INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA BAHIA
CAMPUS - SANTO ANTÔNIO DE JESUS - BAHIA

CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISITEMAS

MARCELO DE JESUS

RONALDO CORREIA

FRAKLIN FELIX

ATIVIDADE PADRÕES DE PROJETO- REESTRUTURAÇÃO DE SISTEMAS COM ARQUITETURA MAL PROJETADA COM IOC/ SERVICE LOCATOR

SANTO ANTONIO DE JESUS – BA
2025

MARCELO DE JESUS

RONALDO CORREIA

FRAKLIN FELIX

ATIVIDADE PADRÕES DE PROJETO- REESTRUTURAÇÃO DE SISTEMAS COM ARQUITETURA MAL PROJETADA COM IOC/ SERVICE LOCATOR

Relatório técnico da atividade de Atividade padrões de projeto- Da Arquitetura Mal Projetada à Reestruturação com IoC/Service Locator, elaborado como requisito parcial de avaliação para a disciplina de Padrões de Projeto, ministrada pelo Prof. Felipe Silva.

SANTO ANTONIO DE JESUS – BA

2025

1. INTRODUÇÃO

Este relatório tem como objetivo demonstrar na prática os impactos de um projeto mal projetado ou sem nenhuma estrutura arquitetural, como também a aplicação de técnicas de refatoração e padrões arquiteturais modernos. É comum, haver softwares que mesmo mal estruturados consigam até atender às necessidades imediatas do cliente, mas que, no entanto, viola diretamente os princípios SOLID por exemplo. Para isso, de acordo com o que foi solicitado para elaboração desse trabalho, foi desenvolvido um sistema funcional, inicialmente, de gestão de produtos e vendas, de forma proposital foi desenvolvido sem a adoção dos padrões de projeto, com forte acoplamento, responsabilidades misturadas e ausência de interfaces, de forma resumida gerando automaticamente e natural Anti-Patterns como o God Object, Spaghetti Code, Lava Flow.

No desenvolvimento do sistema, foi composto por duas pastas, uma com a Versão Inicial e outra com a Versão Refatorada. No sistema inicial, foi composto por três classes principais, como: **SistemaLoja**, **Produto** e **Relatório,** onde é possível visualizar concentração múltiplas de responsabilidades, tornando evidente as más práticas de programação e a violação de princípios fundamentais de design, especialmente os princípios SOLID. Esse formato de abordagem permite que na etapa posterior possa se ter um diagnóstico arquitetural crítico, identificando anti-patterns e violações de SOLID, além de aplicar na segunda pasta, Versão Refatorada, táticas de refatoração reorganizando o sistema com a Injeção de Dependência (IoC) e/ou Service Locator. O objetivo final é transformar o sistema inicial em uma versão mais modular, coesa e de fácil manutenção, evidenciando a importância de boas práticas de engenharia de software.

2. DESCRIÇÃO DO SISTEMA

2. Etapa 1 – Implementação Inicial

2.1 Objetivo

Criar um sistema funcional, porém propositalmente mal projetado, com 2 a 4 classes, responsabilidades misturadas, forte acoplamento e ausência de padrões de projeto, permitindo que surjam naturalmente anti-patterns.

2.2 Descrição do Sistema

O sistema implementado possui três classes principais:

SistemaLoja.java

Concentra cadastro, listagem e venda de produtos.

Realiza chamadas diretas para Produto e Relatorio, caracterizando um God Object.

Produto.java

Contém atributos públicos (nome, preco) sem encapsulamento.

Não possui métodos de validação, tornando a classe vulnerável a alterações externas indevidas.

Relatorio.java

Mistura responsabilidades de lógica de negócio e apresentação (gera relatório + imprime no console).

Não é modular nem reutilizável.

2.3 Más práticas propositalmente criadas

God Object: SistemaLoja centraliza todas as responsabilidades do sistema.

Forte acoplamento: todas as classes dependem diretamente umas das outras, sem interfaces ou abstrações.

Quebra de encapsulamento: atributos públicos em Produto.

Mistura de responsabilidades: Relatorio combina lógica de cálculo e saída de dados.

