, *RSpec* (Cucumber.io, 2024b) e a que será aplicado neste trabalho - Cucumber (Cucumber.io, 2024a) em junção com a biblioteca do Python chamada Behave (Behave, 2025).

2.2 DEFINIÇÃO DAS FUNCIONALIDADES DO SISTEMA POR MEIO DE HISTÓRIAS DE USUÁRIO E MAPEAMENTO DE EXEMPLOS

Segundo Rehkopf (2025), histórias de usuário são “uma explicação informal de uma funcionalidade de software escrita na perspectiva do usuário”. Elas representam uma unidade de trabalho, um objetivo de desenvolvimento a ser alcançado e garante que a equipe seja capaz de colaborar e trazer soluções inovadores com o usuário final em foco.

Para isso, adota-se um formato que se utiliza de linguagem natural e assegura a definição de três elementos fundamentais:

- Quem é o usuário?
- Qual funcionalidade deve ser implementada?
- Qual valor essa funcionalidade traz ao usuário?

Esses três pontos são organizados em uma estrutura padronizada que utiliza conjunções específicas, resultando no seguinte modelo: “*Como [tipo de usuário], eu quero [funcionalidade], para que [valor gerado].*” Assim como destacado por Dan North “[...] sua força está em forçar que você identifique o valor de entregar uma história quando você a define [...]” (NORTH, 2006). Essa ênfase em entender o valor entregue ao usuário - desde

a formulação das histórias - é essencial para garantir que as funcionalidades desenvolvidas estejam alinhadas com as reais necessidades do usuário.

Para garantir um bom fluxo nas discussões durante a definição do escopo das histórias, o exemplo prático da aplicação de BDD demonstrado em Lawrence e Rayner (2019) destaca o uso do mapeamento de exemplos como uma forma eficaz para explorar e esclarecer os comportamentos esperados do sistema. Esse método contribui para que toda a equipe tenha uma compreensão da funcionalidade em questão e evita desvios no foco da conversa.

Dessa maneira, a definição dos comportamentos deve ser feita por meio de discussões que incluem toda a equipe e começa com citações de exemplos de uso real, priorizando os cenários de *happy path* - aqueles em que o sistema se comporta conforme o esperado em condições normais. Estes exemplos devem ser sempre descritos a partir da perspectiva do usuário e podem ser trazidos de forma iterativa, sem a necessidade de que todos os casos estejam definidos antes do início do desenvolvimento.

Neste estágio do processo, existem ainda muitas incertezas acerca do sistema, o que pode dificultar a definição de comportamentos concretos que sustentem tecnicamente os exemplos levantados. Essa dificuldade é ainda maior quando não se tem uma descrição física do produto final ou detalhes de implementação - como no caso deste trabalho onde o mecanismo do travamento da porta ainda não está definido durante a etapa de mapeamento de exemplos.

Para lidar com tantos pontos desconhecidos durante discussões com a equipe, costuma-se aplicar o mapeamento de exemplos com o uso de notas adesivas coloridas, assim como descrito na metodologia de Wynne (2015). Ela se utiliza de notas coloridas em 4 cores diferentes que representam diferentes pontos:

- **Amarelo:** História de usuário ou funcionalidade;
- **Azul:** Regra de funcionamento da história;
- **Verde:** Exemplo concreto que demonstra uma regra;
- **Vermelho:** Pergunta que foge do escopo da discussão e que deve ser investigada futuramente.

Dessa maneira, todos os pontos desconhecidos que foram previamente identificados são registrados como perguntas, evitando que a discussão se desvie para tópicos fora do escopo naquele momento. Essas questões são respondidas após a conclusão do mapeamento de exemplos, com base na bibliografia disponível.

Após as dúvidas serem anotadas, a discussão segue focada na experiência do usuário, mesmo que alguns aspectos técnicos ainda não estejam totalmente esclarecidos. Neste caso,

assume-se que o veículo existe e que suas portas funcionam “magicamente”, conforme os comportamentos definidos, permitindo que a discussão possa ser prosseguida com os exemplos concretos que devem ser descritos usando as notas verdes.

2.3 A LINGUAGEM GHERKIN

A linguagem *Gherkin* refere-se ao padrão definido pelo *Cucumber* (Cucumber.io, 2024a) para a escrita dos cenários do BDD. Sua estrutura baseia-se no uso das palavras-chave *Given/When/Then* (Dado que/Quando/Então), juntamente com linguagem natural para tornar os cenários compreensíveis tanto para humanos quanto para máquinas.

A estrutura dos cenários é organizada em arquivos com extensão **.feature**, próprios do *Cucumber*. Esses arquivos contêm conjuntos de cenários que descrevem os seguintes componentes do teste:

- *Given* - estado inicial ou pré-condições.
- *When* - evento ou transição de estado.
- *Then* - estado final do sistema ou ação.

Cada componente *Given/When/Then* é denominado passo, sendo responsável por descrever uma ação a ser executada. O conjunto de passos de um cenário constitui a sequência de processos que orienta a execução do teste, resultando em uma especificação capaz de validar o comportamento esperado do sistema.

Para isso, o cenário descreve como o sistema deve transitar de um estado inicial - definido pela cláusula *Given* - para um estado final - especificado pelo *Then* - em resposta a uma mudança ou evento descrito pelo *When*. Um exemplo disso pode ser demonstrado da seguinte forma:

Scenario: Derrubar dominós

Given eu organizei vários dominós em pé numa fila reta

When eu derrubar o primeiro dominó

Then todos os dominós deveriam cair em sequência

Este exemplo ilustra como a gramática *Gherkin* pode ser utilizada para representar um cenário real na qual dominós são enfileirados e, ao se derrubar o primeiro, ocorre uma reação em cadeia. Ele demonstra uma clara mudança de estado que ocorre como uma resposta ao evento que é derrubar o primeiro dominó.

Um exemplo aplicado a um sistema de software para travamento de portas - como no caso abordado neste trabalho - pode ser descrito da seguinte forma:

Scenario: Travando o carro

Given o meu carro está destravado

When eu pressiono o botão de travamento das portas

Then o carro deveria ser travado

Neste exemplo, o comportamento definido é o travamento das portas, que deveria acontecer uma vez em que o evento - o pressionar do botão - é executado. Dessa maneira, o sistema realiza a transição do estado inicial - em que o carro se encontra destravado - para o cenário final - no qual as portas estão travadas.

Os dois exemplos ilustram como o uso da linguagem natural - um dos pilares do BDD - torna as especificações mais claras e acessíveis. A combinação da leitura facilitada com o padrão *Given/When/Then* contribui tanto para contextualizar o leitor quanto para possibilitar a visualização objetiva do comportamento esperado do sistema.

A partir dos cenários Gherkin, são gerados testes automatizados, capazes de interagir com o software e verificar seu comportamento. Isso é feito por meio de definições de passos que contêm as instruções necessárias para manipular o sistema e gerar situações de teste.

Com a biblioteca *Behave*, as definições dos passos dos cenários podem ser implementadas por meio de funções em Python. Cada função encapsula a lógica necessária para interagir com o sistema e executar a ação descrita no respectivo passo.

No exemplo da cláusula *When* do último cenário descrito nesta seção, a intenção do passo é representar o pressionar do botão de travamento das portas. Essa ação pode ser realizada por meio da alteração do sinal de entrada do sistema que indica o estado do botão, modificando seu valor de 0 para 1. Utilizando a biblioteca *Behave* em Python, isso pode ser realizado da seguinte maneira:

```
@when('eu pressiono o botão de travamento das portas')
def passo_pressionar_botao_de_travamento(context):
    context.botao_trava = 1
```

Onde o decorador **@when** especifica qual passo está sendo definido, e a função **passo_pressionar_botao_de_travamento** contém a lógica que deve ser executada quando esse passo for invocado. O objeto **context** é amplamente utilizado na biblioteca *Behave* por permitir o compartilhamento de informações entre os passos, bem como o acesso a interfaces do sistema sob teste. Nesse caso, assume-se que o valor do sinal referente ao botão de travamento está armazenado no objeto **context** e pode ser modificado por meio de uma simples atribuição.

Na prática, existem algumas complexidades adicionais a serem tratadas na implementação dessas funções, como a determinação do método para estabelecer comunicação

com o sistema. No entanto, a função apresentada ilustra claramente o princípio do mapeamento entre os passos descritos em linguagem natural e as ações que operam sobre o sistema - cada passo, ao ser executado, invoca uma função responsável por manipular o sistema de forma a aplicar o comportamento definido na especificação.

2.4 DESIGN ORIENTADO POR MODELO

O desenvolvimento de software embarcado impõe uma série de desafios, conforme destacado por Sangiovanni-Vincentelli e Martin (2001). Entre os principais, estão os altos custos associados ao ciclo de desenvolvimento e as severas restrições de performance, consumo de energia e capacidade de processamento - fatores diretamente condicionados pelo hardware utilizado. Essas limitações tornam o processo de definição e validação do sistema particularmente complexo, já que até mesmo alterações simples nos requisitos podem demandar modificações significativas no código-fonte, seguidas por longas etapas de recompilação, testes e depuração.

Nesse cenário, o *Model-Based Design (MBD)* surge como uma abordagem eficaz para lidar com tais dificuldades. Segundo a MathWorks (MathWorks, 2020), ele consiste de uma metodologia que utiliza modelos computacionais e simulações ao longo do processo de desenvolvimento de um sistema, substituindo a escrita manual de código. Através dessa abordagem, os sistemas podem ser projetados, simulados e validados em um ambiente integrado.

Os principais benefícios dessa metodologia são:

- Ligação do design diretamente aos requisitos;
- Colaboração em um ambiente de desenvolvimento compartilhado;
- Simulação de vários cenários possíveis;
- Otimização de performance a nível do sistema;
- Geração automática de código embarcado, documentação e relatórios;
- Detectar erros mais cedo por testar mais cedo.

Neste trabalho o último ponto será especialmente demonstrado, devido a utilização dos testes de aceitação das histórias de usuário. Isso será feito ao integrar cenários *Gherkin* com o modelo em *Simulink* - ferramenta da MathWorks para modelagem e simulação - para executar os testes que validam o comportamento do sistema.

Outra barreira frequentemente encontrada no desenvolvimento de software embarcado é a sua forte dependência do hardware utilizado, especialmente quando se trata

da interação com os diversos componentes conectados ao microcontrolador. No caso do sistema de travamento de portas, por exemplo, é necessária a utilização de um atuador que seja capaz de acionar a trava e impedir a abertura da porta - algo que pode ser implementado com o uso de motores de passo ou servo motores.

A dificuldade surge justamente na definição das entradas e saídas do sistema, que precisam ser modeladas de forma compatível com os componentes selecionados. No entanto, muitas vezes o hardware ainda não está totalmente definido nesta etapa do projeto, ou pode vir a ser alterado futuramente por razões de custo, disponibilidade ou requisitos de desempenho. Essas mudanças forçam modificações no modelo do sistema, o que acaba reproduzindo um dos principais problemas da abordagem tradicional baseada em código manual: a necessidade constante de retrabalho sempre que há mudanças na base de hardware.

Para mitigar esses tipo de problema, é comum a adoção de padrões de desenvolvimento de software, como o *Automotive Open System Architecture* (AUTOSAR) (AUTOSAR, n.d.), que separa o sistema em camadas, cada uma com um nível específico de abstração. Essa arquitetura em camadas permite isolar as dependências de hardware, facilitando a reutilização de código e a manutenção do sistema.

No modelo AUTOSAR, a estrutura do software é decomposta nas seguintes camadas:

- **Camada de Aplicação** - contém componentes de software que são responsáveis pela lógica funcional do sistema, que são completamente independentes do hardware;
- **Software básico** - implementa serviços que permitem o acesso direto ao hardware, lidando com os detalhes técnicos da plataforma física utilizada;
- **RTE (*Run-Time Environment*)** - faz a interface entre as duas camadas anteriores e gerencia a comunicação entre diferentes componentes de software.

Neste caso, o desenvolvimento do modelo tem foco na camada de aplicação, similar à presente no padrão AUTOSAR, definindo interfaces de entrada e saída como abstrações simplificadas que satisfazem o comportamento do sistema sem se prender em detalhes técnicos. Por exemplo, uma possível abstração da saída do sistema, considerando o problema levantado anteriormente - o estado de travamento da porta - pode ser feita ao representar cada trava por meio de um sinal binário: onde 1 indica o estado travado e 0, o destravado.

Essa interface corresponde à saída do modelo, a qual, na prática, é transmitida através da camada de RTE para o software básico. Nele acontece a conversão desse valor binário para um formato compatível com o componente físico selecionado, gerando uma interface física que é transmitida para o atuador - a trava da porta.



Figura 1 – Interfaces binárias e físicas utilizadas no travamento da porta.

Um benefício adicional dessa abstração é que ela permite a utilização direta das interfaces nos cenários *Gherkin* e torna a validação independente dos detalhes técnicos do sistema, assim como é definido em testes de caixa preta (SOMMERVILLE, 2011). Essa metodologia vai de acordo com a definição dos comportamentos no BDD, afinal, o valor gerado na funcionalidade de travamento de portas não depende da capacidade de movimentar um motor, mas sim da lógica que habilita o estado de travamento, independentemente do mecanismo utilizado para isso.

2.5 SISTEMAS DE TRAVAMENTO DE PORTA VEICULARES

Segundo Gsmh (2022), o sistema de travamento de portas veicular engloba todos os componentes responsáveis pelo gerenciamento do acesso ao interior e à saída do veículo. Sua função principal é oferecer ao motorista a possibilidade de limitar a abertura das portas, assegurando o veículo contra intrusões e acessos não autorizados.

Esse sistema constitui um importante recurso de conveniência e segurança, sendo capaz de realizar o travamento central (REIF, 2015), que abrange não apenas as portas laterais, mas também o porta-malas e a tampa de acesso ao tanque de combustível. Além disso, pode incorporar diversas funcionalidades adicionais, como a prevenção da abertura das portas durante o uso do veículo ou o destravamento automático em situações de emergência, como em acidentes.

O mecanismo de travamento é geralmente implementado por meio de atuadores elétricos ou pneumáticos, que mantêm a porta na posição fechada e permitem, conforme a limitação imposta pela trava, sua abertura pelas maçanetas. Parte dos componentes estruturais do sistema é integrada à própria carroceria do veículo, como barras de reforço nos pilares, além dos atuadores responsáveis pelo gerenciamento das trancas.

Para assegurar o acesso seguro ao veículo contra possíveis ataques, a autenticação da chave utiliza protocolos de criptografia (GLOCKER; MANTERE; ELMUSRATI, 2017), que garantem a proteção tanto no registro quanto na validação das chaves. Sua implementação, conforme o padrão AUTOSAR, é realizada em um componente de software básico denominado *Key Manager* (AUTOSAR, 2024), o qual abstrai a metodologia empregada e disponibiliza ao software de aplicação uma interface simplificada para validação da chave.

A leitura da chave é normalmente feita por meio de dispositivos transceivers, responsáveis pela troca de informações com a chave utilizando tecnologias como o *Radio*

Frequency Identification (RFID) (TEAM, 2025). Assim, o registro da chave indentificada é encaminhado ao *Key Manager*, que executa a validação de acordo com o protocolo de criptografia definido.

Alguns fatores técnicos relevantes para a implementação física do sistema de travamento das portas e da validação das chaves serão detalhados na Seção 4, apresentados sob a forma de perguntas levantadas durante o mapeamento de exemplos da Etapa 4.2. Isso se deve à natureza do processo de BDD, que prioriza a definição das funcionalidades a partir da perspectiva do usuário, em vez de se concentrar na tecnologia ou na metodologia empregada. Assim, a elaboração das histórias de usuário independe da definição prévia do mecanismo específico de travamento adotado ou do protocolo de autenticação de chaves a ser implementado.

3 METODOLOGIA

Neste capítulo, será apresentada a metodologia aplicada para o desenvolvimento deste produto do Trabalho de Conclusão de Curso (TCC), que possui natureza técnica e tem como objetivo a adaptação do processo de *Behavior-Driven Development* (BDD) ao contexto da Engenharia Automotiva. A prova de conceito é realizada a partir da aplicação da metodologia na produção de um sistema de travamento de portas veicular.

Esta é uma pesquisa aplicada, voltada à solução de complexidades presentes em metodologias de desenvolvimento de produto tradicionais como a rastreabilidade de requisitos e a execução de testes manuais. Para a avaliação dos resultados, foi aplicada uma abordagem quantitativa, com a finalidade de metrificar a cobertura dos testes de caixa-preta desenvolvidos sobre o modelo do sistema, além de identificar falhas que guiam a sua implementação.

