Universidade Estadual da Paraíba (UEPB)
Centro de Ciências e Tecnologia (CCT)
Curso Bacharelado em Ciências da Computação

Pedro Lucas Vaz Andrade

Otavio Ferreira Clementino

Wagner Tibúrcio da Silva Junior

Relatório de Desenvolvimento

Campina Grande - 2025

1. Introdução

Este relatório apresenta as abordagens adotadas para a criação da primeira release do projeto **Marketplace**, desenvolvido no contexto da disciplina **Engenharia de Software 2**. O objetivo principal é a aplicação prática dos conceitos adquiridos em **Engenharia de Software 1**, estruturando um sistema funcional com base em boas práticas de desenvolvimento.

O projeto desenvolvido trata-se de um **Marketplace**, inspirado em plataformas como Amazon e Mercado Livre, onde há dois tipos principais de usuários: **lojas**, que vendem produtos, e **compradores**, que realizam compras dentro do sistema. Durante esta primeira release, foram desenvolvidos os principais componentes estruturais do sistema, garantindo as funcionalidades básicas de cadastro e gestão de entidades.

O sistema foi implementado utilizando a linguagem **Java**, com **Maven** como gerenciador de dependências e ferramenta de automação de compilação. Para a estruturação da aplicação, adotamos o padrão arquitetural **MVC (Model-View-Controller)**, proporcionando melhor organização e modularização do código, facilitando a manutenção e futuras expansões. Ademais, foi utilizado o padrão **Facade**, visando desacoplar a classe Main, permitindo uma separação mais eficiente entre a camada de visualização e a lógica de controle.

2. Arquitetura utilizada no Projeto

A arquitetura **MVC** foi empregada para estruturar o projeto:

- **Model**: Contém as classes que representam os dados da aplicação, como `Loja`, `Comprador` e `Produto`.
- **View**: Responsável pela interação com o usuário através de uma interface de comandos.
- **Controller**: Contém a lógica de manipulação dos dados e regras de negócio.

Adicionalmente, o padrão **Facade** foi utilizado para centralizar a lógica do sistema em uma classe intermediária (`MarketplaceFacade`), reduzindo a complexidade da `Main` e melhorando a modularização.

3. Desenvolvimento com o Azure-Devops

Para garantir uma organização eficiente do desenvolvimento do projeto, utilizamos o **Azure DevOps** como ferramenta principal de planejamento e acompanhamento das atividades.

Dentro do Azure DevOps, estruturamos o **Product Backlog**, onde foram definidos os **PBIs (Product Backlog Items)**, representando os principais pontos de desenvolvimento do sistema. Cada PBI foi detalhado e decomposto em **Tasks**, que foram distribuídas entre os integrantes da equipe.

Além disso, para cada **Task**, foram atribuídas informações essenciais para o gerenciamento, como:

- **Prioridade:** Definição do nível de importância da tarefa para a entrega da release.
- **Tempo Estimado:** Cálculo da quantidade de horas previstas para a conclusão da atividade.
- **Responsável:** Designação do integrante da equipe responsável por sua execução.

Esse modelo de organização permitiu acompanhar o progresso do desenvolvimento de maneira clara e objetiva, garantindo que todas as entregas fossem feitas dentro do prazo estabelecido para a **Release.**

3.1 Desenvolvimento com o Azure-Devops

Para organizar e acompanhar a implementação da Release 1 do sistema, utilizamos a plataforma **Azure DevOps**, que oferece uma estrutura robusta para a gestão de projetos ágeis, permitindo o acompanhamento detalhado de cada etapa do desenvolvimento. Através dessa ferramenta, foi possível estruturar e distribuir as tarefas de maneira estratégica, garantindo um fluxo de trabalho contínuo, colaborativo e eficiente entre os membros da equipe.

A Release 1 foi planejada em **duas sprints**, com duração equilibrada, visando facilitar a divisão das funcionalidades e promover entregas incrementais e iterativas. Cada sprint foi planejada com foco em atender requisitos específicos, com um escopo bem definido e alinhado aos objetivos do projeto. Essa abordagem permitiu uma melhor organização do trabalho, identificação de prioridades e acompanhamento preciso do progresso.

Sprint 1: Foco no Cadastro de Entidades Principais

Durante a primeira sprint, priorizamos a implementação das funcionalidades **essenciais** para o funcionamento básico do sistema, principalmente aquelas relacionadas ao cadastro e gerenciamento das principais entidades. As tarefas desenvolvidas foram:

- **PBI 01: CRUD de Lojas** – Implementação completa das funcionalidades de criação, leitura, atualização e exclusão de lojas, incluindo validações, persistência dos dados e interface amigável para o usuário.

- **PBI 02: CRUD de Compradores** – Desenvolvimento das operações de cadastro de compradores, permitindo seu gerenciamento dentro do sistema de forma segura e eficiente.

Essa sprint teve como responsável principal **Pedro**, que atuou como **Product Owner (PO)**, sendo o elo entre a equipe de desenvolvimento e os requisitos do produto. Ele ficou encarregado de organizar o backlog, validar as entregas e garantir que o time estivesse focado nas prioridades corretas.

Sprint 2: Expansão dos Módulos e Aprimoramento da Arquitetura

Na segunda sprint, avançamos para uma etapa mais técnica e estratégica do projeto, voltada à estruturação de novos módulos, refino da arquitetura e documentação da release. As atividades contempladas foram:

- **PBI 03: CRUD de Produtos** – Desenvolvimento completo das operações de cadastro de produtos, incluindo controle de atributos específicos, vinculação com lojas e compradores, e tratamento adequado de dados.