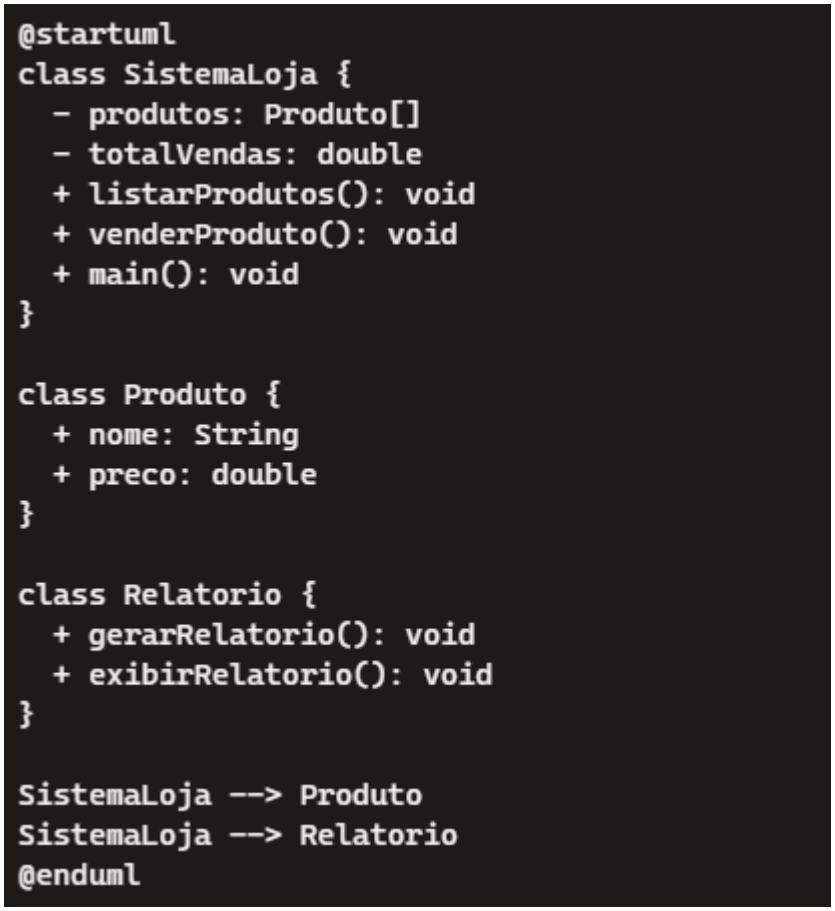
Código repetitivo/confuso: listagem de produtos e vendas repetem lógica sem modularização.

2.4 Observações

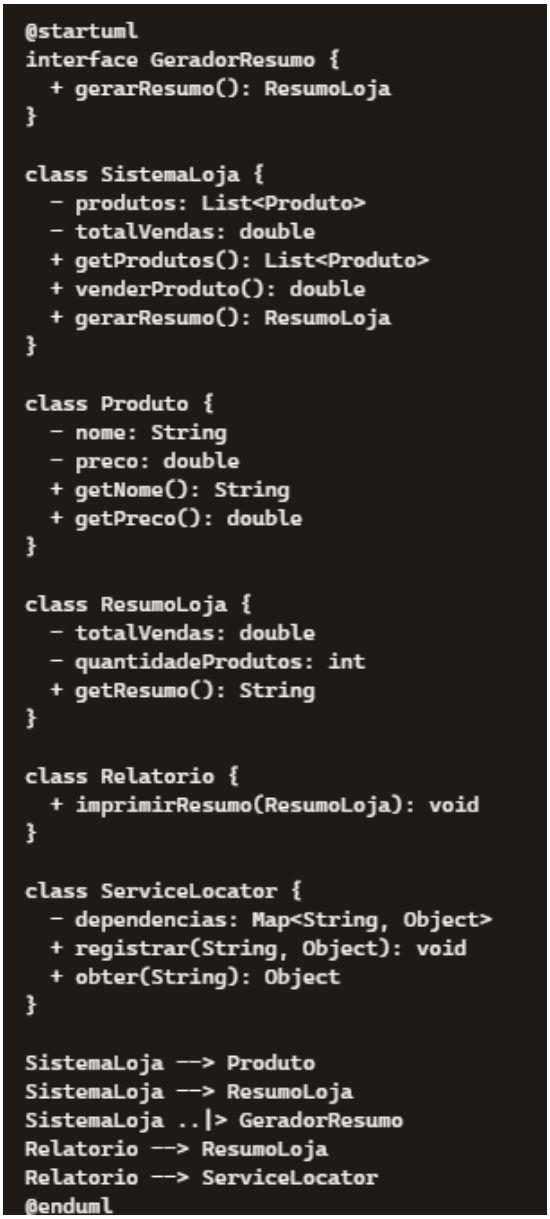
A versão inicial serve como base para estudo comparativo, permitindo observar os problemas que surgem em sistemas mal projetados e justificando, a partir de exemplos concretos, a aplicação de padrões de projeto e princípios de design na versão refatorada.

2.5 UMLs Antes e Depois

UML da Versão Inicial



UML da Versão Refatorada



3 Diagnóstico Arquitetural

3.1 Anti-patterns Identificados

A identificação de *anti-patterns* como *God Object* e *Spaghetti Code* neste sistema segue a definição de Parr (2020), segundo a qual *anti-patterns* representam formas ineficazes de resolver problemas recorrentes, geralmente resultando em aumento de dívida técnica e dificultando a manutenção.

