

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.

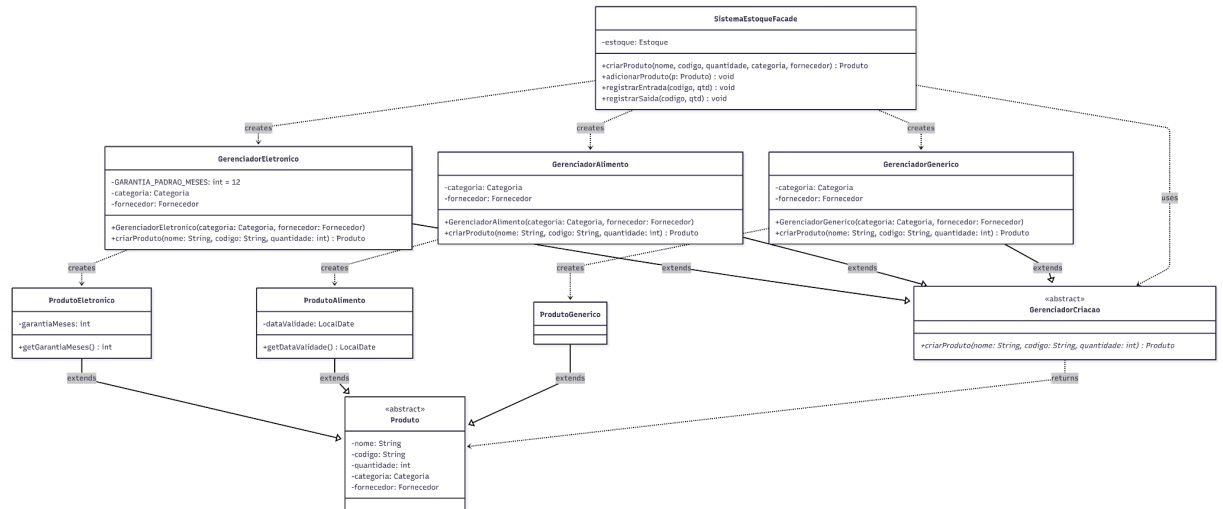
A proposta enfatiza a criação de uma arquitetura modular, extensível e de baixo acoplamento, demonstrando a importância dos padrões de projeto na organização e manutenção de sistemas reais.

2. Análise e modelagem

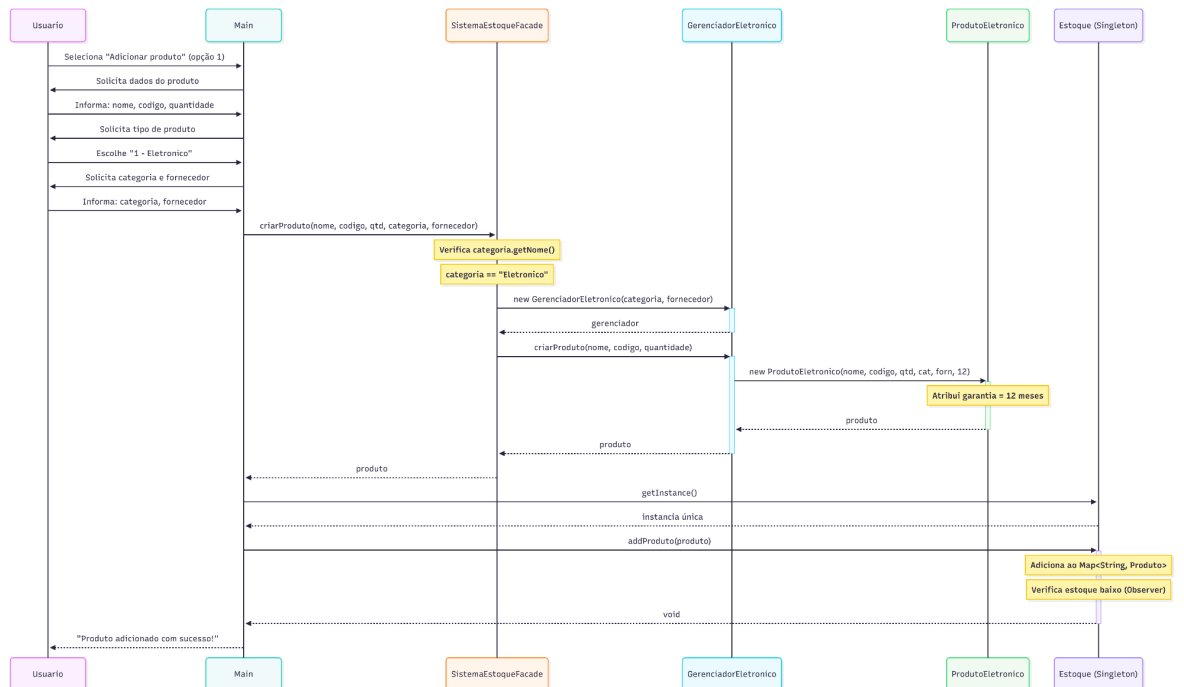
2.1 Lista de User Stories:

- **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 Padrão Factory Method (Criação)

A implementação da criação de produtos utiliza o padrão **Factory Method** clássico (GoF), substituindo a abordagem anterior de uma fábrica estática com condicionais.

Foi desenvolvida uma hierarquia de classes criadoras (Creators) para encapsular a lógica de instanciação.

- **Estrutura:** A classe abstrata `GerenciadorCriacao` define o contrato (método abstrato `criarProduto`).
- **Polimorfismo:** As subclasses concretas (`GerenciadorEletronico`, `GerenciadorAlimento` e `GerenciadorGenerico`) implementam esse método. Cada uma é responsável por instanciar seu produto específico e aplicar regras de negócio iniciais, como definir a garantia padrão de 12 meses para eletrônicos ou calcular a data de validade para alimentos.
- **Benefício:** O código cliente não depende mais de classes concretas de produtos, mas sim da abstração do gerenciador. Isso atende plenamente ao princípio **Open/Closed** (SOLID): para adicionar um novo tipo de produto, basta criar um novo Gerenciador, sem necessidade de alterar o código existente.

```
package factory;

import model.Categoria;
import model.Fornecedor;
import model.Produto;
import model.ProdutoEletronico;

public class GerenciadorEletronico extends GerenciadorCriacao {

    private static final int GARANTIA_PADRAO_MESES = 12;
    private Categoria categoria;
    private Fornecedor fornecedor;

    public GerenciadorEletronico(Categoria categoria, Fornecedor fornecedor) {
        this.categoria = categoria;
        this.fornecedor = fornecedor;
    }

    @Override
    public Produto criarProduto(String nome, String codigo, int quantidade) {
        return new ProdutoEletronico(
            nome,
            codigo,
            quantidade,
            categoria,
            fornecedor,
            GARANTIA_PADRAO_MESES
        );
    }
}
```

3.2 Padrão Facade (Estrutural)

A classe SistemaEstoqueFacade atua como uma fachada unificadora que simplifica a complexidade do subsistema, servindo como ponto único de entrada para a camada de apresentação (Main).

- **Orquestração:** Com a adoção do Factory Method, a responsabilidade de escolher *qual* criador usar foi delegada ao Facade. Baseado na categoria informada pelo usuário, o Facade instancia o GerenciadorCriacao apropriado (ex: GerenciadorEletronico) e solicita a criação do produto.
- **Desacoplamento:** A fachada esconde do cliente (Main) os detalhes complexos, como a hierarquia de factories, a instância única do Singleton Estoque e o registro de Observadores.
- **Evolução:** Se futuramente o sistema receber uma interface gráfica ou uma API REST, esses novos clientes interagirão apenas com o SistemaEstoqueFacade, garantindo que as regras de negócio e criação permaneçam protegidas e centralizadas.

```
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.