3.1 PROCEDIMENTOS METODOLÓGICOS

Este produto teve seu desenvolvimento com base na metodologia do processo de BDD como demonstrado em Solis e Wang (2011), com uma série de adaptações específicas para o contexto da Engenharia Automotiva, resultando nas seguintes etapas:

1. Descrição das histórias de usuário: Capturar as histórias definidas em forma de requisitos funcionais com linguagem natural e que capture o valor gerado pela perspectiva do usuário;
2. Mapeamento de exemplos: Definição de exemplos concretos tomados da perspectiva do usuário final;
3. Desenho do diagrama de caixa preta: Desenho do diagrama do sistema que demonstra suas interfaces de entrada e saída, sem demonstrar detalhes de suas interações;
4. Desenvolvimento dos cenários Gherkin: Escrita dos cenários aplicando o padrão cucumber e que aborde todas as regras definidas;
5. Modelagem iterativa do sistema em simulink: Modelagem feita de forma iterativa com a criação das definições de passos dos cenários, seguido da sua aplicação como critérios de aceitação;
6. Análise quantitativa do produto final: Validação dos resultados obtidos nas etapas anteriores.

Durante a primeira etapa (4.1), foram elaboradas as histórias de usuário que descrevem as funcionalidades a serem implementadas no sistema, considerando a perspectiva do usuário final. Para essa definição, utilizou-se como referência a bibliografia disponível sobre o funcionamento de sistemas de travamento de portas em veículos, com destaque para os exemplos da Volkswagen (VOLKSWAGEN, 2025) e da Bosch (GSMH, 2022; REIF, 2015).

Concluída essa fase, a segunda etapa (4.2) concentrou-se na aplicação do mapeamento de exemplos, cujo objetivo é refinar o design do sistema ao explicitar a utilização das funcionalidades sob a perspectiva do usuário. Nesse processo, foram definidos os seguintes elementos:

- Exemplos concretos de uso da funcionalidade, descritos pela perspectiva do usuário;
- Regras de comportamento que esclarecem a resposta esperada do sistema diante dos exemplos apresentados;
- Questões consideradas não pertinentes para a definição dos exemplos, mas que são respondidas ao término da elaboração da história.

As perguntas levantadas durante a definição dos exemplos poderiam, em princípio, ser respondidas apenas ao final do processo, uma vez que não impactam diretamente a especificação dos comportamentos do sistema. Contudo, optou-se por tratá-las antes de avançar para as histórias seguintes, a fim de garantir um entendimento mais detalhado do comportamento esperado e permitir a evolução incremental das funcionalidades definidas.

Na terceira etapa (4.3), elaborou-se um diagrama do sistema no formato de caixa-preta, utilizado como suporte para a criação dos cenários em *Gherkin*, definindo as interações com o sistema que devem ser utilizadas durante os passos. Esse diagrama também serve como base para a modelagem no Simulink, especificando as entradas e saídas necessárias para a execução dos testes.

Na quarta etapa (4.4), são definidos os cenários em *Gherkin*, que especificam o comportamento esperado do sistema e funcionam como testes de aceitação das histórias de usuário. Cada exemplo levantado durante o mapeamento deve ser representado em um ou mais cenários, assegurando que todas as interações previstas estejam contempladas nos testes.

A quinta etapa (5.1) consiste na modelagem iterativa do sistema, orientada pela execução dos testes de aceitação definidos nos cenários. Esse processo é realizado por meio da repetição cíclica dos seguintes passos:

1. Executar os cenários de uma das histórias de usuário a partir do arquivo **.feature**;

2. Identificar os erros reportados na execução dos testes;
3. Implementar as correções no modelo, nos passos ou na definição dos passos;
4. Repetir os passos 1 a 3 até que todas as falhas sejam corrigidas e os testes da história sejam aprovados;
5. Aprovar a história, caso todos os seus cenários tenham sido validados com sucesso;
6. Executar novamente os testes das histórias já aprovadas para verificar possíveis falhas introduzidas pela implementação de novos comportamentos;
7. Corrigir eventuais falhas identificadas nas histórias de usuário anteriores;
8. Repetir o ciclo para a história seguinte.

Dessa forma, a execução dos testes ocorre de maneira repetitiva, assegurando que a evolução gradual do sistema não comprometa comportamentos previamente aprovados. Essa verificação contínua amplia a cobertura dos testes, permitindo identificar conflitos entre diferentes histórias de usuário e aumentando a robustez do sistema, dos cenários e das definições de passo.

Ao término da modelagem iterativa, todo o conjunto de cenários de todas as histórias de usuário deve ser executado como uma validação final, garantindo que o comportamento especificado para o sistema seja plenamente atendido.

Por fim, na sexta etapa (5.2), realiza-se uma análise quantitativa dos resultados obtidos durante a aplicação do processo. Nessa fase, são apresentadas instâncias de falhas detectadas graças à maior cobertura de testes proporcionada pelo desenvolvimento iterativo.

3.2 FERRAMENTAS UTILIZADAS

As ferramentas e tecnologias adotadas no desenvolvimento deste produto foram escolhidas com base na compatibilidade com o modelo proposto e na sua capacidade de integrar diferentes etapas do processo. A seguir, descreve-se cada uma delas:

- **Cucumber (Gherkin)**: utilizado como padrão para a escrita de cenários comportamentais, permitindo a especificação dos requisitos no formato Given-When-Then, de forma legível por humanos e máquinas;
- **Miro**: website empregado para a criação de diagramas não técnicos, bem como para auxiliar na discussão colaborativa de exemplos, fluxos e histórias de usuário;
- **Visual Studio Code**: ambiente de desenvolvimento (IDE) utilizado para escrever os cenários em Gherkin, desenvolver a tradução para testes executáveis e integrar o código gerado ao microcontrolador;

- **Python:** linguagem escolhida para implementar a lógica de tradução dos cenários Gherkin em testes executáveis, permitindo automatização e validação do comportamento esperado do sistema.
 - **Behave:** biblioteca utilizada para interpretar e executar os cenários comportamentais no formato BDD, integrando as especificações Gherkin à lógica de teste.
 - **Matlab Engine API for Python:** biblioteca usada para permitir a comunicação entre scripts Python e o ambiente Simulink, possibilitando a execução dos testes durante a simulação.
- **Simulink:** ferramenta adotada para a modelagem do sistema embarcado funcional, possibilitando simulações e geração automática de código C para o sistema-alvo.
- **Git/Github:** utilizado para o versionamento do projeto que inclui o modelo simulink do sistema, código embarcado gerado do modelo, arquivos feature dos cenários gherkin e códigos python dos testes executáveis.

3.3 CRITÉRIOS DE AVALIAÇÃO

Para a verificação e validação do produto gerado, serão aplicados critérios quantitativos de análise, com base nos testes de aceitação de cada história de usuário. Este processo será realizado durante as etapas 5 (5.1) e 6 (5.2), correspondente à modelagem iterativa do sistema no Simulink e à análise dos resultados obtidos.

A cada iteração da modelagem, os testes de aceitação de uma história de usuário serão executados e, com base nos cenários não aprovados, serão identificadas falhas no comportamento do modelo. A partir dessas falhas, serão implementadas soluções para garantir a aprovação dos testes que podem envolver ajustes na lógica do modelo, alterações na escrita dos cenários Gherkin ou atualizações nas funções de definição de passos.

Uma história de usuário é considerada aprovada quando todos os cenários definidos no arquivo **.feature** são satisfeitos, assim como os cenários das histórias já aprovadas anteriormente. Esse procedimento assegura que as modificações implementadas para atender a um cenário específico não impactem os demais, contribuindo para a robustez do sistema.

Ao final do processo, todos os cenários referentes às histórias de usuário foram executados em conjunto, garantindo que o produto final contemplasse todas as funcionalidades mapeadas e que 100% dos testes de aceitação fossem aprovados. Em seguida, foi realizada uma análise global sobre a aplicação do processo, destacando os benefícios da metodologia em comparação com alternativas existentes.

4 IMPLEMENTAÇÃO

Neste capítulo será demonstrado em detalhes o procedimento da aplicação de desenvolvimento de produto como anteriormente apresentado, com 6 etapas que expandem o modelo de Lawrence e Rayner (2019) para adaptá-lo ao produto em questão.

4.1 ETAPA 1: DESCRIÇÃO DAS HISTÓRIAS DE USUÁRIO

A etapa inicial do Desenvolvimento Orientado por Comportamento (BDD) consiste na definição das funcionalidades a serem implementadas por meio de histórias de usuário. Nessa fase, a perspectiva do usuário - ou da pessoa que se beneficia da funcionalidade - deve ser o foco principal. Essa abordagem é um dos pilares fundamentais do BDD, pois garante que o produto desenvolvido entregue valor real ao usuário final.

Portanto, no contexto deste trabalho que trata do sistema de travamento de portas veicular, serão consideradas implementações práticas amplamente adotadas pela indústria automotiva como base para as histórias de usuário criadas.

O travamento e destravamento remoto das portas, acionado por meio da chave com controle remoto ou até mesmo via celular, permite que todas as portas do veículo sejam travadas ou destravadas à distância, sem a necessidade de inserção da chave na fechadura (GSMH, 2022). Esse caso de uso mais simples é vastamente presente em veículos de diversas montadoras e pode ser capturado por meio de duas histórias de usuário:

- Como dono de um veículo automotivo, quero travar todas as portas do meu carro utilizando o botão de travamento, para que eu possa mantê-lo seguro contra furtos e impedir o acesso de pessoas não autorizadas;
- Como dono de um veículo automotivo, quero destravar todas as portas do meu carro utilizando o botão de destravamento, para que eu possa acessá-lo.

Além disso, outra funcionalidade que é também vastamente presente é a indicação da confirmação do travamento ou destravamento das portas, que normalmente utiliza um sinal visual com os faróis do carro, assim como descrito por Volkswagen (2025). A seguinte história de usuário será implementada para capturar essa funcionalidade:

- Como dono de um veículo automotivo, quero receber uma indicação clara sempre que o travamento ou destravamento das portas ocorrer, para que eu tenha certeza do estado de segurança do meu carro.

Em alguns veículos como os da Volkswagen (VOLKSWAGEN, 2025), há uma funcionalidade que permite destravar as portas sem a necessidade de usar a chave, desde que ela esteja próxima ao veículo. O sistema *Keyless Access* possibilita que, ao interagir com as maçanetas das portas, o motorista possa destravar ou travar o carro sem precisar pressionar o botão no controle.

Esse recurso é útil em situações cotidianas, como quando o motorista deseja entrar no carro enquanto a chave está, por exemplo, no bolso. Embora o processo tradicional exija que o usuário retire a chave do bolso, pressione o botão de destravamento e, em seguida, guarde-a novamente, o uso do *Keyless Access* simplifica essa sequência. A diferença é pequena - apenas alguns segundos de tempo -, porém a implementação de uma funcionalidade que elimina etapas desnecessárias torna a experiência mais prática, especialmente considerando que esse mesmo uso será repetido diversas vezes ao longo do dia.

O manual do veículo **Tera** descreve funcionalidades adicionais, como o travamento das portas ou o destravamento total do veículo ao se interagir duas vezes com a maçaneta. No entanto, para fins de simplificação, este trabalho irá considerar apenas o cenário básico de destravamento de uma única porta - especificamente, aquela com a qual o usuário interage. Com base nisso, a seguinte história de usuário é adotada:

- Como dono de um veículo automotivo, quero que a porta destrave automaticamente ao tentar abri-la com a chave em minha posse, para que eu possa acessar o veículo de forma mais rápida e conveniente.

Por fim, será implementada mais uma história de usuário, também baseada em funcionalidades descritas no manual do proprietário. Trata-se de um recurso que realiza o travamento automático do veículo após um determinado período de tempo, ativado em diferentes situações. Neste trabalho, a funcionalidade implementada será especificamente aquela em que o travamento automático é causado para evitar o destravamento não intencional.

Considerando que o usuário deseja manter seu veículo seguro contra furtos, garantir que ele permaneça travado é fundamental. Por esse motivo, identificar uma possível destrava indevida possui valor semelhante ao travamento convencional. A lógica adotada para essa funcionalidade depende de como o sistema reconhece a não intenção do motorista, o que deve ser inferido com base na interação do usuário com o veículo. Isso será definido na sessão do mapeamento de exemplos da história em questão, na próxima etapa.

Essa funcionalidade será capturada pela seguinte história de usuário:

- Como dono de um veículo automotivo, quero que as portas sejam re-travadas automaticamente caso o carro seja destravado sem intenção, para que o veículo permaneça seguro contra acessos não autorizados.

4.2 ETAPA 2: MAPEAMENTO DE EXEMPLOS

As histórias definidas na etapa anterior têm o papel de estabelecer um escopo claro da funcionalidade em discussão, garantindo que toda a equipe compreenda de forma alinhada o incremento de produto que está sendo proposto. Essa clareza é essencial para a etapa de mapeamento de exemplos, na qual ocorrem discussões colaborativas com pessoas de diferentes áreas.

É fundamental que todos os membros da equipe estejam cientes da ideia proposta e do valor que ela gera, pois isso permite um engajamento mais efetivo nas conversas. Em equipes ágeis (Atlassian, n.d.), a colaboração é um dos pilares para assegurar a qualidade do produto em desenvolvimento, especialmente em discussões que envolvem profissionais com diferentes formações e níveis de familiaridade técnica ou de negócio.

Para guiar a discussão, o padrão de mapeamento de exemplos utilizando notas coloridas como descrito por Wynne (2015). Com essa estrutura em mente, todos os pontos desconhecidos previamente identificados são registrados como perguntas, evitando que a discussão se desvie para tópicos fora do escopo naquele momento.

As seguintes perguntas serão adicionadas em notas vermelhas na discussão da primeira história:

- Como a porta é mecanicamente travada?
- Como a porta travada impede a sua abertura?

Após as dúvidas serem anotadas, a discussão segue focada na experiência do usuário, mesmo que alguns aspectos técnicos ainda não estejam totalmente esclarecidos.

4.2.1 História de usuário 1

Para garantir o foco na perspectiva do usuário, a discussão da história será iniciada com a listagem de exemplos concretos. Assim como recomendado por Matt Wynne (WYNNE, 2015), pode-se utilizar uma nomenclatura do exemplo capturado como em um episódio de seriado, que é simplificado no formato de "aquele em que...".

Para a primeira história, alguns exemplos podem ser:

- **Aquele em que o usuário estacionou o carro e travou as portas** - situação típica de uso normal, especialmente quando o veículo está em uma área onde pessoas não autorizadas podem tentar acessá-lo. Nesse caso, a intenção do usuário é evidente: as portas devem ser travadas;

- **Aquele em que o usuário tentou travar o veículo que já estava travado** - para o caso em que, talvez por incerteza, o botão de travamento foi pressionado mais de uma vez, para garantir que o veículo realmente estava seguro;
- **Aquele em que uma pessoa não autorizada tentou abrir a porta do veículo travado usando a maçaneta** - como o usuário do veículo previamente acionou o travamento, presume-se que sua intenção era impedir qualquer acesso não autorizado. Assim, neste caso, a maçaneta não deve permitir a abertura da porta.

Destes exemplos concretos, é possível extrair as seguintes regras acerca do comportamento esperado do produto:

- Ao pressionar o botão de travamento, todas as portas devem ser travadas;
- Qualquer porta travada não deve ser aberta ao utilizar a maçaneta.

Um dos principais benefícios da metodologia de desenvolvimento orientado a comportamento manifesta-se já nesta etapa do processo: os questionamentos gerados durante a discussão. Esses surgem de forma natural ao adotar a perspectiva do usuário em um ambiente colaborativo, no qual o foco está no mapeamento de exemplos concretos.

Por exemplo, ao definir a primeira regra - “ao pressionar o botão de travamento, todas as portas devem ser travadas” - surge uma pergunta instintiva: “E se o veículo já estiver travado quando o botão for pressionado?”. Essas dúvidas enriquecem a especificação e ajudam a prever cenários reais de uso.

Neste exemplo, a história de usuário fornece uma indicação clara da intenção do usuário, expressa pelo valor que se busca alcançar. O objetivo é travar todas as portas, protegendo o veículo contra furtos e impedindo o acesso de pessoas não autorizadas. Diante disso, independentemente do estado inicial do veículo, o comportamento esperado do sistema é o de garantir o travamento completo de todas as portas.

Outras perguntas podem também surgir a partir de variações no questionamento original que foi levantado. Por exemplo: seria possível que, em determinado cenário, o veículo estivesse com apenas algumas portas travadas? Poderia haver um estado intermediário, em que nem todas as portas estejam travadas ou destravadas?

Como será detalhado em uma história de usuário futura, tais situações são de fato possíveis - por exemplo, quando o sistema permite o destravamento de portas específicas com o *Keyless Access*. Ainda assim, a resposta anterior permanece válida: ao acionar o botão de travamento, o sistema deve garantir que todas as portas estejam devidamente travadas.

A segunda regra trata da abertura da porta e estabelece que ela não deve ser permitida quando estiver travada. Esse comportamento é fundamental para a existência de um sistema de travamento, pois assegura que o acesso ao veículo seja restrito a pessoas autorizadas.