1. God Object (Classe SistemaLoja)

A classe SistemaLoja concentra diversas responsabilidades que deveriam estar distribuídas em diferentes componentes. Nela, encontramos:

- O gerenciamento de produtos (armazenamento em array, inserção e listagem);

- O controle de vendas (atualização do valor total vendido);
- A lógica de interação com o usuário (saída via System.out.println);
- A orquestração do fluxo principal da aplicação, incluindo a criação e utilização da classe Relatorio.

Esse acúmulo de responsabilidades caracteriza o anti-pattern God Object, pois a classe acaba se tornando um "super objeto" que controla praticamente todo o funcionamento do sistema. O impacto negativo é claro: qualquer alteração em uma parte da lógica (exemplo: mudar a forma de listar produtos) pode afetar diretamente o restante do código, tornando o sistema frágil, pouco flexível e difícil de manter.

2. Spaghetti Code (Fluxo misturado)

Outro problema evidente está no estilo de implementação do código. Em diversas partes, a lógica de negócios, a interação com o usuário e o controle de fluxo estão fortemente acoplados. Exemplos:

- O método listarProdutos() mistura a regra de negócio (percorrer a lista de produtos) com a lógica de apresentação (System.out.println);
- O método venderProduto() tanto aplica a lógica da venda quanto imprime a mensagem de saída para o usuário;
- O método main assume um papel de controlador, mas ao mesmo tempo instancia diretamente objetos e executa operações de negócio.

Essa falta de separação entre camadas (apresentação, domínio e persistência) gera um código de difícil leitura e manutenção, caracterizando o anti-pattern conhecido como Spaghetti Code. O termo reflete a ideia de um fluxo "enrolado", em que diferentes responsabilidades estão misturadas sem uma clara organização.

O resultado prático desse anti-pattern é que qualquer tentativa de evolução (como, por exemplo, alterar a forma de exibição das informações de console para interface gráfica ou API) exigiria reescrever métodos inteiros, aumentando o risco de introduzir erros em partes que deveriam ser independentes.

3.2 Violações aos Princípios SOLID

Single Responsibility Principle (SRP)

O princípio da responsabilidade única é violado em mais de uma classe. A classe `SistemaLoja` concentra múltiplas responsabilidades, como controle de produtos, execução de vendas e fluxo principal. Já a classe `Relatorio` mistura a lógica de geração do relatório com a sua exibição em console. Isso viola diretamente a ideia de que uma classe deve ter apenas um motivo para mudar.

Open/Closed Principle (OCP)
O sistema não está aberto para extensão, apenas para modificação. Por exemplo, se fosse necessário alterar a forma de exibição do relatório (de console para PDF ou interface gráfica), a classe `Relatorio` precisaria ser modificada. Isso contraria o princípio OCP, segundo o qual entidades de software devem estar abertas para extensão, mas fechadas para modificação.

Dependency Inversion Principle (DIP)
A dependência entre `SistemaLoja` e `Relatorio` é direta e baseada em implementações concretas. O método `main` instancia um objeto da classe `Relatorio` e o utiliza sem abstrações intermediárias, o que cria forte acoplamento. Caso fosse necessário trocar o tipo de relatório, a classe `SistemaLoja` precisaria ser alterada, demonstrando violação ao DIP, que defende a dependência de abstrações em vez de implementações concretas.

3.3 Táticas de Refatoração Propostas

Com base nos problemas identificados, é possível aplicar algumas táticas de refatoração que tornariam o sistema mais modular, coeso e aderente aos princípios de design.

Encapsular Campo (Classe `Produto`)
Atualmente, os atributos `nome` e `preco` são públicos, permitindo acesso irrestrito. A refatoração proposta consiste em torná-los privados e disponibilizar métodos `getters` e `setters` apropriados. Isso garante encapsulamento, evitando estados inválidos e assegurando maior controle sobre o objeto.

Extrair Método (Classe `SistemaLoja`)
Métodos como `listarProdutos()` e `venderProduto()` acumulam lógica de negócio e apresentação. A refatoração sugerida é extrair a lógica de listagem e de cálculo de vendas para métodos separados que não dependam de exibição em console. Dessa forma, a responsabilidade de apresentar informações ficaria em outra camada ou classe dedicada.

Mover Método (Entre `SistemaLoja` e `Relatorio`)
Atualmente, `Relatorio` depende de detalhes internos de `SistemaLoja` ao receber arrays e variáveis soltas. Uma refatoração adequada seria mover a responsabilidade de fornecer os dados para dentro de `SistemaLoja`, retornando um objeto com as informações de resumo. A classe `Relatorio` ficaria responsável apenas por formatar e exibir o resultado, reduzindo o acoplamento.