- **PBI 04: Documentação da Release 1** – Produção de uma documentação técnica detalhada, descrevendo a arquitetura, decisões de design, funcionalidades implementadas, padrões

utilizados e instruções para uso e manutenção do sistema.

- **PBI 05: Implementação do Padrão Facade** – Aplicação do padrão de projeto **Facade** para promover maior modularização do código, facilitando a manutenção e evolução do sistema. Essa atividade também envolveu a **implementação da persistência de arquivos**, garantindo que os dados sejam armazenados de forma permanente, mesmo após o encerramento da aplicação.

Nesta sprint, o papel de **Product Owner (PO)** foi assumido por **Wagner**, que coordenou a equipe na definição das prioridades, revisão das tarefas e validação dos resultados entregues.

Essa divisão das sprints permitiu um **desenvolvimento progressivo e bem estruturado**, possibilitando a entrega de uma primeira versão funcional e documentada do sistema. Com a decomposição dos **Product Backlog Items (PBIs)** em **tasks** menores, foi possível detalhar cada atividade, atribuir responsabilidades claras, definir prioridades e estimar com mais precisão o tempo de conclusão de cada item.

Cada **task** conta com uma descrição completa, o responsável pela execução, a **prioridade** definida com base no impacto e urgência da funcionalidade, além do **tempo estimado para conclusão**, o que permite um acompanhamento constante do andamento do projeto, facilitando a identificação de gargalos e a reavaliação de prazos sempre que necessário.

A utilização do Azure DevOps como ferramenta de apoio foi essencial para garantir a **transparência**, **colaboração** e **organização** do time durante o processo de desenvolvimento da Release 1.

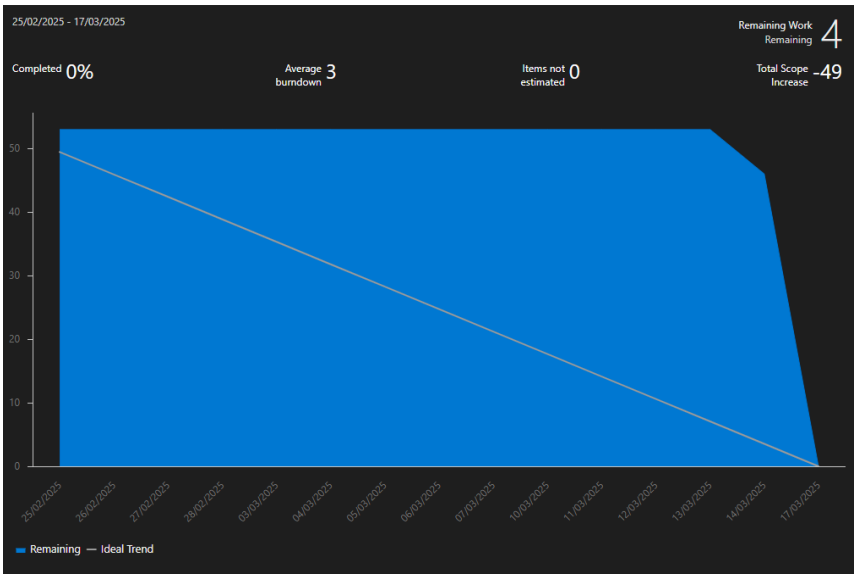

Grafico Burndown 1 Sprint

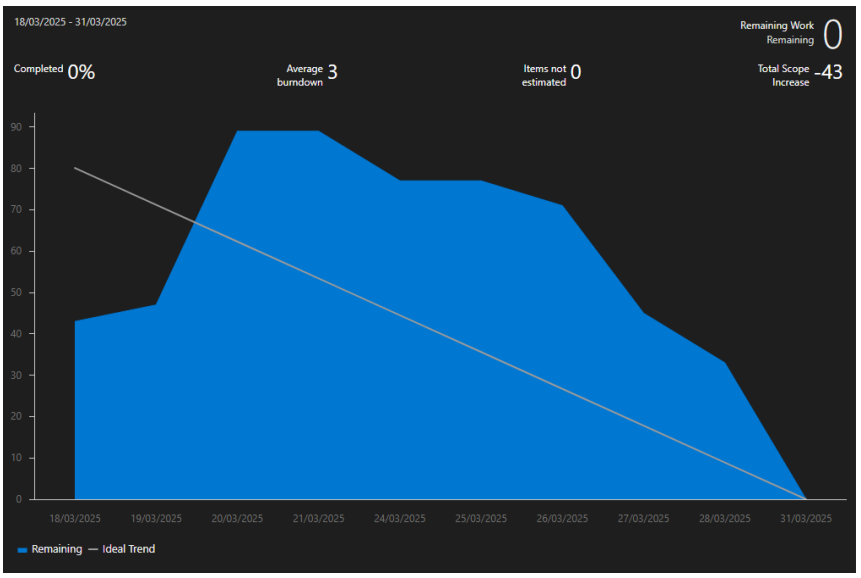

Gráfico Burndown 2 Sprint

4.0 PBI 01 - CRUD de Lojas

Implementação das Operações de Cadastro, Atualização, Exclusão e Listagem de Lojas

O PBI busca implementar operações essenciais para o gerenciamento de lojas no sistema. Cada tarefa tem sua importância para garantir que as lojas sejam manipuladas de forma eficiente.