Embora existam incertezas técnicas quanto à forma de implementação - seja ela mecânica ou eletrônica -, define-se neste momento que, independentemente da abordagem adotada, uma porta travada não deve abrir quando a maçaneta for acionada. Como nos casos anteriores, as dúvidas ainda existentes serão registradas como perguntas adicionais para orientar futuras discussões da equipe.

Ao fim da definição da história, foram capturadas 2 regras e 2 perguntas relativas a essa história de usuário. Está é a representação final das notas coloridas:

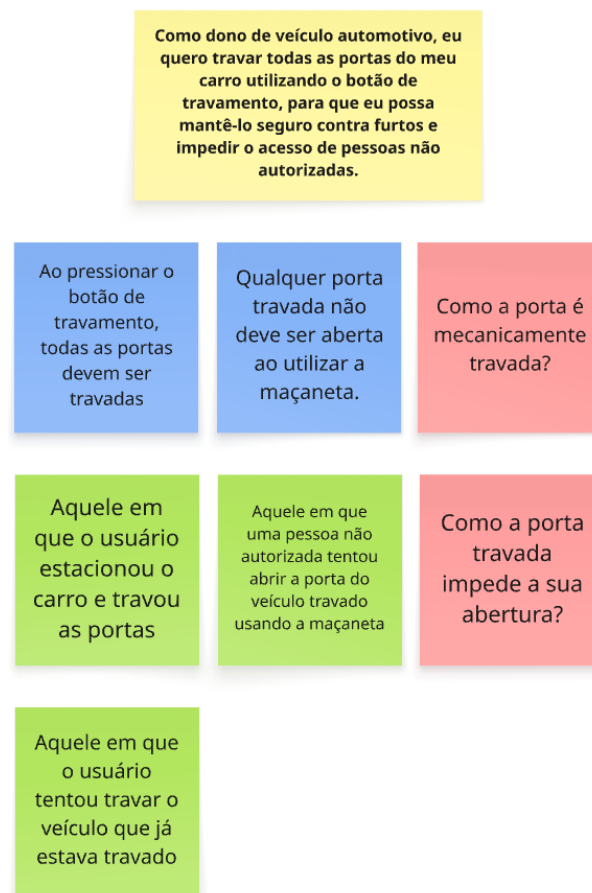


Figura 2 – História de Usuário 1: Travamento de todas as portas.

Antes de seguir para a próxima história de usuário, aqui serão tratadas as perguntas que foram anotadas nesta seção. Esta etapa poderia ser postergada para o final do processo de discussão de exemplos, mas para garantir uma evolução incremental do produto, a próxima história será tratada quando os pontos abertos da atual forem respondidos.

Assim como descrito por Reif (2015), sabe-se que existem 2 principais tipos de

sistemas de travamento de porta que podem ser encontrados em veículos modernos:

- Trava mecânica que é operada de maneira elétrica ou pneumática;
- Trava eletrônica que é montada com a trava e os eletrônicos embutidos.

Neste projeto, será implementado uma versão simulada de um sistema de trava eletrônica para se aproveitar de sua simplificação e menor número de componentes mecânicos no produto físico. Tipicamente, nesse sistema as maçanetas não precisam se mover ou podem até serem removidas inteiramente, podendo ser substituídas por botões equipados com sensores que se comunicam com a trava. Além disso, todos os componentes que transmitem o movimento da maçaneta no sistema mecânico aqui são desnecessários, substituídos pela fiação que se liga ao componente.

Embutido na trava eletrônica também é incluído o mecanismo básico de travamento como descrito por Reif (2015). Ele é composto por três peças principais: o trinco (1) e engate (2) - que são acoplados à porta - e o pino de batente (3) - que é acoplado ao chassi do veículo. Eles são responsáveis por manter a porta firmemente fechada, conforme ilustrado na figura 2:

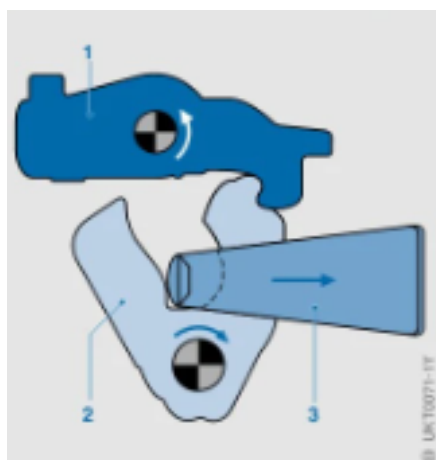


Figura 3 – Mecanismo de travamento da porta. Fonte: (REIF, 2015).

Durante o fechamento da porta, o engate (2) colide com o pino de batente (3) e gira no sentido anti-horário, passando por baixo do pino até retornar à sua posição inicial e fazer contato com o trinco (1). Nesse ponto, a trava encontra-se na posição fechada, exatamente como demonstrado na imagem, e qualquer tentativa de abrir a porta força o engate no sentido horário - movimento que é impedido pela presença do trinco.

Quando a porta está destravada, no entanto, é possível abri-la ao girar o trinco no sentido anti-horário. Esse movimento libera o engate, permitindo que ele se afaste do pino de batente e possibilite a abertura da porta.

No caso da trava eletrônica, o acionamento desse mecanismo é controlado por lógica implementada em software e transmitido por meio de um motor que é mecanicamente conectado ao trinco. Assim, respondendo à segunda pergunta, a porta trava se mantém fechada ao bloquear o movimento do trinco mediante o uso da maçaneta, da mesma forma que a abertura é feita ao mover o trinco.

Para modelar esse comportamento do sistema, o seu escopo pode ser definido em termos de lógica de aplicação e de software básico da seguinte maneira:

- **Lógica de aplicação** - gera saídas em binário para cada porta, definindo se ela está travada ou destravada e se ela deve ser aberta ou mantida fechada;
- **Lógica de software básico** - recebe o sinal de saída em 0 ou 1 e o converte em sinais compreensíveis para o hardware de travamento, que resultam no movimento do motor.

Além do mecanismo de travamento em si, serão utilizados botões para realizar o travamento e abertura das portas. Seu escopo será feito de maneira similar, utilizando valores binários como abstrações dos sinais existentes a nível de hardware que são convertidos no software básico. Em suma, para atender a primeira história de usuário discutida, o modelo do sistema será desenvolvido tomando como base a lógica de aplicação assim como destacado na figura 3:

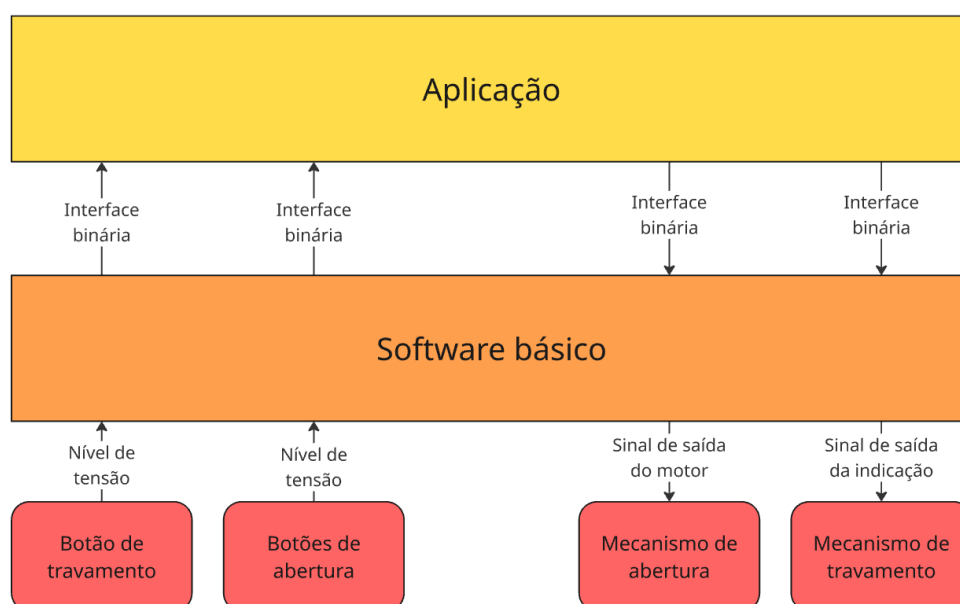


Figura 4 – Diagrama de abstração do sistema em camadas de aplicação, software básico e componentes físicos.

Portanto, para a lógica de aplicação do modelo, receber o valor 1 recebido pela interface binária do botão de travamento é interpretado como um comando de pressionamento, da mesma forma que o valor 1 na interface do botão de abertura indica que a

maçaneta foi acionada. No sentido oposto, ao enviar a saída para o software básico, esses valores são interpretados e convertidos em sinais de controle que determinam o movimento do motor.

Na prática, o travamento ou destravamento funciona apenas como uma limitação à abertura das portas e portanto, especialmente no caso da trava eletrônica, não há distinção física evidente entre os dois estados. Para facilitar essa identificação, este trabalho introduz uma indicação adicional, representada pelo “Mecanismo de travamento” na Figura 4. Ela servirá apenas para indicar o estado atual de cada porta, facilitando a execução dos testes e a validação funcional do sistema.

4.2.2 História de usuário 2

Ao considerar a história de usuário de destravamento das portas, é importante pensar na perspectiva da intenção de acessar o veículo. Isso é uma situação cotidiana que acontece com altíssima frequência, e portanto possui exemplos muito semelhantes a história 1:

- **Aquele em que o usuário destravou as portas e entrou no carro** - situação típica de uso, ocorre sempre que o usuário retorna ao carro e precisa acessá-lo normalmente após o destravamento;
- **Aquele em que algumas portas já estavam destravadas** - caso em que o veículo se encontrava parcial ou totalmente destravado no momento em que o botão de destravamento foi acionado;
- **Aquele em que o usuário abriu a porta que estava destravada** - situação em que, após o destravamento, o usuário acessa o veículo por meio da maçaneta, uma vez que a porta está destravada.

Para satisfazer os exemplos listados, as seguintes regras serão implementadas:

- Ao pressionar o botão de destravamento, todas as portas devem ser destravadas.
- Qualquer porta destravada deve ser aberta ao utilizar a maçaneta.

Ao final, foram capturados 3 exemplos e 2 regras na história de usuário, o que compõe as seguintes notas coloridas:

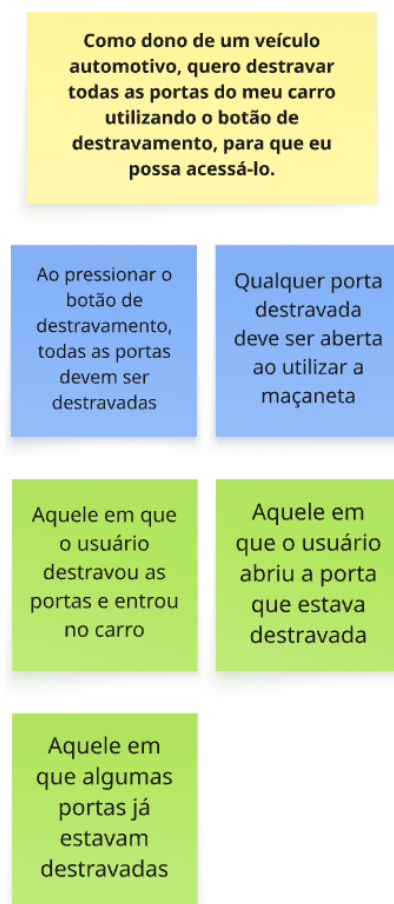


Figura 5 – História de Usuário 2: Destravamento de todas as portas.

Com base nas respostas obtidas na história anterior, diversas incertezas acerca do funcionamento do destravamento das portas já foram determinadas. Sabe-se, por exemplo, que a porta destravada possui uma indicação de estado e que, ao pressionar o botão de abertura, a saída do modelo deve informar ao software básico que o mecanismo da porta deve ser liberado - ação que será realizada por meio do sinal de controle enviado ao motor.

A única distinção nesta história é a introdução de um novo botão: o botão de destravamento. Ele exige uma interface semelhante à descrita anteriormente, sendo igualmente interpretada pelo software básico como um valor de tensão e convertido em um sinal binário que indica o estado atual do botão.

4.2.3 História de usuário 3

A indicação da confirmação de travamento ou destravamento do veículo tem a grande importância de permitir que o usuário esteja assegurado de que o seu comando foi recebido e executado pelo veículo. Ela deve portanto ser invocada sempre que o usuário estiver utilizando as histórias 1 e 2, além de capturar os seguintes exemplos:

- Aquele em que o usuário travou seu veículo e ficou na dúvida se as portas

foram travadas ou destravadas - caso em que o usuário utiliza o botão de travamento e espera que o seu veículo seja assegurado. Deve invocar uma indicação distinta da indicação de destravamento;

- **Aquele em que o usuário destravou seu veículo e ficou na dúvida se as portas foram travadas ou destravadas** - caso em que o usuário utiliza o botão de destravamento e espera que o seu veículo seja destravado. Deve invocar uma indicação distinta da indicação de travamento;
- **Aquele em que o usuário tentou travar seu veículo sem perceber que uma das portas ficou aberta** - pode acontecer quando uma ou mais portas ficam abertas por conta de alguém esquecer de fechá-la. Neste caso, mesmo travando a porta que está aberta, o veículo ainda não fica seguro contra o acesso indevido e portanto deve invocar uma indicação de falha.

O último exemplo destacado surgiu a partir de um questionamento levantado durante a idealização da história de usuário em análise. O objetivo é garantir que o usuário seja informado em situações nas quais, mesmo tendo solicitado o travamento das portas, o veículo ainda não possa ser devidamente assegurado devido à permanência de uma porta aberta.

Esse comportamento exige a emissão de um aviso distinto ao usuário, por meio de uma terceira forma de feedback que sinalize a condição de porta aberta. Dessa maneira, assegura-se que o valor identificado seja atendido, especialmente considerando que, conforme definido na primeira história, o acionamento do botão de travamento expressa a intenção do usuário de proteger o veículo.

As regras que cobrem os exemplos listados são:

- O veículo deve indicar que foi travado ou destravado com sucesso utilizando sinais distintos para cada comando;
- O veículo deve indicar com um sinal de falha quando pelo menos uma das portas está aberta ao receber o comando de travamento.

O último exemplo levanta questões que precisam ser exploradas por meio de perguntas e trazem consigo um novo fator que o sistema deve considerar: o estado de abertura das portas. Para isso, as seguintes perguntas são criadas:

- Como o sistema detecta se uma porta está aberta ou fechada?
- É possível travar ou destravar uma porta que está aberta?

Dessa forma, essa história de usuário foi mapeada com 3 exemplos, 2 regras e 2 perguntas, de acordo com a Figura 6:

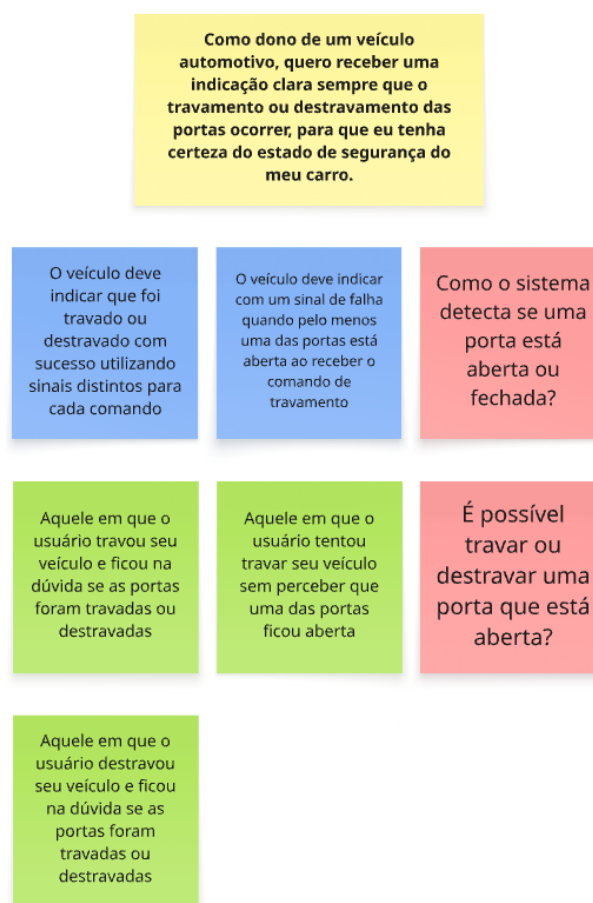


Figura 6 – História de Usuário 3: Feedback de travamento.

A investigação das perguntas levantadas envolvem detalhes que dependem diretamente da implementação física do produto. Para respondê-las, é necessário compreender como um sistema veicular real consegue identificar se as portas estão fisicamente abertas.

Esta detecção é normalmente feita por meio de sensores ou mecanismos capazes de indicar quando a porta está entreaberta (estado *ajar*). Um exemplo é apresentado em Tershak e Thieneman (1986), que descreve o uso de switches físicos: eles são pressionados quando a porta está completamente fechada e liberados quando a porta se abre.

