UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIAS
CURSO DE SISTEMAS DE INFORMAÇÃO

CHRISTIAN SCHNEIDER, DIEGO ROCKENBACH, DIOGO ANTONIO, GUSTAVO TEIXEIRA E LUCAS AUED.

RELATÓRIO: PROJETO INTEGRADOR FINAL 1 – ANÁLISE E PROJETO ORIENTADO A OBJETOS

Santa Maria - RS
2025

1. Introdução

Este projeto tem como objetivo desenvolver um sistema orientado a objetos para gerenciamento de estoque, aplicando práticas de modelagem UML, princípios de design e padrões de projeto estudados na disciplina. O sistema permite o cadastro, listagem e manipulação de produtos, bem como o monitoramento de estoque crítico por meio de observadores. Além disso, incorpora diferentes políticas de reposição automática utilizando o padrão Strategy.

A proposta enfatiza a criação de uma arquitetura modular, extensível e de baixo acoplamento, demonstrando a importância dos padrões de projeto na organização e manutenção de sistemas reais.

2. Análise e modelagem

2.1 Lista de User Stories:

- **US01 – Cadastrar produto**
  Como gerente de estoque, quero cadastrar produtos no sistema para manter o controle de itens disponíveis.
- **US02 – Registrar entrada de produtos**
  Como funcionário do estoque, quero registrar a entrada de novas unidades para atualizar a quantidade disponível.
- **US03 – Registrar saída de produtos**
  Como funcionário do estoque, quero registrar saídas para refletir o consumo ou venda de itens.
- **US04 – Listar todos os produtos**
  Como usuário, quero visualizar a lista completa de produtos para acompanhar suas quantidades e informações gerais.
- **US05 – Receber alertas de estoque baixo**
  Como gerente, quero ser notificado quando um produto estiver com quantidade baixa para tomar providências.
- **US06 – Definir política de reposição automática**
  Como administrador, quero escolher uma estratégia de reposição de estoque para receber sugestões automáticas de compra.

2.2 Diagrama de Classes

2.3 Diagrama de Sequência

2.4 Justificativa dos padrões aplicados

O projeto utiliza cinco padrões de projeto que se complementam para construir uma arquitetura robusta: Factory, Singleton, Facade, Observer e Strategy. A justificativa detalhada está descrita na seção de Implementação a seguir.

3. Implementação

3.1 Factory:

A criação de ProdutoEletronico, ProdutoAlimento e ProdutoGenerico ficou concentrada em uma única classe, evitando new espalhado pelo código cliente. O Main apenas chama ProdutoFactory.criarProduto(), ficando independentes de qual subclasse concreta é usada.

Para acrescentar um novo tipo de produto (por exemplo, ProdutoLimpeza), basta criar a subclasse e ajustar a lógica na fábrica, mantendo o restante do sistema intacto (princípio Open/Closed).

```java
public static Produto criarProduto(String nome, String codigo, int quantidade,
                                   Categoria categoria, Fornecedor fornecedor) {

    String nomeCategoria = categoria.getNome();

    // Lógica de decisão: qual tipo de produto criar?
    if (nomeCategoria.equalsIgnoreCase(CATEGORIA_ELETRONICO)) {
        System.out.println("[Factory] Criando produto eletrônico com garantia de " + GARANTIA_PADRAO_MESES + " meses...");
        return new ProdutoEletronico(nome, codigo, quantidade, categoria, fornecedor, GARANTIA_PADRAO_MESES);

    } else if (nomeCategoria.equalsIgnoreCase(CATEGORIA_ALIMENTO)) {
        LocalDate dataValidade = LocalDate.now().plusDays(VALIDADE_PADRAO_DIAS);
        System.out.println("[Factory] Criando produto alimentício com validade até " + dataValidade + "...");
        return new ProdutoAlimento(nome, codigo, quantidade, categoria, fornecedor, dataValidade);

    } else {
        System.out.println("[Factory] Criando produto genérico...");
        return new ProdutoGenerico(nome, codigo, quantidade, categoria, fornecedor);
    }
}
```

3.2 Facade:

A fachada esconde detalhes como Estoque, ProdutoFactory e o mecanismo de observadores.

Se no futuro o sistema tiver uma interface gráfica ou uma API REST, esses clientes podem falar apenas com a SistemaEstoqueFacade, reduzindo o acoplamento.

```java
public class SistemaEstoqueFacade {

    private Estoque estoque;

    public SistemaEstoqueFacade() {
        this.estoque = Estoque.getInstance();
    }

    public Categoria criarCategoria(String nome) {
        return new Categoria(nome);
    }

    public Fornecedor criarFornecedor(String nome) {
        return new Fornecedor(nome);
    }

    public Produto criarProduto(String nome, String codigo, int quantidade, Categoria categoria, Fornecedor fornecedor) {
        return ProdutoFactory.criarProduto(nome, codigo, quantidade, categoria, fornecedor);
    }
```

3.3 Singleton:

A classe Estoque, implementa o padrão Singleton para garantir uma única instância de estoque em toda a aplicação.