T1.1: Criar a Classe Loja com Atributos (Pedro - 2h)

Pedro criará a classe Loja, que incluirá atributos essenciais como nome, email, senha, CPF/CNPJ e endereço. A definição correta dessa classe é fundamental para o funcionamento do sistema, pois ela serve como base para todas as operações posteriores.

T1.2: Implementar Métodos de CRUD na Classe Loja (Pedro - 4h)

Pedro implementará os métodos de CRUD (Create, Read, Update e Delete) na classe Loja, permitindo realizar as operações de inserção, consulta, atualização e remoção das lojas. Essa etapa é essencial para garantir a interação com os dados da loja.

T1.3: Criar Persistência das Lojas em Arquivo (Tiago - 4h)

Tiago será responsável por garantir que as lojas sejam salvas e recuperadas de um arquivo. Isso é crucial para que os dados das lojas não sejam perdidos entre as sessões do sistema.

T1.4: Implementar Testes Unitários para CRUD de Lojas (Otávio - 4h)

Otávio criará testes unitários para garantir que os métodos de CRUD funcionem corretamente. Os testes são importantes para detectar falhas rapidamente e garantir a qualidade do código.

T1.5: Criar Interface de Comandos para Manipulação de Lojas (Wagner - 4h)

Wagner criará uma interface de comandos para interação com o sistema, permitindo aos usuários realizar operações de CRUD de forma simples. Essa interface facilita o uso do sistema e torna a experiência do usuário mais intuitiva.

4.1 PBI 02 - CRUD de Compradores

Este PBI visa implementar as funcionalidades de gestão de compradores, permitindo o cadastro, atualização, exclusão e listagem de compradores na plataforma.

T2.1: Criar a Classe Comprador com Atributos (Pedro - 2h)

Pedro criará a classe Comprador, que incluirá atributos como nome, email, senha, CPF e endereço. Essa definição é essencial para armazenar os dados do comprador de maneira estruturada.

T2.2: Implementar Métodos de CRUD na Classe Comprador (Pedro - 4h)

Pedro implementará os métodos de CRUD (Create, Read, Update, Delete) para permitir o gerenciamento completo dos compradores na plataforma, permitindo que os dados sejam inseridos, consultados, alterados e removidos.

T2.3: Criar Persistência dos Compradores em Arquivo (Tiago - 4h)

Tiago será responsável por garantir que os dados dos compradores sejam armazenados em um arquivo. Isso é necessário para manter os registros entre as sessões do sistema.

T2.4: Implementar Testes Unitários para CRUD de Compradores (Otávio - 4h)

Otávio criará testes unitários para verificar se os métodos de CRUD funcionam corretamente. A implementação desses testes é importante para garantir a qualidade e a confiabilidade do sistema.

T2.5: Criar Interface de Comandos para Manipulação de Compradores (Wagner - 4h)

Wagner criará uma interface de comandos para facilitar a interação dos usuários com o sistema, permitindo o gerenciamento dos compradores de maneira intuitiva e eficiente.

4.2 PBI 03 - CRUD de Produtos

Este PBI visa implementar as operações de cadastro e gestão de produtos dentro das lojas da plataforma.

T3.1: Criar a Classe Produto e Definir Atributos (Pedro - 3h)

Pedro criará a classe Produto, incluindo atributos essenciais como nome, valor, tipo, quantidade, marca e descrição. A definição dessa classe é crucial para armazenar os dados dos produtos de maneira estruturada e acessível.

T3.2: Criar Relacionamento entre Loja e Produto (Tiago - 3h)

Tiago estabelecerá o relacionamento entre as classes Loja e Produto, garantindo que cada loja possa ter múltiplos produtos. Esse relacionamento é necessário para associar produtos às lojas de forma eficiente.

T3.3: Implementar Métodos de CRUD para Produtos (Pedro - 4h)

Pedro implementará os métodos de CRUD (Create, Read, Update, Delete) para os produtos, permitindo que o sistema gerencie os produtos de forma eficaz, com operações para criar, consultar, atualizar e excluir produtos.

T3.4: Criar Persistência dos Produtos em Arquivo (Tiago - 4h)

Tiago será responsável por garantir que os produtos sejam salvos e carregados de um arquivo. Isso garante que os dados dos produtos persistam entre as execuções do sistema.

T3.5: Criar Testes Unitários para CRUD de Produtos (Otávio - 4h)

Otávio criará os testes unitários para validar o funcionamento dos métodos de CRUD de produtos, assegurando que o sistema esteja operando conforme o esperado e sem erros.

T3.6: Criar Interface de Comandos para Manipulação de Produtos (Wagner - 4h)

Wagner criará uma interface de comandos que permitirá aos usuários interagir com o sistema de forma intuitiva, realizando as operações de CRUD para produtos de maneira simples e eficiente.

T3.7: Implementar o Padrão Facade (Wagner - 4h)

Foi adicionada uma nova tarefa para implementar o **Facade**. Pedro criará uma camada de fachada que irá simplificar a interação com as classes do sistema, facilitando as operações de CRUD de produtos sem expor a complexidade interna.

4.3 PBI 04 - Repositório Git

Este PBI visa estruturar o repositório do projeto e definir um fluxo de trabalho para o desenvolvimento colaborativo.

T4.1: Criar Repositório no GitHub e Estruturar Diretórios (Pedro - 3h)

Pedro criará o repositório no GitHub e estruturará os diretórios principais do projeto, como `src/`, `test/`, `docs/` e `releases/`. A estruturação do repositório é essencial para organizar o código e os recursos do projeto de forma eficiente.