Esse tipo de sensor pode ser integrado de forma semelhante às interfaces binárias utilizadas nos botões de travamento e destravamento, empregando sinais binários (0 ou 1) para representar os estados “fechada” ou “aberta” da porta. Com essa informação, o sistema pode combinar o estado físico da porta com a entrada da interface do botão de travamento para determinar quando o alerta para o usuário deve ser realizado.

É importante destacar que, embora o controle de abertura da porta seja gerenciado pelo sistema, a utilização de um sensor continua sendo essencial para determinar seu

estado real. Esse cenário é comum no controle de componentes mecânicos, em que parte do comportamento não é governada diretamente pela lógica de software ou eletrônica.

No caso em questão, a saída final do sistema de travamento consiste no acionamento do motor que movimenta o trinco e libera o mecanismo da trava, permitindo que a porta seja aberta. No entanto, uma vez que essa ação é executada, o sistema não tem como saber se o usuário realmente abriu a porta - ou se, após abri-la, voltou a fechá-la - sem o auxílio de um sensor. Isso ocorre porque a abertura da porta é uma ação mecânica realizada pelo usuário, sendo apenas condicionada, e não diretamente executada, pelo sistema eletrônico em desenvolvimento.

Além disso, o travamento e destravamento das portas é realizado de maneira independente do estado da porta e pode ser realizado mesmo que ela esteja aberta. Isso deve-se ao fato de que o controlador eletrônico é responsável por definir quando o mecanismo da trava deve ser acionado. Caso a porta esteja destravada, esta ação é tomada após o pressionar do botão de abertura da porta.

Em termos práticos, isso significa que não existe diferença física entre uma porta travada ou destravada, este é apenas um estado que é virtualmente definido pelo controlador. É importante ressaltar que travar uma porta aberta ainda não satisfaz a condição de assegurar o veículo, como definido na primeira história de usuário e por conta disso o alerta do usuário se vê necessário.

4.2.4 História de usuário 4

A funcionalidade de *Keyless Access* permite o destravamento individual das portas assim que o usuário aciona a maçaneta de uma porta travada, desde que a chave autorizada esteja presente em sua posse. Os exemplos de uso dessa funcionalidade são bastante semelhantes aos do destravamento convencional, pois refletem situações em que o usuário tem a intenção de acessar o veículo.

Neste estágio, ainda não foi definida a tecnologia que será utilizada para a identificação da chave. Por isso, essa incerteza será tratada por meio de uma pergunta associada a esta história de usuário.

Os exemplos capturados na discussão da história são:

- **Aquele em que o usuário tinha a chave no bolso e usou a maçaneta** - Este é o caso típico de uso da funcionalidade: a chave autorizada está próxima e é reconhecida pelo sistema, permitindo o destravamento imediato da porta acionada;
- **Aquele em que outra pessoa abriu outra porta simultaneamente** - Neste caso, mais uma porta é aberta simultaneamente, possivelmente por alguém acompanhando

o dono do veículo. O sistema deve tratar as portas de forma independente, e portanto ambas devem ser destravadas individualmente, permitindo o acesso simultâneo;

- **Aquele em que o usuário destravou a porta e a fechou de novo** - Após acessar o veículo utilizando o *Keyless Access* e em seguida fechar a porta, o sistema deve manter a porta destravada até que o usuário execute explicitamente o comando de travamento. Isso garante que a intenção do usuário seja respeitada e evita travamentos não intencionais.

Para cobrir os exemplos, as seguintes regras serão utilizadas:

- Qualquer porta que tiver sua maçaneta utilizada deve ser destravada desde que a chave esteja próxima ao veículo;
- A porta deve ser mantida destravada após a operação de *Keyless Access*.

O mapeamento de exemplos dessa história na forma de notas coloridas é composto por 3 exemplos, 2 regras e 1 pergunta:

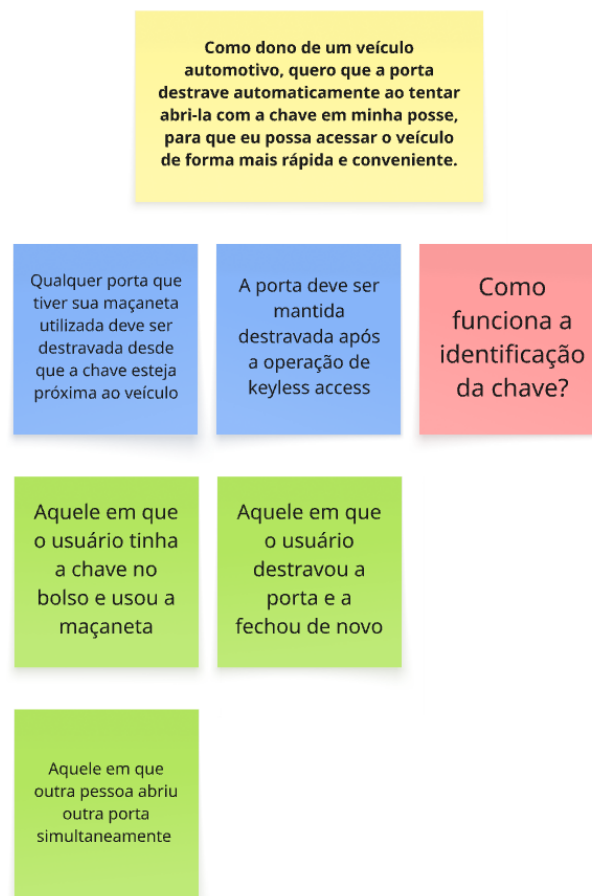


Figura 7 – História de Usuário 4: *Keyless Access*.

Por fim, uma nova pergunta foi levantada no mapeamento de exemplos da história e deve ser respondida antes do trabalho na próxima história. Para fazê-lo, é preciso investigar alguma opção de tecnologia que possa ser utilizada para representar a validação de chave de usuário.

A implementação de sistemas de detecção de chave, conforme descrito por Glocker, Mantere e Elmusrati (2017), costumam utilizar módulos de comunicação por rádio-frequência (RF), capazes de estabelecer conexão com um dispositivo autorizado. A validação desse dispositivo pode ser realizada por meio de diferentes métodos e protocolos de segurança, que incluem mecanismos de autenticação mútua entre a chave e o veículo. Esses protocolos aumentam a robustez do sistema contra tentativas de acesso não autorizado.

Em casos mais simples, como demonstrado em Team (2025), o sistema utiliza um sensor *RFID* (*Radio Frequency Identification*) e uma tag que contém uma chave de identificação única. O funcionamento baseia-se na transmissão do identificador por meio de um sinal de rádio quando a tag é aproximada do sensor. O sensor, por sua vez, envia o código recebido ao microcontrolador, que o compara com um identificador previamente armazenado em sua memória. Caso haja correspondência, a chave é considerada válida e o acesso é autorizado.

Esse tipo de autenticação, no contexto da arquitetura *AUTOSAR*, é normalmente responsabilidade de um componente específico do software básico pertencente à pilha de cibersegurança. Esse componente, conhecido como *Key Manager* (AUTOSAR, 2024), fornece serviços utilizados para a validação de chaves, utilizando diferentes protocolos de segurança. Ele também se comunica com a unidade de memória do controlador para acessar os dados das chaves previamente registradas.

Considerando a presença desse componente no sistema *AUTOSAR*, será adotada novamente uma abstração da tecnologia de hardware envolvida, por meio de uma interface binária. Nesse cenário, a verificação da chave de identificação em relação ao valor armazenado em memória é realizada pelo *Key Manager* e o resultado dessa validação é então comunicado à aplicação de forma semelhante ao funcionamento dos botões de entrada: por meio de um sinal binário que indica se a chave foi validada (1) ou não (0).

4.2.5 História de usuário 5

A funcionalidade de auto travamento é comumente implementada para contemplar diferentes cenários de uso do veículo. Neste caso específico, o foco está na situação em que as portas são destravadas de forma não intencional. Antes de apresentar os exemplos, será feita uma definição do que se entende por “não intenção” por parte do usuário, considerando que, na prática, essa intenção precisa ser inferida com base nas entradas disponíveis no sistema.

Esse tipo de questão costuma ser capturado em perguntas, nas notas vermelhas, para que possa ser investigado após o mapeamento de exemplos. No entanto, para essa história é essencial que a definição de não intencional seja feita, pois ela definirá com clareza que exemplos de uso devem ser mapeados.

De modo geral, o destravamento das portas tem como objetivo permitir o acesso ao veículo, o que se concretiza quando o usuário abre pelo menos uma das portas. Utilizando esse critério, a ausência da abertura das portas dentro de um determinado intervalo de tempo após o destravamento pode ser interpretada como um indicativo de que o usuário não teve a intenção consciente de destravar o veículo.

Essa condição também é presente em veículos Volkswagen, assim como descrito no manual do usuário (VOLKSWAGEN, 2025). Segundo ele, o veículo realiza o travamento automático em até 45 segundos após ter sido destravado, desde que, entre outras possíveis condições, nenhuma das portas tenha sido aberta. A fim de tornar a ação de travamento automático mais rápida e facilitar a execução dos testes, o tempo de espera para essa condição será reduzido para 15 segundos neste trabalho.

Adicionalmente, o estado inicial das portas no momento do destravamento também é de relevância para esse comportamento. Nos casos em que o veículo não estava completamente travado previamente, infere-se que o usuário não possuía a intenção de garantir a segurança do veículo, mesmo antes do botão de destravamento ser pressionado, e portanto ele não precisa ser travado novamente.

Sabe-se que o principal objetivo do travamento do veículo é garantir sua segurança contra furtos e impedir o acesso de pessoas não autorizadas. No entanto, existem cenários em que o usuário pode intencionalmente optar por deixar o veículo destravado, mesmo que não pretenda acessá-lo de imediato. Um exemplo disso seria quando o veículo é deixado em uma garagem particular, onde o usuário deseja mantê-lo acessível para buscar um item posteriormente ou por qualquer outro motivo, sem a necessidade de utilizar novamente a chave ou o sistema de *Keyless Access*.

Para contemplar essa situação e permitir que a intenção do usuário seja respeitada, é necessário que exista pelo menos uma regra específica que permita o destravamento completo, sem que seja necessário o acesso ao veículo. Neste caso, será adotada a estratégia de considerar o duplo acionamento do botão de destravamento como uma indicação explícita da intenção de manter o veículo destravado.

Por fim, os exemplos capturados na história são:

- **Aquele em que o usuário destravou mas não foi acessar o carro** - este é o cenário padrão de destravamento não intencional, que pode ocorrer por diversos motivos, como o acionamento acidental do botão de destravamento. Nessa situação, o

sistema realizará o travamento automático após um intervalo de tempo pré-definido, interpretando que não houve intenção de acesso;

- **Aquele em que o usuário destravou e foi acessar o carro** - neste cenário, a situação oposta à do exemplo anterior acontece, pois o usuário acessou o veículo, abrindo uma das portas, em tempo. O travamento automático não deve ser acionado neste caso;
- **Aquele em que o usuário pressionou o botão de destravamento duas vezes para manter o carro destravado** - Quando for da intenção do usuário deixar o veículo destravado sem abrir nenhuma porta, a função de travamento automático será inibida mediante o duplo acionamento do botão de destravamento. Essa ação será interpretada como uma indicação explícita da intenção de manter o veículo acessível;
- **Aquele em que o usuário destravou o veículo quando ele já estava acessível** - refere-se às situações em que o veículo não estava completamente travado inicialmente, seja porque uma ou mais portas já estavam destravadas ou por conta do acionamento repetido do botão de destravamento. Nestes casos, o sistema entende que o usuário já interagiu com o veículo e portanto o travamento automático também não será executado.

Com os exemplos mapeados, o comportamento definido garante que o travamento automático trata em específico os casos em que a ação do usuário indicou que ele não possuía a intenção de mudar o veículo de um estado inicial em que ele está absolutamente seguro contra furtos - todas as portas estavam travadas - para o caso em que qualquer pessoa poderia acessá-lo. As regras que satisfazem os exemplos são:

- Travar automaticamente caso o botão de destravamento seja usado uma vez e nenhuma porta seja aberta;
- Não travar caso o botão de destravamento seja pressionado uma segunda vez;
- Não travar caso o veículo não esteja completamente travado antes do destravamento.

O mapeamento de exemplos dessa história na forma de notas coloridas é composto por 4 exemplos e 3 regras:

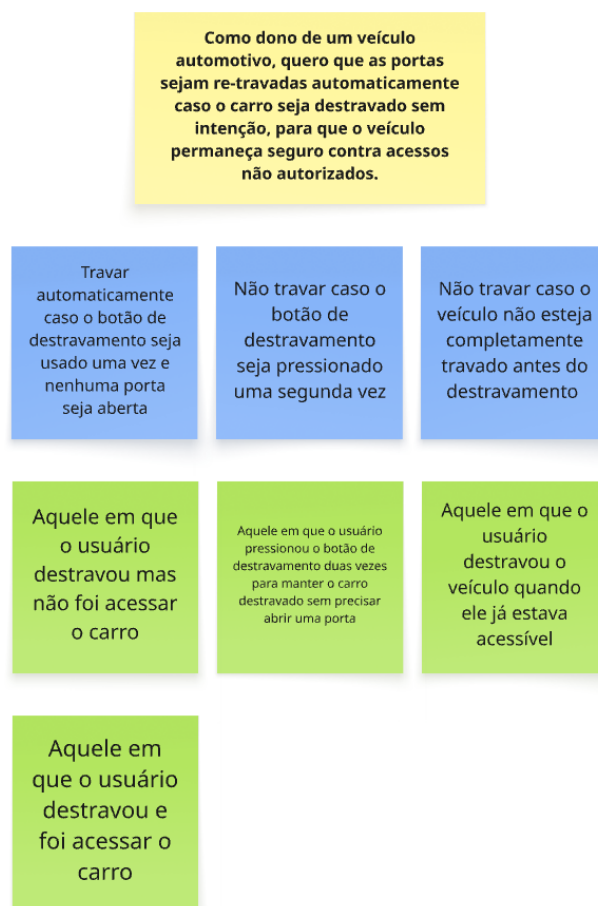


Figura 8 – História de Usuário 5: Travamento automático.

4.3 ETAPA 3: DESENHO DO DIAGRAMA DE CAIXA PRETA

Com todas as histórias de usuário definidas, o próximo passo do trabalho consiste na elaboração do diagrama de caixa preta, com o objetivo de identificar todas as interfaces necessárias para suportar os comportamentos previamente levantados. Essa etapa é essencial para garantir que todas as interações entre o usuário e o sistema estejam claramente especificadas e para suportar a criação dos cenários *Gherkin* a seguir.

Para dar início ao processo, é necessário listar todas as entradas e saídas do sistema:

Entradas:

- Botão de travamento
- Botão de destravamento
- Botões de abertura das portas (4 no total)
- Sensores de abertura das portas (4 no total)
- Detecção de chave autenticada.

Saídas:

- Indicação de estado de travamento de todas as portas (4 no total)
- Estado da tranca de cada porta (4 no total)
- Feedback para o usuário.

O diagrama é então elaborado com um bloco central representando o sistema, ao qual estão conectadas todas as interfaces previamente listadas. Neste estágio, o bloco central permanece sem conteúdo interno, uma vez que o objetivo é abstrair os detalhes da implementação e focar exclusivamente na definição das interações entre o sistema e o usuário.

Dessa forma, o diagrama assume a configuração ilustrada na Figura 9:

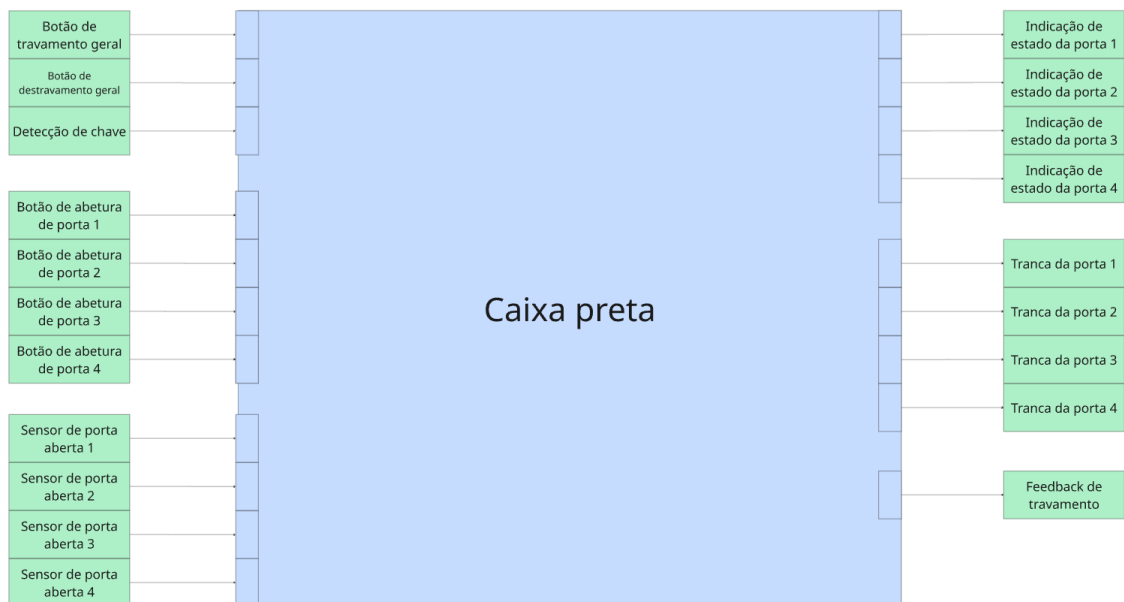


Figura 9 – Diagrama de Caixa Preta do sistema de travamento de portas.

