Instituto Federal do Maranhão

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

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

Santa Inês - MA
2025

ESTA SERÁ SUA FICHA CATALOGRÁFICA

ESTA SERÁ SUA FOLHA DE APROVAÇÃO

Lorem Ipsum...

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

Se pude enxergar a tão grande distância, foi subindo nos ombros de gigantes.

Isaac Newton
Carta à Robert Hooke, 1676

RESUMO

O presente trabalho tem como objetivo explorar a aplicação da metodologia de Desenvolvimento Orientado por Comportamento (BDD) no contexto de software embarcado, utilizando o mapeamento de exemplos e a linguagem Gherkin para definição de funcionalidades, geração de testes de aceitação automatizados e modelagem do sistema. Serão apresentados os principais benefícios decorrentes do uso dessa abordagem, como a ampla cobertura de testes, e a geração de documentação viva que valida o comportamento do sistema contra os resultados esperados. A metodologia será aplicada no desenvolvimento de um protótipo que simula um sistema automotivo de travamento de portas, atendendo a todas as especificações de comportamento previamente definidas.

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

ABSTRACT

This study aims to explore the application of the Behavior Driven Development (BDD) methodology in the context of embedded software, using example mapping and the Gherkin language for defining functionalities, generating automated acceptance tests and modeling the system. The main benefits of this approach will be presented, such as extensive test coverage and the creation of living documentation that validates the system's behavior against the expected results. The methodology is going to be applied on the development of a prototype that simulates an automotive door locking system, fulfilling all previously defined behavioral specifications.

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

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

LI	Lorem Ipsum
LII	Lorem Ipsum Ipsum

SUMÁRIO

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.

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 PROBLEMÁTICA

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?”.

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 (Bosch, 2022), sendo necessário integrar componentes mecânicos, considerar falhas potenciais e contemplar casos de uso extremos.

O principal desafio deste trabalho está em identificar técnicas capazes de capturar essas complexidades e torná-las compatíveis com as premissas do BDD.

1.2 OBJETIVOS GERAIS E ESPECÍFICOS

Para responder à problemática levantada, o presente estudo tem como 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. Para isso, os objetivos específicos serão focados em:

- Abordar as complexidades dos sistemas automotivos de maneira compatível com o processo de BDD;
- Empregar técnicas de desenvolvimento que são amplamente utilizadas na indústria automotiva;
- Realizar o levantamento e definição de funcionalidades considerando possíveis limitações decorrentes da dependência de hardware e de sistemas mecânicos;
- Definir um escopo de desenvolvimento que possibilite a realização do design do sistema forma clara e compreensível;
- Empregar testes de aceitação automatizados para possibilitar o desenvolvimento do sistema de maneira iterativa, consistente com o design definido;
- Possibilitar a adaptação às descobertas surgidas durante a aplicação do processo, garantindo a melhoria contínua do design do sistema e do valor entregue ao usuário.

1.3 METODOLOGIA

A metodologia utilizada neste estudo baseia-se na revisão da bibliografia disponível, com o objetivo de esclarecer o processo de BDD e investigar como ele pode ser adaptado para incorporar práticas consolidadas na Engenharia Automotiva. Para tanto, serão aplicadas técnicas como o mapeamento de exemplos (LAWRENCE; RAYNER, 2019; WYNNE, 2015) , voltado ao levantamento e definição de funcionalidades, e testes de aceitação automatizados (SOLIS; WANG, 2011) , que servirão para guiar o desenvolvimento do sistema.

A implementação do sistema simulado também fará uso de técnicas típicas da Engenharia Automotiva, incluindo o Model Based Design (SANGIOVANNI-VINCENTELLI; MARTIN, 2001; MathWorks, 2020) e a geração automática de código segundo o padrão de software AUTOSAR (AUTOSAR, n.d.). Conceitos adicionais do domínio automotivo serão considerados para assegurar a adequada abstração nas diferentes etapas do desenvolvimento.

Espera-se que os resultados obtidos evidenciam os benefícios da integração do BDD no contexto automotivo, destacando sua contribuição para a qualidade e confiabilidade do sistema. A validação será realizada por meio da execução dos testes de aceitação e da análise do uso do sistema final sob a perspectiva do usuário.