T4.2: Estruturar o Projeto em Arquitetura MVC (Pedro - 4h)

Pedro será responsável por estruturar o projeto em **MVC**, separando a lógica de controle, visualização e dados. Isso proporcionará maior escalabilidade e manutenção do sistema.

5.0 Processo de Desenvolvimento do Marketplace

O desenvolvimento do Marketplace foi estruturado seguindo a arquitetura MVC (Model-View-Controller), com uso do padrão Facade para centralizar a lógica de interação entre os componentes. A implementação foi realizada em Java, com o Maven como gerenciador de dependências. O sistema foi pensado para simular um ambiente de compra e venda semelhante a plataformas como Amazon e Mercado Livre, com foco inicial no gerenciamento de entidades fundamentais como lojas, produtos e compradores.

Organização das Classes

O projeto foi dividido em pacotes organizados de acordo com suas responsabilidades:

- `model`: contém as classes que representam as entidades do sistema (como `Loja`, `Produto`, `Comprador`).
- `repository`: classes responsáveis pela persistência de dados, utilizando arquivos `.txt` para simular um banco de dados.
- `controller`: intermedia as ações entre a `view` e os dados.
- `view`: onde está a interface textual com o usuário.
- `fachade`: centraliza as operações do sistema, funcionando como uma fachada única para encapsular a lógica de negócios e comunicação entre os módulos.

Essa divisão ajuda a manter o código coeso e com responsabilidades bem definidas.

Camada Model

A camada de modelo contém a lógica de representação das entidades. Cada classe define atributos, construtores e métodos getters e setters. Por exemplo, a classe `Produto` representa um item à venda no marketplace:

```
public class Produto {

    private String nome;

    private double valor;

    private String tipo;

    private int quantidade;

    private String marca;

    private String descricao;

    private Loja loja;

    // Construtores, getters e setters...

}
```

O relacionamento entre produto e loja foi implementado diretamente por meio do atributo `Loja loja`, permitindo vincular o produto à loja de origem.

Camada Repository

A camada de repositório foi desenvolvida para fazer a persistência de dados em arquivos de texto. Por exemplo, o `ProdutoRepository` salva os dados de produtos no arquivo `produtos.txt`, utilizando a serialização de dados com delimitadores:

```
public void salvar(Produto produto) {
    List<Produto> produtos = listar();
    produtos.add(produto);
    salvarArquivo(produtos);
}
private void salvarArquivo(List<Produto> produtos) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(FILE_NAME))) {
        for (Produto produto : produtos) {
            bw.write(produto.getNome() + ";" + produto.getValor() + ";" + produto.getTipo() + ";" + ...);
            bw.newLine();
        }
    } catch (IOException e) {
        System.out.println("Erro ao salvar produtos: " + e.getMessage());
    }
}
```

Esse padrão se repetiu nos repositórios de lojas e compradores.

Camada View

A view foi desenvolvida com menus textuais simples, utilizando o `Scanner` para interagir com o usuário. Essa abordagem facilitou os testes e a navegação entre as funcionalidades.

```
System.out.println("1 - Cadastrar Produto");
System.out.println("2 - Listar Produtos");
int opcao = scanner.nextInt();
```

Cada ação do menu chama métodos do `Controller` ou diretamente da `Fachada`, dependendo da abstração.

Camada Controller

Os controllers foram responsáveis por coordenar as regras de negócio e chamar os métodos do repositório e da fachada. Por exemplo, ao cadastrar um produto, o `ProdutoController` faz a ligação entre a entrada do usuário e a chamada à camada de persistência:

```
public class ProdutoController {

    private ProdutoRepository produtoRepo = new ProdutoRepository();



    public void cadastrarProduto(Produto produto) {

        produtoRepo.salvar(produto);

    }

}
```

Essa estrutura promoveu uma separação clara entre a interface, a lógica e os dados.

Camada Facade

A classe `Fachada` foi um dos principais elementos para centralizar a lógica do sistema. Ela expõe métodos de alto nível que encapsulam toda a lógica de cadastro, busca e remoção de entidades, sem que a camada de visualização precise conhecer detalhes de implementação.

```
public class Fachada {

    private ProdutoController produtoController = new ProdutoController();

    private LojaController lojaController = new LojaController();

    public void cadastrarProduto(String nome, double valor, Loja loja, ...) {

        Produto produto = new Produto(nome, valor, ..., loja);

        produtoController.cadastrarProduto(produto);

    }

}
```

Com isso, o `main` ou a `view` interage apenas com a `Fachada`, tornando o sistema mais coeso e desacoplado.

Relacionamento Produto/Loja

Foi implementado o vínculo direto entre o produto e a loja por meio do atributo `loja` na classe `Produto`. Ao cadastrar um produto, a loja é passada como argumento e salva junto aos dados do produto. Assim, ao consultar os produtos de uma loja, é possível filtrar todos os produtos que pertencem a ela.

```
public List<Produto> listarProdutosDaLoja(String nomeLoja) {

   List<Produto> todos = listar();

   return todos.stream()

      .filter(p -> p.getLoja().getNome().equals(nomeLoja))

      .collect(Collectors.toList());

}
```

Essa função permite que cada loja visualize apenas os produtos cadastrados por ela, promovendo maior organização e controle de inventário.