```

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.

```

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.

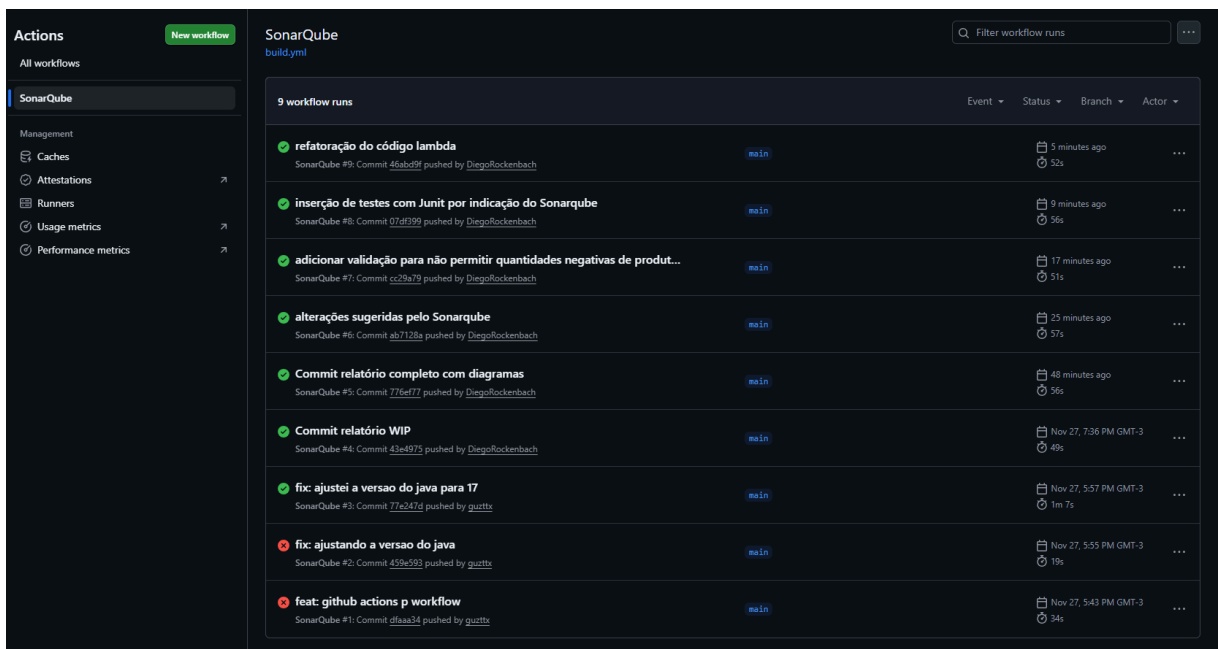
Isso permite adicionar novos observadores sem alterar `Estoque`, basta registrá-los com `addObservador`.

```
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

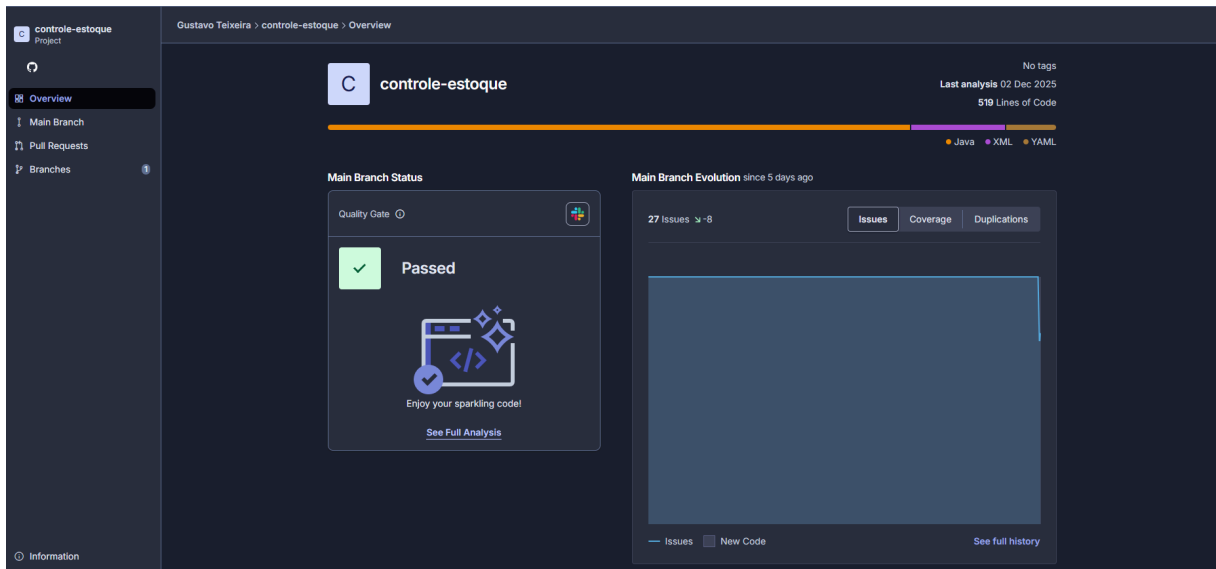
Para garantir a qualidade e manutenibilidade do código, o projeto utiliza SonarQube (via SonarCloud) integrado ao pipeline de CI/CD através do GitHub Actions. A cada commit na branch principal (main) ou abertura/atualização de Pull Request, o workflow automatizado executa análise estática completa do código-fonte, validação de testes unitários e coleta de métricas de cobertura.