De maneira análoga, o modelo desenvolvido no Simulink também deverá adotar uma estrutura semelhante, composta por um modelo de alto nível - que define apenas as entradas e saídas do sistema - e um modelo de baixo nível, responsável por conter toda a lógica funcional implementada.

É importante destacar que algumas interfaces adicionais serão necessárias no modelo do sistema, embora não estejam representadas no diagrama de caixa preta, pois não estão diretamente relacionadas aos comportamentos observáveis pelo usuário. Essas interfaces serão detalhadas no Capítulo 5, e são fundamentais para suportar aspectos técnicos que garantem o funcionamento correto do sistema. Um exemplo dessas interfaces é o tempo de

simulação, necessário para validar a condição de auto travamento descrita na história de usuário 5.

Para finalizar essa etapa, é necessário definir quais são os valores possíveis que devem ser capturados em cada uma das interfaces criadas no diagrama da Figura 9.

Quadro 1 – Valores possíveis para as interfaces de entrada e saída

Nome da interface 1	Valores Possíveis
Botão de travamento geral	<ul style="list-style-type: none"> • Pressionado • Solto
Botão de destravamento geral	<ul style="list-style-type: none"> • Pressionado • Solto
Detecção de chave	<ul style="list-style-type: none"> • Presente • Não presente
Botão de abertura de porta (de 1 a 4)	<ul style="list-style-type: none"> • Pressionado • Solto
Sensor de porta aberta (de 1 a 4)	<ul style="list-style-type: none"> • Fechada • Aberta
Indicação de estado da porta (de 1 a 4)	<ul style="list-style-type: none"> • Travada • Destravada
Tranca da porta 1	<ul style="list-style-type: none"> • Segura • Solta
Feedback de travamento	<ul style="list-style-type: none"> • Sem status • Confirmação de trava • Confirmação de destrava • Operação falha

Neste quadro, os valores das interfaces foram descritos de maneira textual, com valores que se aplicam à linguagem natural proposta pela metodologia de BDD. Na prática, todos os valores serão utilizados como enumerações convertidas para um valor numérico que representa o estado da interface.

4.4 ETAPA 4: DESENVOLVIMENTO DOS CENÁRIOS GHERKIN

Nesta etapa, os cenários *Gherkin* serão desenvolvidos e utilizados como critérios de aceitação das histórias previamente definidas. Estes cenários devem ser capazes de cobrir todos os exemplos capturados durante o mapeamento, assegurando que o desenvolvimento do sistema atenda à experiência de usuário desejada. Como primeiro passo, será definida a estrutura de pastas do projeto, responsável por organizar tanto os arquivos `.feature` - um para cada história de usuário - quanto o modelo em Simulink. Seguindo a estrutura recomendada na documentação da biblioteca Behave (Behave, 2025), o projeto será organizado da seguinte forma hierárquica:

```
projeto/
|-- features/
|   |-- lock\_all.feature
|   |-- unlock\_all.feature
|   |-- locking\_feedback.feature
|   |-- unlock\_individual.feature
|   |-- auto\_relock.feature
|   `-- steps/
|       `-- lock\_unlock.py
```

Cada história de usuário possui um arquivo `feature` correspondente, todos organizados dentro da pasta `features` do projeto. A nomenclatura desses arquivos foi definida de forma a evidenciar a funcionalidade representada em cada um, facilitando a identificação e o rastreamento das histórias implementadas. Por exemplo, o arquivo `lock_all.feature` refere-se à primeira história de usuário, que descreve a funcionalidade de travamento de todas as portas por meio do botão de travamento.

Além disso, o arquivo `environment.py` é utilizado para definir funções de inicialização que são executadas antes do início dos testes ou de cada passo individual. Uma de suas responsabilidades é iniciar a simulação quando os testes são acionados, além de disponibilizar interfaces que permitem a comunicação com o modelo Simulink.

Por fim, o arquivo de definição dos passos (*step definitions*) é armazenado na pasta `steps` e foi nomeado de `lock_unlock.py`. Ele é responsável por mapear os passos descritos nos cenários *Gherkin* para funções executáveis, que realizam as ações esperadas e interagem com o modelo.

É importante destacar que o desenvolvimento dos cenários *Gherkin* é um processo iterativo, que deve evoluir de acordo com o grau de maturidade do sistema. Por esse motivo, como será demonstrado na etapa de modelagem, ajustes pontuais nos cenários -

como a adição de passos que facilitem a execução da simulação - poderão ser realizados ao longo do tempo.

Durante a criação dos cenários, alguns cuidados serão adotados para garantir consistência e reutilização. Um desses cuidados é a definição de passos genéricos e reutilizáveis, especialmente para ações recorrentes. Por exemplo, nas histórias 1 e 5, que descrevem funcionalidades que resultam no travamento completo do veículo, é possível validar o estado final utilizando um mesmo passo compartilhado, como **Then todas as portas deveriam estar travadas**.

Uma possível definição para esse passo consiste em realizar a leitura dos valores presentes nas quatro interfaces de saída do modelo, que indicam o estado atual de cada porta. Considerando que o valor 1 representa uma porta travada, a função deve verificar se todas as saídas possuem esse valor, aprovando o teste apenas se essa condição for satisfeita.

Para torná-la ainda mais reutilizável, essa função pode incorporar um parâmetro textual extraído diretamente do passo *Gherkin*, permitindo que parte da frase do cenário seja utilizada como argumento. Dessa forma, a lógica de verificação pode ser adaptada dinamicamente: por exemplo, a função pode identificar se a última palavra do passo é “travadas” ou “destravadas” e, com base nisso, validar se os valores lidos são 1 (para travadas) ou 0 (para destravadas), aprovando o teste conforme o caso.

Outra funcionalidade importante empregada na escrita dos cenários é o uso de tabelas de exemplos (*Examples*), que possibilitam a execução repetida de um mesmo cenário com diferentes combinações de parâmetros. Essa abordagem é especialmente útil para validar comportamentos sob múltiplas condições iniciais, evitando a necessidade de duplicar cenários com estruturas idênticas.

Por exemplo, a primeira história de usuário define que, ao pressionar o botão de travamento, todas as portas devem ser travadas, independentemente de seu estado inicial. Para verificar esse comportamento, é possível parametrizar o estado de cada porta utilizando a cláusula *Given* combinada com a tabela de exemplos, como mostrado abaixo:

```
Given a porta 1 está <estado1>
```

```
Examples:
```

```
| estado1 |
| travada |
| destravada |
```

Ao utilizar a sintaxe `<...>`, o Behave interpreta o conteúdo como uma variável, cujo valor será substituído com base nas linhas da tabela *Examples*. Cada linha da tabela

gera um cenário independente, preservando a estrutura original, mas substituindo os parâmetros pelos respectivos valores.

Dessa forma, durante a execução, o Behave processa o cenário duas vezes. Na primeira execução, o passo é `Given a porta 1 está travada` e na segunda ele é `Given a porta 1 está destravada`.

Essa estrutura oferece grande flexibilidade na criação dos cenários e assegura a cobertura completa das possíveis combinações. No cenário real que será apresentado a seguir, existem 16 combinações possíveis, resultantes dos diferentes estados (travada ou destravada) das quatro portas.

Por fim, a funcionalidade *Background* também será utilizada nos arquivos `.feature` para definir um estado inicial comum a todos os cenários. Essa estrutura permite declarar uma sequência de passos `Given` que serão executados automaticamente antes de cada cenário presente no arquivo. Isso evita a repetição de passos que se aplicam a múltiplos cenários, promovendo maior clareza e reutilização na especificação dos testes.

4.4.1 História de usuário 1

Uma maneira eficaz de estruturar o cenário *Gherkin* é adotar a perspectiva do usuário e tentar descrever o comportamento esperado como uma sequência de ações, utilizando as cláusulas `Given`, `When` e `Then`. Tomando como base o primeiro exemplo da história - “aquele em que o usuário estacionou o carro e travou as portas” - o cenário pode ser representado da seguinte forma:

```
Given o carro estava destravado ao ser estacionado
When o usuário pressionou o botão de travamento
Then todas as portas foram travadas
```

Além disso, é possível aplicar o recurso de tabelas de exemplos (*Examples*) para generalizar o cenário e incluir o segundo exemplo descrito na história: “aquele em que o usuário tentou travar o veículo que já estava travado”. Nesse caso, substitui-se a primeira condição (*Given*) por um conjunto de passos que definem individualmente o estado inicial de cada porta, utilizando os valores fornecidos pela tabela para simular todas as combinações possíveis de estados iniciais. A resposta final do sistema é comum entre os exemplos já que ambos compartilham o mesmo resultado final - a mesma cláusula *Then*.

Dessa forma, o primeiro exemplo é coberto em uma das linhas da tabela - a que define que todas as portas estão inicialmente destravadas - e o segundo é coberto por todas as demais linhas - que definem todas as possíveis combinações de estado para cada porta. O cenário geral pode assumir a seguinte forma:

Scenario Outline: Locking all doors

Given the door '1' is <door_1_state>
 And the door '2' is <door_2_state>
 And the door '3' is <door_3_state>
 And the door '4' is <door_4_state>

When I press the vehicle 'lock' button
 Then all doors should be 'locked'

Examples:

door_1_state	door_2_state	door_3_state	door_4_state
unlocked	unlocked	unlocked	unlocked
unlocked	unlocked	unlocked	locked
unlocked	unlocked	locked	unlocked
unlocked	unlocked	locked	locked
unlocked	locked	unlocked	unlocked
unlocked	locked	unlocked	locked
unlocked	locked	locked	unlocked
unlocked	locked	locked	locked
locked	unlocked	unlocked	unlocked
locked	unlocked	unlocked	locked
locked	unlocked	locked	unlocked
locked	unlocked	locked	locked
locked	locked	unlocked	unlocked
locked	locked	unlocked	locked
locked	locked	locked	unlocked
locked	locked	locked	locked

Figura 10 – Primeiro cenário *Gherkin* da história de usuário 1.

A cláusula *Given* utiliza quatro passos para definir os valores iniciais de cada porta, de forma que cada um recebe uma variável correspondente a uma coluna da tabela de exemplos. Essa tabela contém 16 linhas, que representam todas as combinações possíveis entre os estados “travada” e “destravada” para as quatro portas.

Na cláusula *When*, é descrita a ação do usuário ao pressionar o botão de travamento. Em seguida, a cláusula *Then* especifica que todas as portas devem estar travadas, validando assim o comportamento esperado de assegurar o veículo independentemente de seu estado inicial.

O último exemplo a ser coberto é: “aquele em que uma pessoa não autorizada tentou abrir a porta do veículo travado usando a maçaneta”. Ele é validado por um cenário em que a porta continua fechada após a o botão de abertura ser pressionado, tendo em vista que ela está travada.

Para evitar a duplicação de cenários, a tabela de exemplos é novamente utilizada, permitindo a execução automática do teste para cada porta individualmente, variando apenas o ponto de interação. Ao final, o cenário assume a seguinte forma:

```

Scenario Outline: Door cannot be released when locked
  Given the door <door_id> is 'locked'
  When I 'press' the door <door_id> release button
  Then the door <door_id> should be 'held'

  Examples:
  | door_id |
  | 1       |
  | 2       |
  | 3       |
  | 4       |

```

Figura 11 – Segundo cenário *Gherkin* da história de usuário 1.

4.4.2 História de usuário 2

A segunda história de usuário tem grande semelhança com a primeira e possui exemplos mapeados com casos de uso que são análogos à de travamento das portas. O primeiro cenário criado para esta história é:

```

Scenario Outline: Unlocking all doors
  Given the door '1' is <door_1_state>
  And the door '2' is <door_2_state>
  And the door '3' is <door_3_state>
  And the door '4' is <door_4_state>

  When I press the vehicle 'unlock' button
  Then all doors should be 'unlocked'

  Examples:
  | door_1_state | door_2_state | door_3_state | door_4_state |
  | unlocked    | unlocked    | unlocked    | unlocked    |
  | unlocked    | unlocked    | unlocked    | locked      |
  | unlocked    | unlocked    | locked      | unlocked    |
  | unlocked    | unlocked    | locked      | locked      |
  | unlocked    | locked      | unlocked    | unlocked    |
  | unlocked    | locked      | unlocked    | locked      |
  | unlocked    | locked      | locked      | unlocked    |
  | unlocked    | locked      | locked      | locked      |
  | locked      | unlocked    | unlocked    | unlocked    |
  | locked      | unlocked    | unlocked    | locked      |
  | locked      | unlocked    | locked      | unlocked    |
  | locked      | unlocked    | locked      | locked      |
  | locked      | locked      | unlocked    | unlocked    |
  | locked      | locked      | unlocked    | locked      |
  | locked      | locked      | locked      | unlocked    |
  | locked      | locked      | locked      | locked      |

```

Figura 12 – Primeiro cenário *Gherkin* da história de usuário 2.

Este cenário define que, com as pré-condições que cobrem todas as possíveis combi-

nações de travamento e destravamento para cada porta, após o botão de destravamento ser pressionado, o veículo será destravado. Ele cobre os dois primeiros exemplos mapeados:

- Aquele em que o usuário destravou as portas e entrou no carro;
- Aquele em que algumas portas já estavam destravadas.

Para o último exemplo, o seguinte cenário será utilizado:

```
Scenario Outline: Door can be released when unlocked
  Given the door <door_id> is 'unlocked'
  When I 'press' the door <door_id> release button
  Then the door <door_id> should be 'released'

  Examples:
  | door_id |
  | 1       |
  | 2       |
  | 3       |
  | 4       |
```

Figura 13 – Segundo cenário *Gherkin* da história de usuário 2.

Este cenário, que é executado para cada uma das portas, define que a porta será aberta após o botão de abertura ser pressionado, tendo em vista que ela está destravada. Ele cobre o seguinte exemplo mapeado:

- Aquele em que o usuário abriu a porta que estava destravada.

4.4.3 História de usuário 3

Para especificar o comportamento descrito nesta história, é necessário criar um novo tipo de passo que valide qual resposta de feedback foi acionada pelo veículo. Isso pode ser feito de acordo com o cenário:

```
Scenario Outline: Locking feedback when locking or unlocking the vehicle
  When I press the vehicle <operation> button

  Then all doors should be <state>
  And I should receive a <feedback> feedback

  Examples:
  | operation | state   | feedback               |
  | lock      | locked  | locking confirmation   |
  | unlock    | unlocked | unlocking confirmation |
```

Figura 14 – Primeiro cenário *Gherkin* da história de usuário 3.

Esse cenário descreve que, ao pressionar o botão correspondente, todas as portas devem ser travadas ou destravadas, e um feedback específico deve ser emitido. A cláusula *Given* foi omitida neste caso, pois o estado inicial do veículo - seja total ou parcialmente travado/destravado - não deve influenciar o resultado esperado. Ou seja, independentemente da condição inicial, o sistema deve aplicar a ação solicitada e fornecer o feedback apropriado.

Os seguintes exemplos são cobertos neste cenário:

- Aquele em que o usuário travou seu veículo e ficou na dúvida se as portas foram travadas ou destravadas;
- Aquele em que o usuário destravou seu veículo e ficou na dúvida se as portas foram travadas ou destravadas;

O último exemplo dessa história é: “aquele em que o usuário tentou travar seu veículo sem perceber que uma das portas ficou aberta” e será coberto no seguinte cenário:

Scenario Outline: Failure feedback when trying to lock the vehicle with one door open

Given the door <door_id> is 'unlocked'
 And the door <door_id> is <door_open_state>

When I press the vehicle 'lock' button

Then all doors should be 'locked'
 And I should receive a <feedback_received> feedback

Examples:

door_id	door_open_state	feedback_received
1	open	operation failed
2	open	operation failed
3	open	operation failed
4	open	operation failed
1	closed	locking confirmation
2	closed	locking confirmation
3	closed	locking confirmation
4	closed	locking confirmation

Figura 15 – Segundo cenário *Gherkin* da história de usuário 3.

Este cenário estabelece como pré-condição que uma das quatro portas está destravada, podendo estar aberta ou fechada, conforme os valores definidos na tabela de exemplos. Com base nesse estado inicial, ao pressionar o botão de travamento, o sistema deve emitir um feedback específico: “operação falha” caso a porta esteja aberta, ou “confirmação de travamento” caso ela esteja fechada.