O processo de desenvolvimento foi realizado de forma incremental, com definição clara de PBIs e divisão em sprints. Cada camada foi projetada com base nos princípios de responsabilidade única e baixo acoplamento, utilizando padrões como MVC e Facade para garantir uma estrutura robusta e de fácil manutenção. A modelagem de dados e o uso de arquivos para simular persistência permitiram validar as principais operações do sistema, preparando o ambiente para expansões futuras como login de usuários, sistema de pedidos e integração com banco de dados.

6.0 Análise de Usabilidade do Projeto

O diagrama de atividades ilustra o fluxo de ações dentro do **Marketplace**, representando os caminhos possíveis tanto para **lojas** quanto para **compradores**.

O processo inicia com o **acesso ao sistema**, onde o usuário deve se autenticar. Caso seja uma **loja**, ele poderá **gerenciar produtos** (adicionar, editar e remover) e consultar o **histórico de vendas**. Se for um **comprador**, ele pode **navegar pelos produtos**, adicionar itens ao **carrinho de compras** e decidir se deseja **finalizar a compra**.

Se optar por concluir a compra, o usuário realiza o **pagamento**, recebe uma **confirmação do pedido** e pode acompanhar suas compras no **histórico de pedidos**. Caso contrário, ele pode continuar navegando pelo marketplace.

Este diagrama demonstra a fluidez do sistema e como a experiência do usuário é estruturada, garantindo que as interações sejam intuitivas e bem definidas.