A equipe reagiu ativamente aos feedbacks automáticos da ferramenta: observa-se, por exemplo, commits dedicados especificamente à "inserção de testes com JUnit por indicação do Sonarqube" e à "refatoração do código lambda", evidenciando que a verificação de código não foi apenas passiva, mas serviu como guia para aprimoramento da implementação.



Actions		SonarQube	
All workflows		build.yml	
SonarQube		9 workflow runs	
Management		Event • Status • Branch • Actor •	
Caches	Attestations		
Runners	Usage metrics		
Performance metrics			
refatoração do código lambda	SonarQube #9: Commit 46ab09f pushed by DiegoRockenbach	main	5 minutes ago 52s
inserção de testes com Junit por indicação do Sonarqube	SonarQube #8: Commit 07df399 pushed by DiegoRockenbach	main	9 minutes ago 56s
adicionar validação para não permitir quantidades negativas de produt...	SonarQube #7: Commit cc2da79 pushed by DiegoRockenbach	main	17 minutes ago 51s
alterações sugeridas pelo Sonarqube	SonarQube #6: Commit ab7128a pushed by DiegoRockenbach	main	25 minutes ago 57s
Commit relatório completo com diagramas	SonarQube #5: Commit 776ef77 pushed by DiegoRockenbach	main	48 minutes ago 56s
Commit relatório WIP	SonarQube #4: Commit 43e4975 pushed by DiegoRockenbach	main	Nov 27, 7:36 PM GMT-3 49s
fix: ajustei a versao do java para 17	SonarQube #3: Commit 77e241d pushed by gusttts	main	Nov 27, 5:57 PM GMT-3 1m 7s
fix: ajustando a versao do java	SonarQube #2: Commit 459e593 pushed by gusttts	main	Nov 27, 5:55 PM GMT-3 19s
feat: github actions p workflow	SonarQube #1: Commit d1aa34 pushed by gusttts	main	Nov 27, 5:43 PM GMT-3 34s

Após todas as mudanças sugeridas pelo Sonarqube serem aplicadas o projeto atingiu o critério de qualidade, assegurando que o código entregue está em conformidade com as boas práticas definidas.



Além de apontar vulnerabilidades e code smells, a verificação automatizada também garantiu a consistência do ambiente de desenvolvimento, forçando a padronização da versão do Java (Java 17) utilizada tanto no ambiente local quanto no de produção. Essa infraestrutura de verificação complementa a arquitetura robusta descrita anteriormente, garantindo que os padrões de projeto aplicados (Factory, Facade, Strategy, etc.) permaneçam íntegros durante a evolução do sistema.

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;
- criar testes automatizados sugeridos pelo Sonarqube;
- 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.