1.4 ORGANIZAÇÃO DO TRABALHO

Este trabalho está estruturado da seguinte forma: o capítulo 2 apresenta a fundamentação teórica sobre BDD e testes automatizados, o capítulo 3 descreve a metodologia aplicada, o capítulo 4 detalha a implementação e os resultados obtidos, e o capítulo 5 apresenta as conclusões e considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação deste trabalho é a aplicação prática do Desenvolvimento Orientado a Comportamento (BDD) ao desenvolvimento de sistemas de software automotivo. Esse processo será aplicado no mapeamento de funcionalidades e modelagem 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, que compartilhou em seu blog (NORTH, 2006) as lições aprendidas ao aplicar e ensinar o processo de desenvolvimento orientado a testes (TDD). Segundo o mesmo, "Ele (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, como: "como eu testo isso?" ou "como eu nomeio este teste?".

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. Dessa forma, a sua nomenclatura se torna uma sentença natural e provoca a reflexão: Ao afirmar que "o sistema deveria fazer algo", é natural que surja a dúvida: "mas deveria mesmo?". 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.

A trajetória de Dan North, marcada por suas descobertas, consolidaram um processo de desenvolvimento robusto e que traz diversas práticas voltadas a entregar valor ao usuário de forma mais rápida. Esse amadurecimento é evidente nas características apontadas por (SOLIS; 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.

Para garantir que todas as características listadas sejam capturadas neste trabalho, o uso de histórias de usuário (REHKOPF, 2025) e da linguagem Gherkin (cucumber) (Cucumber.io, 2024b) serão aplicadas.

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; 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 comportamento 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. Na etapa de implementação, serão apresentados exemplos dessas perguntas e a forma como cada uma será tratada ao longo do processo.

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 (CUCUMBER)

A validação das funcionalidades definidas, como detalhado no 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).

Cucumber utiliza a gramática Gherkin para a escrita dos cenários, adotando as palavras-chave Given/When/Then (Dado que/Quando/Então), que permitem descrever o comportamento esperado do sistema de forma compreensível para todas as partes envolvidas. Esses cenários são traduzidos em testes automatizados que interagem com o software e verificam seu comportamento por meio de passos e definições de passos, os quais contêm as instruções necessárias para manipular o sistema e gerar situações de teste.

A estrutura da especificação é organizada em arquivos com extensão .feature, próprios do Cucumber. Esses arquivos descrevem conjuntos de cenários, compostos por passos que são interpretados e associados às definições implementadas em Python, por meio da biblioteca Behave. Cada cenário dentro da feature descreve, utilizando as palavras guias, os seguintes componentes do teste:

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

Dessa forma, um 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
```

Tanto este quanto o exemplo anterior ilustram como o uso da linguagem natural proposto pelo processo de BDD na escrita dos cenários torna o entendimento claro e evita confusões. Outro benefício dessa abordagem é sua independência em relação à tecnologia utilizada — ao definir que o comportamento esperado do sistema é travar o carro em resposta ao pressionar o botão, essa especificação permanece válida independentemente do tipo de trava adotado ou mesmo do número de portas no veículo.

Com a biblioteca Behave, do Python, é possível implementar os passos definidos nos cenários Gherkin por meio de funções associadas a decoradores específicos. Esses decoradores utilizam as palavras-chave da gramática Gherkin — como Given, When e Then — para vincular cada trecho do cenário a uma função que executa a lógica correspondente.

Assim como citado por (SOLIS; WANG, 2011) as definições de passo e integração com o software é feita de maneira iterativa, similar ao do processo de TDD, seguindo o ciclo “vermelho-verde-refatorar”. Nesse contexto, cada passo do cenário assume o papel de um teste: inicialmente, o passo falha (“vermelho”); em seguida, são feitas alterações mínimas para que ele seja satisfeito (“verde”); por fim, o código é aprimorado mantendo o comportamento esperado (“refatoração”).

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 comunicação entre o código Python e o modelo Simulink, além da identificação do bloco cujo valor deve ser alterado. 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; 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 para executar os testes que validam o seu comportamento durante as simulações.

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 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 será focado 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. Para isso, a definição do modelo do sistema e dos testes de aceitação tomará como base o teste de caixa preta (SOMMERVILLE, 2011).

Dessa forma, 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 que é responsável por converter esse sinal em um formato compatível com o componente físico selecionado.

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. 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.

REFERÊNCIAS

Atlassian. **Agile Teams**. n.d. Acesso em: 12 jul. 2025. Disponível em: <https://www.atlassian.com/agile/teams>. Citado 2 vezes nas páginas 15 e 18.

AUTOSAR. **Classic Platform**. n.d. Acesso em: 26 ago. 2025. Disponível em: <https://www.autosar.org/standards/classic-platform>. Citado 2 vezes nas páginas 16 e 24.

Behave. **Behave Documentation**. 2025. Acesso em: 12 jul. 2025. Disponível em: <https://behave.readthedocs.io/en/latest/>. Citado na página 21.

Bosch. **Automotive Handbook**. 9. ed. [S.l.: s.n.], 2022. Citado na página 15.

Cucumber.io. **Cucumber Documentation**. 2024. Acesso em: 12 jul. 2025. Disponível em: <https://cucumber.io/docs/>. Citado na página 21.

Cucumber.io. **History of BDD**. 2024. Acesso em: 12 jul. 2025. Disponível em: <https://cucumber.io/docs/bdd/history/>. Citado 2 vezes nas páginas 19 e 21.

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 16 e 20.

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 16 e 23.

NORTH, D. **Introducing Behavior-Driven Development**. 2006. Acesso em: 12 jul. 2025. Disponível em: <https://dannorth.net/introducing-bdd/>. Citado 3 vezes nas páginas 15, 18 e 19.

REHKOPF, M. **User Stories**. 2025. Acesso em: 12 jul. 2025. Disponível em: <https://www.atlassian.com/agile/project-management/user-stories>. Citado na página 19.

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 2 vezes nas páginas 16 e 23.

SOLIS, C.; WANG, X. **A Study of the Characteristics of Behaviour Driven Development**. 2011. 383–387 p. Doi: 10.1109/SEAA.2011.76. Citado 3 vezes nas páginas 16, 18 e 22.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011. Tradução: Ivan Bosnic e Kalinka Oliveira; Revisão técnica: Kechi Hirama. ISBN 978-85-7936-108-1. Citado na página 25.

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 16 e 20.

ANEXO A – COMANDOS SERIAIS DA ESTAÇÃO METEOROLÓGICA *VANTAGE VUE™*

Tabela 1 – Comandos seriais suportados pela estação meteorológica *Vantage Vue™*

Instrução	Descrição
Comandos de teste	
TESTE	Envia a *string* "TEST\n"de volta
WRD	Responde com o tipo de estação meteorológica
RXCHECK	Responde com o diagnóstico do Console
RXTEST	Muda a tela do console de *"Receiving from"* para tela de dados atuais
VER	Responde com a data do *firmware*
RECEIVERS	Responde com a lista das estações que o console "enxerga"
NVER	Responde com a versão do *firmware*
Comandos de dados atuais	
LOOP	Responde com a quantidade de pacotes especificada a cada 2s
LPS	Responde a cada 2s com a quantidade de pacotes diferentes especificada
HILOWS	Responde com todo os dados de *high/low*
PUTRAIN	Seta a quantidade anual de precipitação
PUTET	Seta a quantidade anual de evapotranspiração
Comandos de *download*	
DMP	Faz o *download* de todo o arquivo de memória
DMAFT	Faz o *download* de todo o arquivo de memória após a data especificada
Comandos da EEPROM	
GETEE	Lê toda a memória EEPROM
EEWR	Escreve um *byte* de dados à partir do endereço especificado
EERD	Lê a quantidade de dados especificada iniciando no endereço especificado
EEBWR	Escreve os dados na EEPROM
EEBRD	Lê os dados da EEPROM
Comandos de calibração	
CALED	Envia os dados da temperatura e umidade corrente para atribuir à calibração
CALFIX	Atualiza o *display* quando os números de calibração mudam
BAR	Seta os valores da elevação e o *offset* do barômetro quando a localização é alterada
BARDATA	Mostra os valores atuais da calibração do barômetro

Continua na próxima página

Tabela 1 – Continuação da página anterior

Instrução	Descrição
Comandos de limpeza	
CLRLOG	Limpa todo o arquivo de dados
CLRALM	Limpa todos os limiares dos alarmes
CLRCAL	Limpa todos os *offsets* da calibração da temperatura e da umidade
CLRGRA	Limpa o gráfico do console
CLRVAR	Limpa o valor da precipitação ou da evapotranspiração
CLRHIGHS	Limpa todos os valores de pico diários, mensais ou anuais
CLRLOWS	Limpa todos os valores de mínimos diários, mensais ou anuais
CLRBITS	Limpa os *bits* de alarme ativos
CLRDATA	Limpa todos os dados atuais
Comandos de configuração	
BAUD	Atribui o valor do *baudrate* do console
SETTIME	Define a data e a hora do console
GAIN	Define o ganho do receptor de rádio
GETTIME	Retorna a hora e a data atual do console
SETPER	Define o intervalo de arquivamento
STOP	Desabilita a criação dos registros
START	Habilita a criação dos arquivos
NEWSETUP	Reinicia o console após alguma configuração nova
LAMPS	Liga ou desliga as lâmpadas do console

Fonte: ??) (Traduzido).