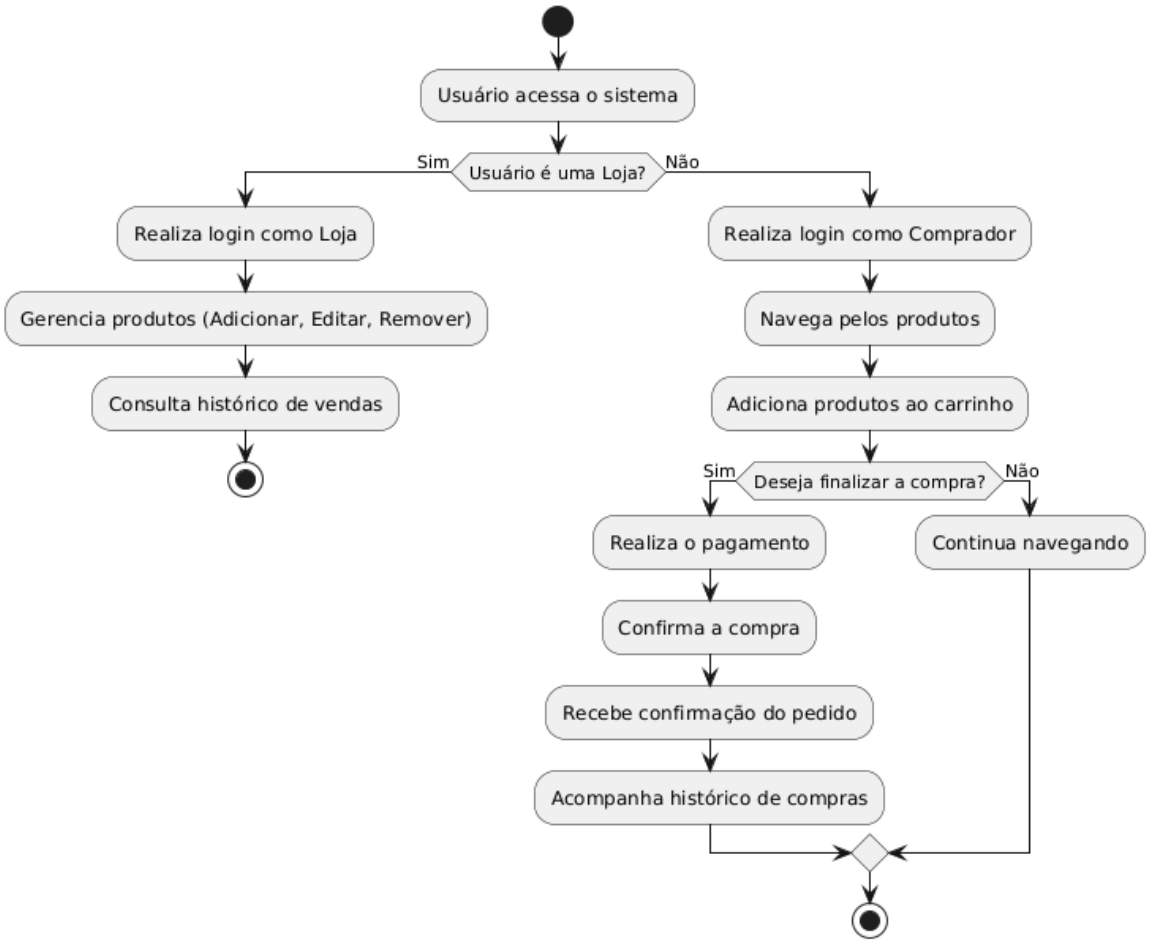

Figura 1.0 - Diagrama de Atividades

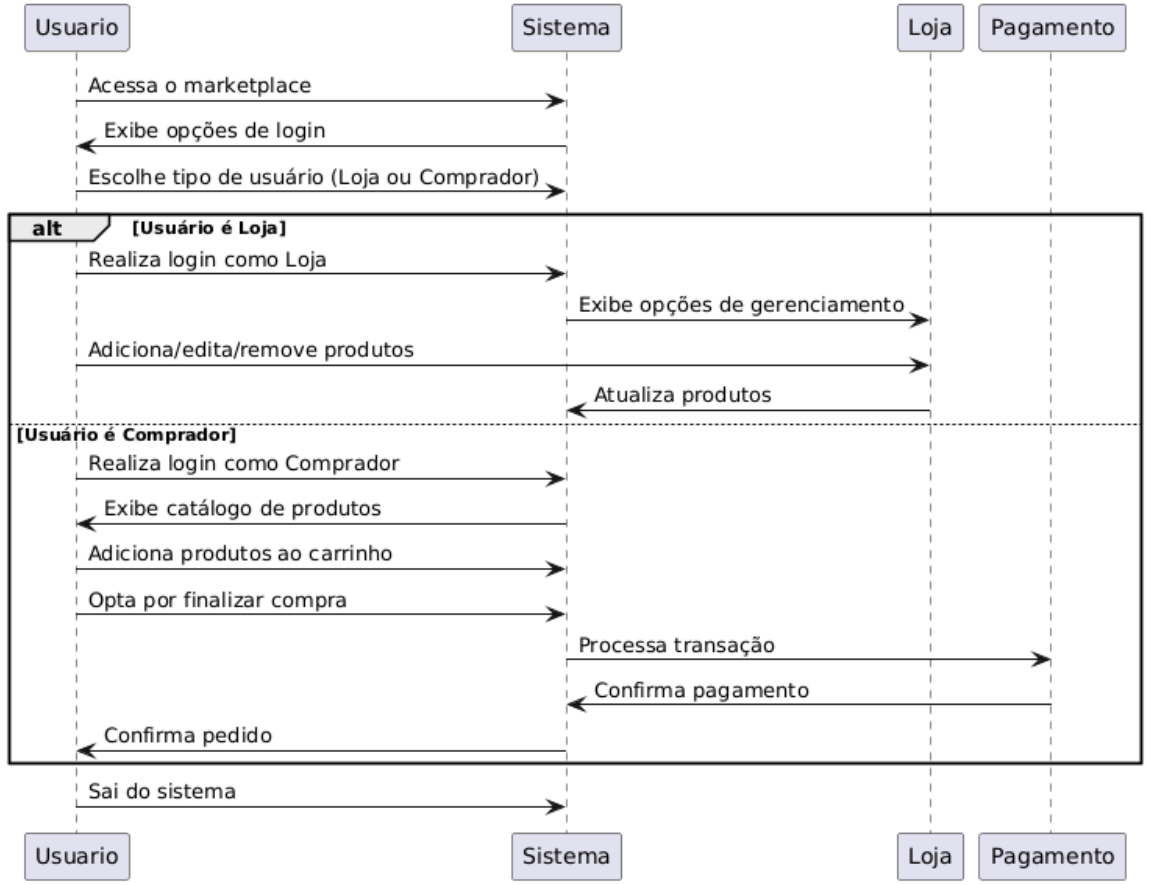

Figura 1.1 Diagrama de Sequência

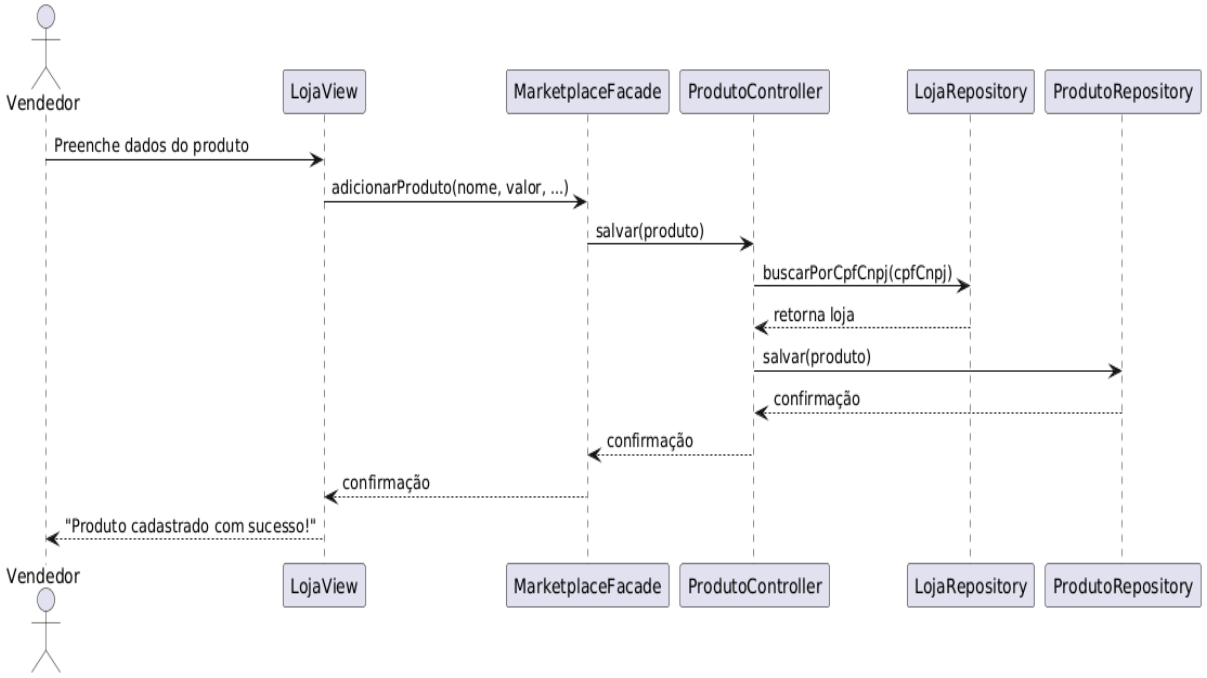

Figura 1.2 Diagrama de Sequência

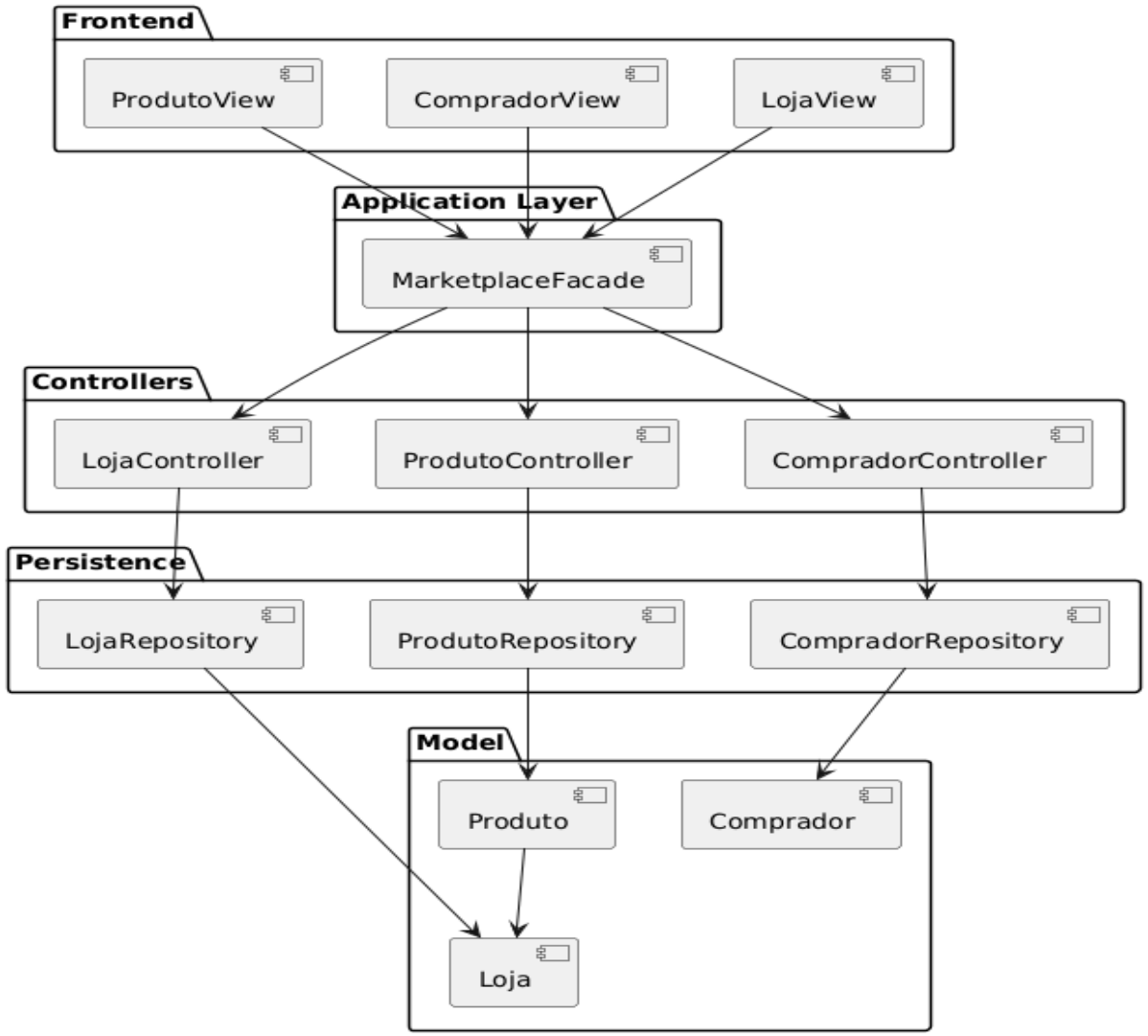

Figura 1.3 Diagrama de Conteinêrs

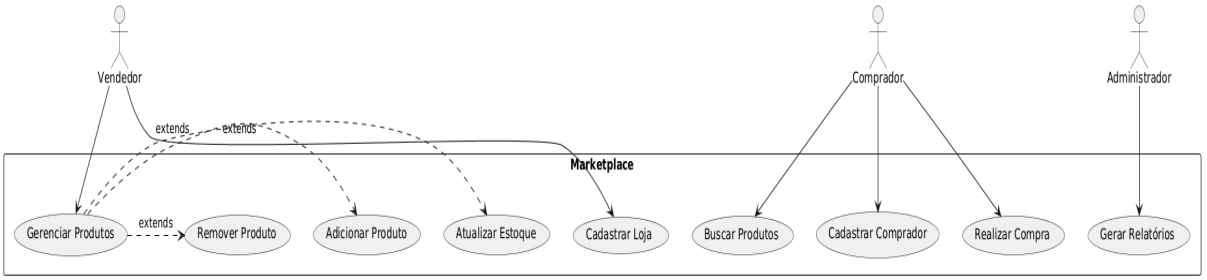

Figura 1.4 Diagrama de caso de uso

7 . Testes Realizados

Durante o desenvolvimento da primeira release do sistema *Marketplace*, foi conduzida uma série de **testes manuais** com o objetivo de validar as funcionalidades principais implementadas nas Sprints 1 e 2. A estratégia de testes foi orientada à verificação da **integração entre os componentes** e ao correto funcionamento de cada camada da arquitetura do sistema.