Além disso, o cenário especifica que, independentemente de a porta estar aberta ou fechada, todas as portas devem ser marcadas como travadas após a ação. Isso está de acordo com a análise realizada durante a investigação das perguntas desta história, a qual

identificou que o travamento ou destravamento de uma porta aberta é possível. Isso ocorre porque o conceito de “estado da porta” não está vinculado a uma ação física imediata, mas sim a um estado lógico atribuído pelo controlador que é definido de forma virtual no sistema.

4.4.4 História de usuário 4

Para cobrir essa história, o primeiro exemplo considerado é “aquele em que o usuário tinha a chave no bolso e usou a maçaneta”, o qual está representado pelo seguinte cenário:

```
Scenario Outline: Unlocking a single door when I have the key
  Given the door <door_id> is 'locked'
  And I have an authenticated key with me

  When I 'press' the door <door_id> release button

  Then the door <door_id> should be 'unlocked'
  And the door <door_id> should be 'released'

  Examples:
  | door_id |
  | 1       |
  | 2       |
  | 3       |
  | 4       |
```

Figura 16 – Primeiro cenário *Gherkin* da história de usuário 4.

O cenário define que a porta encontra-se inicialmente travada e, ao ser pressionado o botão de abertura enquanto o usuário está em posse da chave, ela deveria ser destravada e aberta. A tabela de exemplos foi novamente utilizada para assegurar que esse comportamento seja testado em todas as portas.

O próximo exemplo é “aquele em que outra pessoa abriu outra porta simultaneamente”, o qual está coberto pelo seguinte cenário:


```

Scenario Outline: Unlocking multiple single doors when I have the key
  Given all doors are 'locked'
  And I have an authenticated key with me

  When I 'press' the door <door_id_1> release button
  And I 'press' the door <door_id_2> release button

  Then the door <door_id_1> should be 'unlocked'
  And the door <door_id_1> should be 'released'
  And the door <door_id_2> should be 'unlocked'
  And the door <door_id_2> should be 'released'

  Examples:
  | door_id_1 | door_id_2 |
  | 1         | 2         |
  | 1         | 3         |
  | 1         | 4         |
  | 2         | 1         |
  | 2         | 3         |
  | 2         | 4         |
  | 3         | 1         |
  | 3         | 2         |
  | 3         | 4         |
  | 4         | 1         |
  | 4         | 2         |
  | 4         | 3         |

```

Figura 17 – Segundo cenário *Gherkin* da história de usuário 4.

A estrutura do cenário é semelhante à anterior, porém com a adição de novos passos que simulam o pressionamento do botão, o destravamento e a abertura de duas portas simultaneamente. Para isso, foram utilizadas variáveis distintas para cada porta, permitindo que a tabela de exemplos cubra todas as possíveis combinações de abertura simultânea entre duas portas.

Por fim, o último exemplo é “aquele em que o usuário destravou a porta e a fechou de novo”, que é coberto pelo cenário:

```

Scenario Outline: Individual door is left unlocked
  Given the door <door_id> is 'locked'
  And I have an authenticated key with me

  When I 'press' the door <door_id> release button
  And I wait '1' seconds
  And I 'release' the door <door_id> release button

  Then the door <door_id> should be 'unlocked'
  And the door <door_id> should be 'held'

Examples:
  | door_id |
  | 1       |
  | 2       |
  | 3       |
  | 4       |

```

Figura 18 – Terceiro cenário *Gherkin* da história de usuário 4.

Este cenário também é executado repetidamente para cada porta, e apresenta a diferença que, após 1 segundo de espera, o botão de abertura que havia sido pressionado é solto. Com essa sequência de execução, a resposta do sistema é a de manter a porta destravada, mas de deixar o trinco da trava em sua posição fechada - o que equivale a porta estar sendo segurada.

4.4.5 História de usuário 5

Nesta história, a utilização de medidas de tempo é um elemento essencial para garantir que a resposta do sistema esteja alinhada ao comportamento esperado. Isso se deve à regra que determina que o auto-travamento das portas deve ocorrer 15 segundos após o destravamento, caso nenhuma porta seja aberta nesse intervalo.

O primeiro exemplo nesta história é “aquele em que o usuário destravou mas não foi acessar o carro” que é coberto pelo cenário:

Scenario Outline: Auto relocking

Given all doors are <initial_state>
 And the vehicle 'unlock' button has been pressed

When I wait <wait_time> seconds

Then all doors should be <expected_state>

Examples:

	initial_state	wait_time	expected_state
	locked	5	unlocked
	locked	10	unlocked
	locked	14	unlocked
	locked	15	locked
	locked	16	locked
	unlocked	5	unlocked
	unlocked	10	unlocked
	unlocked	14	unlocked
	unlocked	15	unlocked
	unlocked	16	unlocked

Figura 19 – Primeiro cenário *Gherkin* da história de usuário 5.

Esse cenário descreve que, após o botão de destravamento ser pressionado, seguido da espera de um determinado valor de tempo, o estado final das portas deve corresponder ao valor definido na variável de saída. O auto-travamento é efetivamente acionado nas linhas 4 e 5 da tabela, que representam portas inicialmente travadas, seguidas por uma espera que atende ao requisito de 15 segundos ou mais.

Os valores de tempo utilizados na tabela foram selecionados para representar situações reais e cobrir tanto casos em que a condição de tempo é satisfeita quanto situações de limite. Como nas linhas que utilizam 14, 15 e 16 segundos, que validam que o sistema responde corretamente em valores que são extremamente próximos da janela temporal especificada.

O próximo exemplo é “aquele em que o usuário destravou e foi acessar o carro”, ilustrando uma situação em que a condição para o auto-travamento não é satisfeita, pois uma das portas foi aberta. Esse caso é coberto no seguinte cenário:

```

Scenario Outline: Not auto relocking if release button is pressed
  Given all doors are 'locked'
  And the vehicle 'unlock' button has been pressed
  And the release button on door <door_id> is 'released'
  And the release button on door <door_id> is 'pressed'

  When I wait '15' seconds

  Then all doors should be 'unlocked'

Examples:
  | door_id |
  | 1       |
  | 2       |
  | 3       |
  | 4       |

```

Figura 20 – Segundo cenário *Gherkin* da história de usuário 5.

Neste exemplo, não foram definidos valores de tabela para testar múltiplas variações de estado inicial do veículo ou de tempos de espera. Essa decisão se deve ao fato de que o próprio exemplo descreve explicitamente uma condição na qual, mesmo a espera de tempo sendo satisfeita, o auto-travamento não deve ser acionado.

A pré-condição definida no primeiro passo estabelece que todas as portas estão inicialmente travadas, o que atende ao requisito inicial para o auto-travamento. A diferença ocorre no momento em que, após o uso do botão de destravamento, o botão de abertura de uma das portas é pressionado. Assim, mesmo após a espera de 15 segundos (tempo exigido para o auto-travamento), as portas devem permanecer destravadas.

O terceiro exemplo desta história é “aquele em que o usuário pressionou o botão de destravamento duas vezes para manter o carro destravado”, representando outra condição em que o auto-travamento não deve ser acionado. Esse caso é coberto pelo seguinte cenário:

```

Scenario: Not auto relocking if unlock button is pressed twice
  Given all doors are 'locked'
  And the vehicle 'unlock' button has been pressed
  And the vehicle 'unlock' button has been pressed

  When I wait '15' seconds

  Then all doors should be 'unlocked'

```

Figura 21 – Terceiro cenário *Gherkin* da história de usuário 5.

Assim como no cenário anterior, não foram utilizadas variáveis para representar diferentes condições iniciais, pois o próprio exemplo também estabelece uma condição em

que o auto-travamento não deve ocorrer. A diferença aqui está na inclusão de um segundo acionamento do botão de destravamento logo no início da sequência.

Por fim, o último exemplo desta história é “aquele em que o usuário destravou o veículo quando ele já estava acessível”, descrevendo a situação em que o auto-travamento não deve ser acionado caso pelo menos uma das portas já esteja destravada no início. Esse caso é tratado no seguinte cenário:

Scenario Outline: Not auto relocking if one or more doors are unlocked

Given the door '1' is <door_1_state>
 And the door '2' is <door_2_state>
 And the door '3' is <door_3_state>
 And the door '4' is <door_4_state>
 And the vehicle 'unlock' button has been pressed

When I wait '15' seconds

Then all doors should be <final_state>

Examples:

door_1_state	door_2_state	door_3_state	door_4_state	final_state
unlocked	unlocked	unlocked	unlocked	unlocked
unlocked	unlocked	unlocked	locked	unlocked
unlocked	unlocked	locked	unlocked	unlocked
unlocked	unlocked	locked	locked	unlocked
unlocked	locked	unlocked	unlocked	unlocked
unlocked	locked	unlocked	locked	unlocked
unlocked	locked	locked	unlocked	unlocked
unlocked	locked	locked	locked	unlocked
locked	unlocked	unlocked	unlocked	unlocked
locked	unlocked	unlocked	locked	unlocked
locked	unlocked	locked	unlocked	unlocked
locked	unlocked	locked	locked	unlocked
locked	locked	unlocked	unlocked	unlocked
locked	locked	unlocked	locked	unlocked
locked	locked	locked	unlocked	unlocked
locked	locked	locked	locked	locked

Figura 22 – Quarto cenário *Gherkin* da história de usuário 5.

Neste cenário, os quatro primeiros passos definem os estados iniciais de cada porta, de forma semelhante ao que foi feito na primeira história. Isso permite configurar todas as combinações possíveis de portas travadas ou destravadas como estado inicial.

Na condição inicial, após o estado de cada porta ser definido com base na tabela, o botão de destravamento é pressionado e leva ao destravamento inicial de todas as portas. Em seguida, é realizada uma espera de 15 segundos e o estado de travamento final é avaliado, verificando se o auto-travamento foi acionado.

Em todas as linhas da tabela de exemplos em que pelo menos uma porta já estava destravada - todas, exceto a última - o destravamento é mantido mesmo após a espera. Já na última linha, em que todas as portas estavam travadas inicialmente, a condição para o

auto-travamento é satisfeita, resultando no estado final de todas as portas travadas.

5 RESULTADOS

5.1 MODELAGEM ITERATIVA DO SISTEMA EM SIMULINK

Após a finalização da criação dos cenários *Gherkin*, a implementação prossegue com o desenvolvimento do modelo e das definições de passos. Inicialmente, o modelo é criado dentro da estrutura do projeto, em uma nova pasta denominada *model* que também inclui:

- **feature__model.slx** - modelo caixa preta, utilizado nos testes;
- **main.slx** - modelo de caixa branca, que contém a lógica do sistema e é referenciado como um bloco subsystem dentro do modelo **feature__model.slx**;
- **write__to__model.m** - função MATLAB responsável por operações de escrita no modelo;
- **read__from__model.m** - função MATLAB responsável por operações de leitura no modelo;
- **initialize__model.m** - função matlab que inicia o modelo e torna a API da aplicação disponível para as funções das definições de passo.

O modelo **feature__model.slx** segue a estrutura do diagrama de caixa preta, contendo apenas as entradas e saídas do sistema, sem detalhar sua lógica interna. Para isso, são utilizados blocos de constantes como entradas e *displays* como saídas.

Durante a execução dos testes, as definições de passos simulam o comportamento do sistema ao interagir com suas entradas, modificando os valores das constantes por meio da função **write__to__model**. Para validar a resposta do sistema, os valores dos displays de saída, determinados pela lógica interna do modelo, são obtidos utilizando a função **read__from__model**.

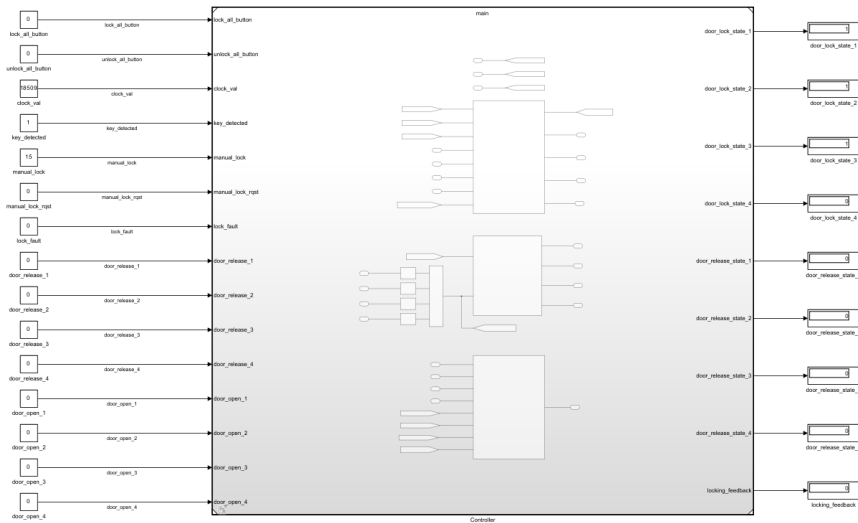


Figura 23 – Modelo `feature_model.slx` que aplica o conceito de caixa preta.

O modelo apresentado na Figura 23 contém entradas auxiliares que ainda não estão contempladas no diagrama de caixa preta. Essas entradas foram adicionadas para tornar certos comportamentos do sistema testáveis durante o desenvolvimento do modelo e serão detalhadas ao longo desta seção.

O segundo modelo, **main.slx**, é referenciado no bloco central do modelo anterior, denominado *controller*. Ele é responsável pela implementação da lógica interna do sistema em desenvolvimento, utilizando as entradas definidas como blocos *Inport* e as saídas como *Outport*, conectadas às constantes e *displays* do **feature_model.slx**.

Em seguida, o desenvolvimento do **environment.py** inclui funções em Python que tornam as operações de escrita e leitura do modelo acessíveis às definições de passos. Para isso, são criadas diversas funções utilitárias, permitindo realizar ações como:

- Iniciar a simulação;
- Parar a simulação;
- Executar operações de escrita no modelo;
- Executar operações de leitura no modelo.

Essas funções utilitárias ficam disponíveis por meio do objeto *context*, o qual pode ser acessado diretamente na lógica das funções das definições de passos. Com isso, cada passo consegue interagir com o modelo ao executar as operações especificadas no arquivo **environment.py**.

Para garantir uma execução iterativa dos testes, cada história é validada individualmente em um primeiro momento. Após a sua aprovação, todas as histórias anteriores

são executadas novamente, assegurando que a implementação de novas funcionalidades não comprometa o comportamento já estabelecido.

A execução dos testes é realizada após a definição da lógica interna das funções de cada passo. Os resultados são então documentados em um relatório automático, gerado em formato **.xml**, que descreve detalhadamente todas as falhas encontradas, conforme o padrão da figura 24:

```
<testsuite name="lock_all.Lock all"
tests="20"
errors="0"
failures="0"
skipped="0"
time="330.643892"
```

Figura 24 – Relatório de testes - resultado da execução do arquivo **.feature**.

O relatório também apresenta os resultados da execução de passos específicos, organizados de acordo com a figura 25:

```
@scenario.begin
Scenario Outline: Locking all doors -- @1.1
  Given I do not have an authenticated key with me ... passed in 0.265s
  And my vehicle is 'locked' with no release buttons pressed ... passed in 3.104s
  And all doors are 'closed' ... passed in 1.038s
  Given the door '1' is unlocked ... passed in 3.097s
  And the door '2' is unlocked ... passed in 3.117s
  And the door '3' is unlocked ... passed in 3.121s
  And the door '4' is unlocked ... passed in 3.102s
  When I press the vehicle 'lock' button ... passed in 0.629s
  Then all doors should be 'locked' ... passed in 1.026s
@scenario.end
```

Figura 25 – Relatório de testes - resultados por cenário.

O relatório apresentado na Figura 24 refere-se à execução dos cenários da história de usuário 1, como indicado pelo nome do arquivo **.feature** **lock_all.Lock All**. Os resultados indicam a realização de um total de 20 testes, que são compostos pelos 16 exemplos do primeiro cenário e pelos 4 exemplos do segundo. Observa-se também que a execução durou 330 segundos e não apresentou erros, falhas ou testes pulados, indicando que todos os testes foram aprovados.

Na seção de execução detalhada do relatório, ilustrada na Figura 25, observa-se que o cenário em questão é **Locking all doors** (primeiro cenário do arquivo **.feature**), correspondente ao exemplo **@1.1** (primeira linha da primeira tabela de exemplos). Cada passo do cenário apresenta seu respectivo resultado de execução, identificado como **passed**, além do tempo necessário para sua conclusão.

Além das informações detalhadas nas Figuras 24 e 25, para facilitar a análise de falhas, as funções implementadas em **environment.py** registram no relatório todas as interações realizadas com o modelo. Isso permite validar manualmente se a execução dos testes ocorreu conforme o esperado.