Substituir Código por Polimorfismo (Futuro)
Embora não implementado na versão inicial, já é possível antecipar cenários de evolução. Se o sistema vier a ter diferentes tipos de produtos (ex.: com desconto, promocionais), a abordagem atual tenderia ao uso de condicionais. A refatoração proposta é criar subclasses especializadas de `Produto` que implementem diferentes comportamentos, eliminando condicionais e aderindo ao polimorfismo.

3.4 Como Evitar Anti-patterns no Futuro

Os problemas identificados na versão inicial poderiam ter sido evitados com a aplicação antecipada de boas práticas de design e princípios SOLID. Para não repetir anti-patterns como God Object e Spaghetti Code, é fundamental adotar desde o início a separação clara de responsabilidades entre classes e camadas, além de trabalhar sempre com abstrações (interfaces) em vez de implementações concretas.

Outra medida importante é aplicar o refactoring contínuo, não esperando que o sistema se torne grande e difícil de manter para só então organizar o código. Pequenas melhorias constantes reduzem a chance de acumular "dívida técnica".

Por fim, a utilização de padrões arquiteturais modernos como Inversão de Controle (IoC) e Injeção de Dependência contribui para reduzir o acoplamento e aumentar a flexibilidade, facilitando a evolução sem comprometer a estabilidade. Assim, o sistema permanece escalável, coeso e preparado para mudanças futuras.

4 Resultados Obtidos

Encapsular Campo (Classe `Produto`)
Os campos da classe foram definidos como private, agora estão protegidos contra acessos não autorizados, garantindo que só podem ser lidos pelos métodos padrão e não podem ser alterados.

Extrair Método (Classe `SistemaLoja`)
Os métodos de listagem e venda foram alterados, agora getProdutos() retorna o List<produto>, a decisão de como exibir fica a cargo do método que solicitar e não mais da classe SistemaLoja. Agora venderProduto apenas faz o cálculo da venda, isso diminui a poluição visual durante o uso e evita chamadas desnecessárias de outros métodos.

Mover Método (Entre `SistemaLoja` e `Relatorio`)
Foi criada a interface GeradorResumo, a qual sua implementação gera um objeto ResumoLoja que é passado para a serviço de impressão no console, isso reduz o acoplamento e permite que mais tipos de resumos sejam adicionados mais tarde.

IoC e Service locator
As dependências Relatorio, Produto e ResumoLoja eram criadas dentro da classe do sistema, agora a criação de objetos fica centralizada em um único registro, agoras as classes pedem suas dependências ao ServiceLocator.

4.1 Comparativo entre IoC e Service Locator

Estes dois padrões buscam resolver os mesmos problemas, mas a sua abordagem é diferente. O que acontece é que normalmente classes que dependem de outras criam instâncias delas dentro de si, então dizemos que a classe está definindo suas próprias dependências. Isto gera um problema de acoplamento, onde alterações na dependência podem quebrar outra parte do código.

É aí que entra a injeção de dependências e a Inversão de Controle (IoC), visando tirar essa responsabilidade das classes e colocando em um container, com isso todas precisam interagir com ele.
O Service Locator é um padrão de projeto e uma forma de IoC, uma de suas diferenças é a transparência em relação as dependências que ele controla, enquanto no caso de um container todas são bem claras, no SL eles são armazenados em um Map, o que dificulta saber quais dependem de quais.

3. CONCLUSÃO

Este projeto demonstrou a importância de aplicar boas práticas de padrões de projeto em sistemas mal projetados. A evolução passou por três etapas fundamentais:

1. **Versão Inicial** — Sistema acoplado, com baixa coesão e difícil manutenção.
2. **Versão Refatorada** — Introdução de uma arquitetura moderna, com inversão de controle e desacoplamento de dependências.

Com essas melhorias, o sistema tornou-se mais flexível, testável e escalável, facilitando futuras alterações e integrações. Além disso, o uso de padrões como Service Locator e injeção de dependência reforça a importância de pensar em arquitetura desde o início do desenvolvimento. Este trabalho evidencia como a refatoração e a adoção de boas práticas podem transformar um código frágil em uma base sólida para crescimento sustentável.

2. LINK REPOSITORIO

- https://github.com/marceloteclas/Reestrutura-o-de-sistema-mal-projetado-com-Ioc-Service-Locator.git

3. REFERÊNCIAS

PARR, Kealan. *Anti-patterns You Should Avoid in Your Code*. FreeCodeCamp, 23 nov. 2020. Disponível em: https://www.freecodecamp.org/news/antipatterns-to-avoid-in-code/. Acesso em: Agosto de 2025