O estoque é um recurso global do sistema: todas as partes (menu principal, fachada, alertas) precisam enxergar o mesmo conjunto de produtos. Usando Estoque.getInstance(), garantimos que não existam dois estoques diferentes com quantidades divergentes. Como consequência, qualquer alteração de quantidade ou cadastro feita em um ponto da aplicação é imediatamente visível em todos os outros.

```java
public class Estoque {
    private static Estoque instance;
    private List<Produto> produtos;
    private List<AlertaEstoque> observadores;
    private EstrategiaReposicao estrategiaReposicao;

    private Estoque() {
        produtos = new ArrayList<>();
        observadores = new ArrayList<>();
    }

    public static Estoque getInstance() {
        if (instance == null) {
            instance = new Estoque();
        }
        return instance;
    }
```

3.4 Strategy:

A lógica de “quando e quanto repor” é um ponto que pode mudar ao longo do tempo ou variar de cliente para cliente. Em vez de colocar if/else dentro de Estoque, extraímos essas regras para objetos separados (ReposicaoConservadoraStrategy, ReposicaoAgressivaStrategy).

Estoque, por sua vez, acaba apenas por chamar estrategiaReposicao.calcularQuantidadeReposicao(p), mantendo-se estável mesmo que novas políticas sejam adicionadas. O Main consegue trocar a política em uma única linha, se necessário.

```java
public class ReposicaoConservadoraStrategy implements EstrategiaReposicao {
    private final int limiteMinimo;
    private final int loteReposicao;

    public ReposicaoConservadoraStrategy() {
        this(5, 10);
    }

    public ReposicaoConservadoraStrategy(int var1, int var2) {
        this.limiteMinimo = var1;
        this.loteReposicao = var2;
    }

    public int calcularQuantidadeReposicao(Produto var1) {
        return var1.getQuantidade() < this.limiteMinimo ? this.loteReposicao : 0;
    }
}
```

3.5 Observer:

Para notificar automaticamente quando há saída de produtos e o estoque fica crítico, usamos o padrão Observer.

A responsabilidade de “enviar alertas” é separada da lógica de controle de estoque. Estoque só sabe que existe algo que implementa AlertaEstoque; não depende de detalhes de envio de e-mail, SMS etc.

Isso permite adicionar novos observadores sem alterar Estoque, basta registrá-los com addObservador.

```java
public class AlertaEmail implements AlertaEstoque {
    @Override
    public void atualizar(Produto p) {
        if (p.getQuantidade() < 10) {
            System.out.println("ALERTA: Produto " + p.getNome() + " com estoque baixo! Quantidade atual: " + p.getQuantidade());
        }
    }
}
```

4. Verificação de código

5. Contribuições de IA Generativa

As ferramentas de IA Generativa foram utilizadas para apoiar a construção da arquitetura do sistema, especialmente na documentação e na compreensão dos padrões de projeto. A utilização da IA ajudou a:

- explicar conceitos de forma clara e gerar exemplos coerentes;
- identificar melhorias e sugerir estruturas de código seguindo boas práticas de arquitetura;
- redigir partes do relatório de maneira organizada e padronizada.

Todas as sugestões oferecidas pela IA foram validadas manualmente, por meio da análise do código, da revisão das estruturas propostas e da verificação da coerência com os requisitos da disciplina. A implementação final foi inteiramente revisada pela equipe, garantindo autoria e compreensão completa do trabalho.

6. Conclusão

A aplicação dos padrões de projeto neste trabalho contribuiu significativamente para a organização, clareza e extensibilidade do sistema. O uso de padrões comportamentais, estruturais e de criação permitiu reduzir acoplamento, aumentar a reutilização de código e facilitar a evolução do projeto.

O padrão Factory simplificou a criação de objetos, enquanto Singleton garantiu integridade no gerenciamento global do estoque. A Facade forneceu uma interface unificada e limpa, o Observer introduziu um mecanismo automático de notificação e o Strategy permitiu adaptar comportamentos do sistema sem modificações internas.

Esses padrões, combinados, resultaram em um sistema modular, escalável e de fácil manutenção, características essenciais para sistemas reais e para o desenvolvimento profissional em orientação a objetos.