Embora nesta etapa inicial não tenham sido aplicados testes automatizados, as validações realizadas abrangeram **diversos cenários práticos**, simulando o uso real da aplicação. Os testes cobriram as quatro principais camadas: **Model**, **Repository**, **Controller** e **View**. A seguir, detalhamos os principais pontos testados em cada uma delas:

7.1 Camada Model (Entidades do Domínio)

Nesta camada, os testes focaram na integridade estrutural e comportamental das entidades centrais do sistema: *Loja*, *Comprador* e *Produto*. Foram realizados testes como:

- Verificação da correta criação de objetos com todos os atributos obrigatórios preenchidos.
- Avaliação do funcionamento dos métodos de acesso (`getters` e `setters`), bem como de métodos auxiliares implementados.
- Testes de associações entre objetos, como o vínculo entre *Produto* e *Loja*.
- Análise da serialização e desserialização das entidades, garantindo que os dados fossem preservados após serem gravados e lidos dos arquivos.

Esses testes garantiram que a **base de dados lógica** estivesse consistente, permitindo seu uso seguro pelas demais camadas.

7.2 Camada Repository (Persistência de Dados)

A camada de repositórios foi intensamente testada, visto que ela é responsável por garantir a **manutenção dos dados em arquivos `.txt`**. Entre os testes realizados, destacam-se:

- Testes de gravação de dados após cada operação de cadastro, atualização ou exclusão.
- Verificação da leitura dos dados ao iniciar a aplicação, garantindo recuperação íntegra das informações.
- Análise do conteúdo dos arquivos para checar a estrutura de escrita e possíveis duplicações.
- Testes com ausência de arquivos ou arquivos mal formatados, simulando falhas na persistência, para validar o tratamento de exceções.
- Verificação da integridade das associações entre entidades gravadas (por exemplo, produtos corretamente vinculados a lojas).

Esses testes asseguraram a **robustez do sistema frente a operações frequentes e variações no ambiente de execução**.

7.3 Camada Controller (Lógica de Controle e Regras de Negócio)

A camada de controle, composta por controllers específicos e pela implementação do padrão **Facade**, foi avaliada em relação ao seu papel de coordenar a lógica de negócio e interligar os demais componentes. Foram testados:

- Fluxos completos de cadastro, listagem e remoção de entidades, via chamadas ao Facade.
- Validação da entrada de dados, impedindo operações com valores nulos ou inválidos.
- Encadeamento correto de métodos e reutilização de lógica compartilhada entre funcionalidades.
- Verificação da separação de responsabilidades: a lógica de negócio está concentrada nos controllers e não na view.
- Comportamento esperado em situações excepcionais, como remoção de itens inexistentes ou tentativas de acessar entidades não cadastradas.

Através desses testes, foi possível assegurar que o sistema **executa suas operações de forma previsível, modularizada e reutilizável**, alinhado aos princípios de boas práticas de arquitetura.

7.4 Camada View (Interface com o Usuário)

A camada de interface foi avaliada a partir da perspectiva do **usuário final**, garantindo que a experiência de interação fosse clara, fluida e sem erros. Os testes incluíram:

- Navegação entre menus e seleção de opções, com mensagens de orientação e confirmação.
- Apresentação organizada e legível dos dados listados (Lojas, Compradores, Produtos).
- Respostas adequadas a entradas inválidas, como letras em campos numéricos, opções inexistentes ou comandos mal formatados.
- Testes de usabilidade e feedback: clareza nas mensagens de erro e nas instruções exibidas.
- Retorno apropriado após operações bem-sucedidas, como remoções ou cadastros.

Com esses testes, garantiu-se que o sistema **fornecesse uma experiência de uso compreensível, funcional e intuitiva**, mesmo para usuários não técnicos.

8. Abrangência dos Testes e Considerações Finais

A primeira release do sistema foi testada de forma **abrangente e multidimensional**, com foco nos seguintes aspectos:

- **Cenários de sucesso**, com operações realizadas corretamente.
- **Cenários de falha**, com entrada de dados inválidos, inconsistências ou tentativas de acessar dados inexistentes.
- **Cenários de uso contínuo**, validando a resistência do sistema em operações repetidas.
- **Testes de integração entre camadas**, do input do usuário até a persistência no arquivo.

Além disso, a implementação do **padrão Facade** foi fundamental para unificar os acessos e simplificar os testes. Sua adoção facilitou o controle de chamadas e a organização das funcionalidades em módulos independentes e coesos.

Apesar da ausência de testes automatizados nesta etapa, a abordagem manual permitiu identificar e corrigir erros de lógica, estrutura e integração de forma eficaz. Para as próximas fases do projeto, está previsto:

- **Adoção de testes automatizados com JUnit**, principalmente para validar regras de negócio e persistência.
- **Implementação de testes de regressão**, evitando a quebra de funcionalidades em futuras releases.
- **Utilização de ferramentas de cobertura de testes**, identificando partes do sistema que ainda não foram testadas.
- **Inclusão de testes de usabilidade com usuários reais**, aprimorando a interface e a experiência geral de uso.

Assim, a Release 1 se encerra com um sistema **estável, funcional e preparado para expansão**, sustentado por uma base sólida de testes manuais criteriosos.