5.1.1 Execução dos testes de aceitação da História de Usuário 1

A execução dos cenários desta história inicia-se com o modelo vazio, o que resulta em falhas na validação de dois comportamentos:

- Definição dos estados iniciais de travamento das portas, conforme descrito na cláusula *Given*;
- Travamento de todas as portas ao pressionar o botão de travamento.

O primeiro ponto decorre do fato de que, em ambos os cenários, a cláusula *Given* estabelece valores iniciais para o estado de travamento de cada porta. Contudo, o teste falha porque o modelo não possui a implementação de um mecanismo que permita modificar diretamente o valor da saída nesse momento.

Para viabilizar essa estrutura, foi criada uma interface auxiliar denominada **manual_lock**, que possibilita alterar o estado inicial das portas sem recorrer aos botões físicos ou ao sistema *Keyless Access*. Essa interface executa uma operação de escrita direta no bloco *Data Store Memory*, responsável por armazenar o estado de travamento das portas no modelo.

A adoção de uma interface auxiliar para definir o estado inicial de travamento de cada porta tem como objetivo preservar a independência entre as funcionalidades. Conforme mencionado anteriormente, é possível obter qualquer combinação de travamento ou destravamento por meio do *Keyless Access* em portas específicas. Entretanto, para que essa sequência de entradas fosse utilizada diretamente na definição do primeiro passo, seria necessário que a história de usuário do *Keyless Access* já estivesse implementada e operacional.

Tanto a interface quanto o bloco de memória utilizam um valor inteiro de 4 bits, no qual cada bit representa o estado de travamento de uma porta. Como cada passo define de forma independente o estado de cada porta, a lógica da função de definição altera apenas o bit correspondente à porta em questão.

No segundo ponto identificado, a lógica implementada para o travamento das portas utiliza a interface de entrada do botão de travamento como gatilho de um *Triggered Subsystem*. Quando o valor do botão altera de 0 para 1, o valor 15 — que corresponde ao bit 1 ativado em cada uma das quatro portas — é gravado no *Data Store Memory*.

Com essa implementação, todos os testes da primeira história são aprovados, embora nenhuma lógica de abertura das portas tenha sido desenvolvida até o momento. Isso ocorre porque o valor esperado no resultado final corresponde ao estado de porta segura (saída igual a 0), o que coincide com o valor padrão atribuído pelo Simulink.

Essa coincidência leva à aprovação do teste, ainda que represente um falso positivo. A situação, contudo, será corretamente tratada na execução dos testes da segunda história, em que o valor esperado é o de porta aberta.

5.1.2 Execução dos testes de aceitação da História de Usuário 2

Na execução dos cenários da segunda história de usuário, foram identificadas falhas nos seguintes comportamentos:

- Destravamento das portas ao acionar o botão de destravamento;
- Abertura de portas específicas ao acionar o botão de abertura.

Para atender à condição de destravamento, aplica-se uma lógica semelhante à utilizada no travamento, baseada em um bloco *Triggered Subsystem*. Nesse caso, a escrita na memória é realizada com o valor inteiro 0, indicando que todos os quatro bits possuem valor 0.

Já para a condição de abertura das portas, a lógica de implementação pode ser construída de diferentes formas. Um exemplo, ainda que sem sentido do ponto de vista funcional do sistema, mas suficiente para satisfazer os testes, seria manter as portas permanentemente abertas, conectando diretamente o valor 1 às saídas. Contudo, essa abordagem é inviável, pois compromete a execução da primeira história, resultando em falhas na implementação anterior ao repetir os testes.

Sob essa perspectiva, o design que satisfaz ambos os testes consiste em permitir a abertura das portas apenas quando duas condições são atendidas simultaneamente:

- A porta está destravada - assegura que a abertura não ocorre enquanto a porta estiver travada;
- O botão de abertura está pressionado - garante que a porta permaneça fechada mesmo quando destravada, caso o botão não seja acionado.

Para implementar essa lógica, utiliza-se um bloco *AND*, que recebe como entradas o estado do botão de abertura e a condição de não travamento da porta. Essa estrutura é replicada para cada porta, empregando as interfaces correspondentes e operações de manipulação de bits para extrair do bloco de memória o estado atual de travamento de cada uma.

5.1.3 Execução dos testes de aceitação da História de Usuário 3

Na terceira história de usuário, foram identificadas as seguintes falhas de comportamento:

- Definição das condições para cada status;
- Utilização do tempo de espera para validar a saída de travamento em relação ao valor esperado;
- Indicação da atualização de status;
- Todas as portas devem ser fechadas durante o início dos testes.

O primeiro ponto observado nos testes ocorre porque a saída de status permanece inicialmente em 0, já que a lógica correspondente ainda não havia sido implementada. Para corrigir esse problema, as condições que determinam cada status foram modeladas em um bloco *Chart*, que possui os 3 estados a seguir:

- Confirmação de travamento: o botão de travamento é pressionado e, em seguida, todas as portas são travadas;
- Confirmação de destravamento: o botão de destravamento é pressionado e, em seguida, todas as portas são destravadas;
- Operação falha: o botão de travamento é pressionado enquanto ao menos um dos sensores de abertura indica que a porta está aberta.

Na teoria, as condições listadas deveriam satisfazer o comportamento esperado; entretanto, durante a execução dos testes, ainda foram observados erros. A investigação levantou a falha constatada no segundo ponto e exige uma pequena modificação nos cenários, incluindo uma condição temporal para validar que o status foi efetivamente gerado na saída.

Essa falha ocorre porque a implementação utiliza múltiplas entradas para determinar o status, o que ocasiona um leve atraso na execução do modelo. Esse efeito é particularmente relevante nos status de travamento e destravamento, que só são acionados após a atualização da saída de travamento.

Consequentemente, os testes falham não por falha no comportamento do sistema, mas porque o passo de validação do status é executado antes da atualização da saída no modelo. Para contornar esse problema, os passos dos cenários foram ajustados para incluir uma janela temporal, durante a qual o status deve ser gerado, conforme exemplificado abaixo:

And I should receive a <feedback> feedback within '500' ms

Para garantir que a implementação utilize valores de tempo compatíveis com a execução dos testes, foi criada uma nova interface auxiliar responsável por informar ao modelo o tempo atual da simulação. A cada passo executado, o arquivo **environment.py** atualiza o valor dessa interface de tempo com o valor gerado pelo Python. Esse valor é então considerado como uma condição adicional na determinação do status.

A terceira falha identificada decorreu de inconsistências observadas durante a execução dos testes. De forma aparentemente aleatória, o valor “sem status” era gerado repetidamente junto com o valor correto, como se o sistema estivesse oscilando continuamente entre os dois estados.

O problema foi evidenciado ao analisar o modelo, especificamente o comportamento interno do bloco *Chart*, que contém o diagrama de estados. Inicialmente, a condição para iniciar a validação baseava-se apenas no acionamento do botão de travamento ou destravamento, verificando se o seu valor era 1 (pressionado). Isso provocava a repetição contínua da validação enquanto o botão permanecia pressionado, gerando os resultados inconsistentes observados nos testes.

A solução adotada consistiu em fazer com que a validação considerasse não o estado do botão em si, mas a transição de solto para pressionado. Dessa forma, assim que o teste indica que o botão foi pressionado, apenas um único evento de validação é gerado, garantindo que o status seja atualizado uma única vez na saída.

Para assegurar que os testes conseguissem detectar esse tipo de falha, a interface de status também foi aprimorada. Agora é possível identificar não apenas qual status está sendo indicado, mas também quando a saída é gerada. Isso é implementado por meio de um valor inteiro que é incrementado toda vez que um novo status é gerado, permitindo acompanhar a sequência de eventos na saída. A interface passa a assumir valores de 0 a 10, com a seguinte codificação:

- 0 - Sem status
- 1, 4 e 7 - Confirmação de travamento
- 2, 5 e 8 - Confirmação de destravamento
- 3, 6 e 9 - Operação falha

Dessa forma, o incremento do valor de status é realizado de maneira condicional, dependendo do tipo de status a ser comunicado. Por exemplo, se o status atual for 2 e o sistema determinar que deve informar uma Confirmação de Travamento, o valor é incrementado para 4, que é o próximo valor disponível correspondente àquele status.

Quando o valor máximo é atingido, a interface realiza um loop, retornando aos valores iniciais. Por exemplo, se o estado atual for 8 e uma nova confirmação de destravamento precisar ser registrada, o valor é incrementado para 2.

Com base nessa estratégia, foi adicionada uma lógica de validação de status como função utilitária no **environment.py**. Essa função mantém uma lista de todos os valores de status, adicionando um novo item sempre que um novo status é gerado. Dessa forma, é possível validar não apenas se o resultado esperado foi alcançado, mas também se a quantidade correta de status foi registrada.

Por fim, um último problema foi identificado durante a execução repetida dos testes desta história: o feedback de operação falha era gerado em situações nas quais outro valor era esperado. A causa estava relacionada à interface do sensor de abertura das portas, que permanecia configurada como “aberta” em cenários que não especificavam explicitamente esse estado nos seus passos.

Esse comportamento ocorria porque, em cenários destinados a validar a saída considerando uma porta aberta, o valor atribuído ao modelo permanecia persistente, afetando a execução de cenários subsequentes ao manter indevidamente o estado de abertura.

Para corrigir essa inconsistência, foi adicionada uma precondição comum a todos os cenários, garantindo que cada teste seja iniciado com todas as portas fechadas. Essa configuração é feita por meio da funcionalidade *Background*, utilizando o seguinte passo:

```
Given all doors are 'closed'
```

Dessa forma, todos os cenários iniciam com o passo do *Background* e portanto as portas são sempre iniciadas com o valor fechado. Após a implementação dessa lógica, todos os cenários de todas as histórias foram executados novamente, sendo aprovados com sucesso.

5.1.4 Execução dos testes de aceitação da História de Usuário 4

Após a execução dos testes da quarta história de usuário, foram identificadas as seguintes falhas:

- A porta não é destravada quando o botão de abertura é pressionado, mesmo com a chave presente;
- No início de todos os cenários, a chave não deve estar presente.

A primeira falha ocorre na validação do comportamento em que a porta deve ser destravada ao pressionar o botão de abertura com a chave presente. Isso se deve ao fato

de que ainda não havia sido implementada uma lógica de destravamento que considerasse a presença da chave como critério para cada porta individual.

Para permitir essa funcionalidade, o destravamento de uma porta individual é realizado ao definir que um dos bits do *Data Store Memory* deve assumir o valor 0. Essa operação é acionada através de um Triggered Subsystem, cuja execução ocorre somente quando duas condições são simultaneamente satisfeitas:

- A chave está presente
- O botão de abertura da porta foi pressionado

Essa estrutura foi replicada para cada porta, ajustando a interface do botão de abertura correspondente e definindo qual bit do *Data Store Memory* deve ser setado para 0. Após a implementação, todos os testes da história foram aprovados. No entanto, um novo problema foi identificado ao executar a história de usuário 1.

Durante os testes do *Keyless Access*, a presença da chave é simulada no modelo ao escrever 1 no bloco de constante de entrada. O problema ocorre porque essa condição não é revertida em nenhum momento, fazendo com que, durante o teste que verifica que uma porta travada não pode ser aberta, o *Keyless Access* seja aplicado inadvertidamente, permitindo que a porta se abra e causando a falha no teste.

Este problema é exatamente igual ao caso demonstrado na história anterior, e sua solução também utiliza o Background para definir estados iniciais comuns para todos os cenários. Neste caso os seguintes passos foram incluídos:

```
Given I do not have an authenticated key with me
And my vehicle is 'locked' with no release buttons pressed
```

As definições desses passos garantem que a condição inicial de todas as entradas seja estabelecida no início da simulação, conforme descrito a seguir:

- A chave não está presente;
- Todas as portas estão travadas;
- Nenhum dos botões de abertura está pressionado.

Dessa forma, ao iniciar a execução dos cenários da primeira história, a chave permanece ausente, o *Keyless Access* não é acionado e a porta travada não se abre quando o botão de abertura é pressionado. Como resultado, todos os testes são aprovados.

5.1.5 Execução dos testes de aceitação da História de Usuário 5

Nesta história, foram identificados os seguintes problemas:

- O travamento automático não ocorre após a passagem do tempo;
- As condições de abertura de uma porta, o estado inicial de travamento das portas e o acionamento do botão de destravamento duas vezes não impedem o travamento automático;
- Problema na determinação correta do tempo decorrido na simulação.

Os dois primeiros pontos estão diretamente relacionados à implementação da lógica responsável pelo auto travamento. Para tratá-los, utiliza-se um bloco *Chart* contendo um diagrama de estados, no qual o pressionamento do botão de destravamento inicia a contagem de tempo. Esse diagrama é estruturado da seguinte maneira:

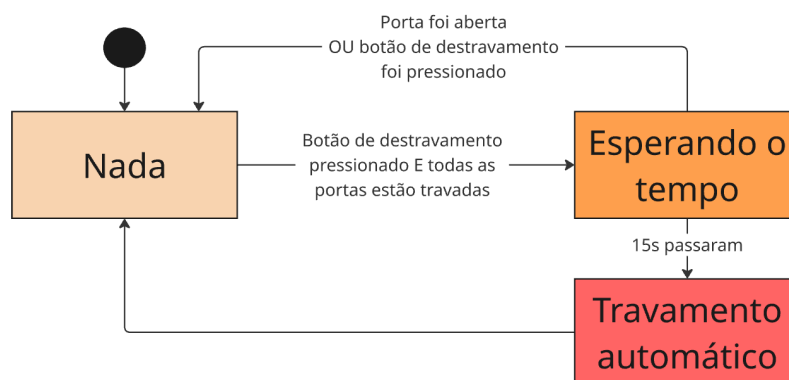


Figura 26 – Diagrama de estados da lógica do auto travamento.

O sistema é iniciado no estado “Nada”, no qual o auto-travamento não é verificado, permanecendo nessa condição até que seja atingida a situação válida para iniciar a contagem de tempo. Essa situação ocorre quando duas condições são simultaneamente satisfeitas:

- Todas as portas estão travadas;
- O botão de travamento foi pressionado.

A primeira condição garante que o auto-travamento não será aplicado caso o veículo não esteja seguro inicialmente, atendendo ao último cenário de teste em que uma ou mais portas permanecem destravadas no início. Durante essa transição, o sistema registra o valor atual do tempo de simulação, permitindo determinar o momento em que a condição de passagem de 15 segundos é satisfeita.

Uma vez realizada a transição para o estado “Esperando o tempo”, o sistema aguarda a passagem dos 15 segundos, enquanto monitora se alguma das condições que

impedem o auto-travamento é satisfeita. Nessa situação, o sistema retorna ao estado “Nada” sem acionar o auto-travamento caso pelo menos uma das seguintes condições seja atendida:

- Uma porta foi aberta;
- O botão de destravamento foi acionado.

Cada uma dessas condições corresponde a um cenário de teste, definindo os comportamentos em que o usuário manifesta a intenção de manter o veículo acessível. Por fim, caso nenhuma dessas condições seja satisfeita durante o período de espera, a passagem do tempo provoca a transição para o estado “Travamento automático”, acionando o mesmo *Triggered Subsystem* utilizado na história de travamento de todas as portas.

Dessa forma, o comportamento esperado do sistema em relação ao auto-travamento é completamente atendido. Entretanto, uma complicação técnica faz com que diversos testes falhem. Nos casos em que a espera é de 10 ou 14 segundos, os testes apresentam falha porque o auto-travamento é acionado antes que o tempo correto tenha decorrido.

A investigação revelou que esse problema ocorre devido à latência nas interações do Python com o modelo, que pode chegar a cerca de 500 ms por operação de escrita e 200 ms por operação de leitura. Como inúmeras operações são executadas a cada instante, o valor do tempo gerado pelo Python é continuamente incrementado em operações que deveriam ser praticamente instantâneas.

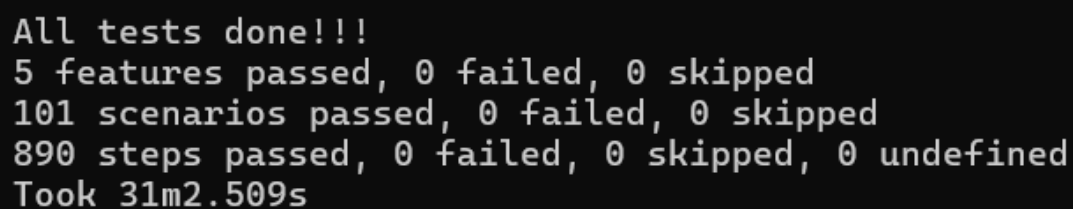
Para resolver essa questão, a lógica de geração do tempo de simulação em Python foi ajustada para contabilizar o tempo gasto nas interações com o modelo. Dessa forma, a passagem do tempo de simulação é incrementada apenas durante passos que envolvem espera de tempo.

Essa abordagem foi adotada para garantir que a latência decorrente da interação entre as ferramentas não comprometesse a validade dos cenários de teste. Como a execução do modelo ocorre de forma independente da execução dos cenários, a indicação de tempo precisa refletir uma representação fiel do tempo real. Alternativas como a utilização de *multi-threading* ou do tempo de simulação interno do Simulink ainda apresentariam problemas semelhantes e, por esse motivo, foram descartadas.

Após a correção da lógica de geração do tempo, todos os testes foram reexecutados, resultando em 100% de aprovação em todos os cenários de todas as histórias de usuário.

5.2 ANÁLISE DOS RESULTADOS

Ao fim do processo de modelagem iterativa, os resultados demonstrados na Figura 27 foram obtidos:



```
All tests done!!!  
5 features passed, 0 failed, 0 skipped  
101 scenarios passed, 0 failed, 0 skipped  
890 steps passed, 0 failed, 0 skipped, 0 undefined  
Took 31m2.509s
```

Figura 27 – Relatório de testes - execução final.

A execução completa dos testes levou pouco mais de 31 minutos e contemplou:

- 5 *features*;
- 101 cenários (incluindo combinação geradas pelas tabelas);
- 890 passos.

O número total de cenários e de passos executados demonstra a ampla cobertura de testes aplicados, que satisfazem todos os comportamentos definidos nas histórias de usuário. Além destes, O Quadro 2 demonstra um sumário das falhas nos testes que foram capturadas durante a etapa, assim como uma demonstração da melhoria necessária para solucionar cada problema:

Quadro 2 – Falhas de comportamento capturadas durante a modelagem iterativa

Falha Capturada	Melhoria Implementada em
Definição dos estados iniciais de travamento das portas, conforme descrito na cláusula Given	Modelo
Travamento de todas as portas ao pressionar o botão de travamento	Modelo
Destravamento das portas ao acionar o botão de destravamento	Modelo
Abertura de portas específicas ao acionar o botão de abertura	Modelo
Definição das condições para cada status	Modelo
Utilização do tempo de espera para validar a saída de travamento em relação ao valor esperado	Cenários
Indicação da atualização de status	Definições dos passos
Todas as portas devem ser fechadas durante o início dos testes	Cenários
A porta não é destravada quando o botão de abertura é pressionado, mesmo com a chave presente	Modelo
No início de todos os cenários, a chave não deve estar presente	Cenários
O travamento automático não ocorre após a passagem do tempo	Modelo
As condições de abertura de uma porta, o estado inicial de travamento das portas e o acionamento do botão de destravamento duas vezes não impedem o travamento automático	Modelo
Problema na determinação correta do tempo decorrido na simulação	Definições dos passos

Um total de 13 falhas foram levantadas, que culminaram na implementação de 8 melhorias no modelo, 3 melhorias no cenários e 2 melhorias nas definições dos passos.

O processo iterativo mostrou-se altamente eficiente para a modelagem do sistema, sobretudo quando comparado a metodologias alternativas de desenvolvimento e testes de software embarcado. Entre os principais benefícios, destaca-se a possibilidade de validar o sistema antes mesmo da escrita ou geração do código, que torna o a detecção de falhas e implementação de soluções mais rápidas e com menos custo associado.

Realizar a validação apenas no sistema físico acarreta complicações que não apenas atrasam a entrega do produto final, mas também elevam os custos do processo. Isso ocorre porque o nível de maturidade necessário para a execução de testes em hardware geralmente

só é alcançado meses após a conclusão da etapa de análise, na qual o design do sistema é definido.

Até que o sistema atinja o nível de maturidade necessário para que o design seja efetivamente colocado à prova, grande parte da implementação já terá sido realizada, demandando inúmeras horas de trabalho dos engenheiros de software. Qualquer problema identificado nessa fase exige processos extensos de diagnóstico e deve ser comunicado ao engenheiro de sistemas responsável pela análise, o que demanda mais tempo entre diferentes profissionais.

Nesse contexto, um erro de design não detectado durante a fase de definição torna-se extremamente oneroso, pois sua identificação ocorre apenas após meses de esforço de diversos engenheiros de diferentes departamentos. Além disso, para que uma solução seja estabelecida, torna-se necessário revisar o design, propor modificações, implementá-las e testá-las novamente — resultando em um elevado custo de retrabalho.

Na metodologia proposta por este trabalho, a execução dos testes pode ocorrer antes mesmo do desenvolvimento do código, podendo inclusive estar sob a responsabilidade do próprio engenheiro que realiza a análise do sistema. Isso proporciona maior confiabilidade em relação ao design e permite que o engenheiro mantenha controle direto sobre eventuais mudanças necessárias. Além disso, quanto mais cedo uma falha é identificada no processo de desenvolvimento, menor é o custo de sua correção.

Um exemplo que ilustra claramente essa eficiência ocorreu durante a execução dos testes da história de usuário 3, quando se constatou a necessidade de incluir uma condição temporal para validar o momento em que o feedback deveria ser realizado. Inicialmente, sem a visualização prática do sistema, essa condição não foi considerada, já que a validação estava restrita apenas ao acionamento do botão, sem contemplar a necessidade de verificar a resposta do sistema em função do tempo.

Outro benefício da metodologia proposta é a facilidade na identificação das causas dos erros de teste, como evidenciado nas três primeiras falhas da história de usuário 3. Os problemas estavam relacionados a detalhes específicos da implementação, em que o diagrama de estados gerou um loop que oscilava repetidamente entre as diferentes respostas de feedback.

A detecção desse exigiu uma visualização precisa do comportamento do sistema, mais especificamente em observar a execução em tempo real do diagrama de estados. Para isso, foi necessário executar os testes repetidamente, realizando pequenos ajustes no modelo a cada iteração.

Em metodologias que realizam testes apenas após a disponibilização do código embarcado, o processo equivalente de identificação de falhas demandaria diversas compilações seguidas de novas execuções de teste. Além disso, a instrumentação utilizada

nesse contexto frequentemente apresenta limitações na observação do comportamento do sistema, o que dificulta a detecção de erros como os observados.

Essas dificuldades são mitigadas pelo uso do ambiente de simulação, que se mostra altamente flexível por permitir alterações rápidas no modelo, sem exigir grande esforço de compilação ou implementação em hardware. Outro diferencial é a possibilidade de acompanhar a execução de forma visual, permitindo observar os diagramas de estados em tempo real, além de incluir *displays* — blocos que exibem valores em interfaces específicas — em qualquer ponto do sistema.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

A adaptação do processo de Behavior-Driven Development (BDD) para o contexto da Engenharia Automotiva demonstra como a aplicação de testes automatizados e o mapeamento de exemplos podem guiar o desenvolvimento de sistemas veiculares, resultando em produtos de maior qualidade. Essa metodologia foi implementada integrando conceitos do BDD, técnicas de desenvolvimento de software embarcado e práticas da indústria automotiva.

O trabalho evidenciou as complexidades inerentes ao desenvolvimento de sistemas automotivos, como a dependência de componentes mecânicos e eletrônicos e a necessidade de testes que exigem a interação com software embarcado. Para superar essas dificuldades, foram aplicadas estratégias de abstração do desenvolvimento em camadas e design orientado a modelo, garantindo a compatibilidade com o BDD.

A etapa de análise do sistema mostrou-se bastante produtiva, possibilitando a aplicação efetiva de técnicas de desenvolvimento ágil. O mapeamento de exemplos na definição do sistema resultou no total de 16 cenários criados, com algumas instâncias que derivaram de questionamentos levantados durante o processo. Isso é demonstrado no caso do exemplo “Aquele em que o usuário tentou travar seu veículo sem perceber que uma das portas ficou aberta”, que ocasionou na definição de um comportamento que garante que o usuário está ciente sobre situações em que, apesar do travamento das portas, o veículo ainda não pode ser assegurado.

Assumir a perspectiva do cliente e compreender o valor agregado ao produto revelou-se fundamental nessa etapa. A definição de funcionalidades orientadas ao comportamento desejado resultou em histórias de usuário mais robustas, capazes de traduzir requisitos em exemplos práticos que guiam a implementação.

Os cenários em Gherkin constituem o eixo central de todo o processo, promovendo alinhamento entre as etapas do desenvolvimento. O uso de linguagem natural, estruturada pelo padrão Given/When/Then, assegura que o comportamento definido seja compreensível para todos os envolvidos, além de gerar especificações executáveis que não apenas descrevem o sistema, mas também validam seu funcionamento.

Por fim, durante o desenvolvimento iterativo, todo o processo foi colocado à prova com a geração do produto final: o modelo do sistema. A execução repetitiva dos testes assegurou que as especificações fossem atendidas e que o produto desenvolvido apresentasse robustez e conformidade com os requisitos definidos. No fim do processo do desenvolvimento iterativo, o conjunto de cenários resultou em um relatório abrangente, assim como foi demonstrado na Figura 27.

É possível observar que as melhorias capturadas a cada iteração dos testes resultaram não apenas na evolução do produto — neste caso, o modelo do sistema — mas também no aprimoramento da própria definição do sistema, com cenários mais bem estruturados e definições de passos mais detalhadas. Cada modificação decorrente de um problema identificado nos testes, como detalhado no Quadro 2, não representa apenas a adição de lógica, mas sim um incremento na qualidade e na robustez do produto.

Esse aspecto torna-se ainda mais evidente ao considerar como a utilização do modelo possibilitou a detecção antecipada de falhas que dificilmente seriam percebidas sem a execução dos testes. Um exemplo claro foi a definição do feedback: na implementação inicial, ele era atualizado de forma repetitiva, gerando um loop que alternava continuamente entre a mensagem “Sem status” e o feedback esperado. A identificação desse comportamento, seguida pela análise da causa raiz e pela implementação de uma solução, permitiu eliminar uma falha que poderia resultar em insatisfação do usuário, ainda na fase de modelagem, antes mesmo do desenvolvimento do código.

Este tipo de metodologia que permite a descoberta e tratamento de possíveis erros tão cedo no desenvolvimento de um produto é uma ferramenta extremamente eficaz para lidar com as complexidades do setor automotivo.

6.1 TRABALHOS FUTUROS

Considerando o potencial identificado na proposta a aplicação do processo de BDD na Engenharia Automotiva, o desenvolvimento é continuado garantindo que a geração do produto seja validada também em um veículo físico. Para isso, os seguintes passos futuros devem ser realizados como parte de trabalhos que se preveem:

- Geração do software embarcado, a partir do modelo do sistema;
- Testes de software de aplicação, gerado diretamente a partir do modelo;
- Definição do hardware aplicado para o desenvolvimento do produto físico;
- Testes do software básico, desenvolvido para o hardware utilizado;
- Aplicação do software em uma plataforma de desenvolvimento embarcado;
- Testes de hardware, realizados no produto físico.

A abordagem aplicada para possibilitar a interface entre a execução dos testes a partir dos cenários *Gherkin* com o modelo em *Simulink* pode ser feita de maneira similar com outras ferramentas. Isso torna possível aplicar os testes de aceitação gerados pelo *Behave* compatíveis com qualquer sistema que possa estabelecer comunicação com o Python.

Dessa maneira, podem ser incluídos neste processo o uso ferramentas que permitam o teste do software gerado a partir do modelo, assim como de hardware para o produto físico.

Outra necessidade que deve ser atendida em trabalhos futuros é a definição de componentes físicos, como sensores e atuadores, que serão aplicados para desenvolver o produto final. Este passo é diretamente relacionado com o desenvolvimento do software básico, pois determina como a integração com o hardware deve ser feita.

A validação do software básico pode gerar a necessidade da inclusão de níveis de teste extras, que vão além da metodologia de caixa-preta. Isto deve-se ao fato de que nele são criadas funções que realizam a abstração do hardware, não necessariamente expressando um valor à nível do usuário, mas apenas provendo funcionalidades para o software de aplicação.

REFERÊNCIAS

- Atlassian. **Agile Teams**. n.d. Acesso em: 12 jul. 2025. Disponível em: <<https://www.atlassian.com/agile/teams>>. Citado 3 vezes nas páginas 11, 16 e 31.
- AUTOSAR. **Specification of Key Manager**. 2024. Versão R24-11. Acesso em: 02 ago. 2025. Disponível em: <https://www.autosar.org/fileadmin/standards/R24-11/CP/AUTOSAR_CP_SWS_KeyManager.pdf>. Citado 2 vezes nas páginas 23 e 42.
- AUTOSAR. **Classic Platform**. n.d. Acesso em: 26 ago. 2025. Disponível em: <<https://www.autosar.org/standards/classic-platform>>. Citado 2 vezes nas páginas 13 e 22.
- Behave. **Behave Documentation**. 2025. Acesso em: 12 jul. 2025. Disponível em: <<https://behave.readthedocs.io/en/latest/>>. Citado 2 vezes nas páginas 17 e 48.
- Cucumber.io. **Cucumber Documentation**. 2024. Acesso em: 12 jul. 2025. Disponível em: <<https://cucumber.io/docs/>>. Citado 2 vezes nas páginas 17 e 19.
- Cucumber.io. **History of BDD**. 2024. Acesso em: 12 jul. 2025. Disponível em: <<https://cucumber.io/docs/bdd/history/>>. Citado na página 17.
- FINIO, M.; DOWNIE, A. **What is a software-defined vehicle?** 2025. Acesso em: 27 ago. 2025. Disponível em: <<https://www.ibm.com/think/topics/software-defined-vehicle>>. Citado na página 11.
- GLOCKER, T.; MANTERE, T.; ELMUSRATI, M. A protocol for a secure remote keyless entry system applicable in vehicles using symmetric-key cryptography. p. 310–315, 2017. Citado 2 vezes nas páginas 23 e 42.
- GSMH, R. B. **Automotive Handbook**. 11. ed. New Jersey, USA: Wiley, 2022. 1430-1439 p. Citado 4 vezes nas páginas 11, 23, 26 e 29.
- LAWRENCE, R.; RAYNER, P. **Behavior-Driven Development with Cucumber: Better collaboration for better software**. 1. ed. Boston, MA: Addison-Wesley Professional, 2019. Citado 2 vezes nas páginas 18 e 29.
- MathWorks. **Model-Based Design with Simulation**. 2020. Acesso em: 26 ago. 2025. Disponível em: <<https://www.mathworks.com/content/dam/mathworks/white-paper/gated/model-based-design-with-simulation-white-paper.pdf>>. Citado 2 vezes nas páginas 13 e 21.
- NORTH, D. **Introducing Behavior-Driven Development**. 2006. Acesso em: 12 jul. 2025. Disponível em: <<https://dannorth.net/introducing-bdd/>>. Citado 3 vezes nas páginas 11, 16 e 17.
- REHKOPF, M. **User Stories**. 2025. Acesso em: 12 jul. 2025. Disponível em: <<https://www.atlassian.com/agile/project-management/user-stories>>. Citado na página 17.

REIF, K. (Ed.). **Automotive Mechatronics**. 1. ed. Wiesbaden, Germany: Springer Fachmedien, 2015. 488-493 p. P. 490. Disponível em: <https://fliphtml5.com/eraqv/csdz/%5BKonrad-Reif-%28eds.%29%5D-Automotive-Mechatronics_-Auto%28z-lib.org%29/293/>. Citado 7 vezes nas páginas 7, 11, 12, 23, 26, 33 e 34.

SANGIOVANNI-VINCENTELLI, A.; MARTIN, G. Platform-based design and software design methodology for embedded systems. **IEEE Design and Test of Computers**, v. 18, n. 6, p. 23–33, 2001. Citado na página 21.

SOLIS, C.; WANG, X. **A Study of the Characteristics of Behaviour Driven Development**. 2011. 383-387 p. Doi: 10.1109/SEAA.2011.76. Citado 2 vezes nas páginas 16 e 25.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. Tradução: Ivan Bosnic e Kalinka Oliveira; Revisão técnica: Kechi Hiramã. Citado na página 23.

TEAM, S. R. E. **RFID Key Fobs Explained: Types, Frequencies, and How to Duplicate**. 2025. Acesso em: 27 ago. 2025. Disponível em: <<https://sumokey.com/pages/rfid-key-fobs-explained>>. Citado 2 vezes nas páginas 24 e 42.

TERSHAK, A. T.; THIENEMAN, M. D. **Refrigerator door ajar alarm with variable delay**. 1986. Patente US4566285A. Arquivado em 26 de janeiro de 1984. Disponível em: <<https://patentimages.storage.googleapis.com/67/fe/2b/219aa6ea4967ca/US4566285.pdf>>. Citado na página 39.

VOLKSWAGEN. **Manual de instruções Tera**. 2025. Manual do Tera, pág. 61-69. Disponível em: <<https://www.vw.com.br/pt/servicos-e-acessorios/servicos-e-produtos/manuais-e-garantia/manuais.html>>. Citado 5 vezes nas páginas 11, 26, 29, 30 e 43.

WYNNE, M. **Introducing Example Mapping**. 2015. Acesso em: 12 jul. 2025. Disponível em: <<https://cucumber.io/blog/bdd/example-mapping-introduction/>>. Citado 2 vezes nas páginas 18 e 31.