Trabalho 1: Refatoração com Padrões de Projeto em Sistema Legado

Sistema Legado: Arcervo Bibliotech

Alunos:

- Gabriel Alves Pessoa
- Juan David de Bizerra Pimentel

1. Introdução ao Sistema Legado

O sistema legado analisado é o **Acervo Bibliotech**, um Sistema de Gerenciamento de Biblioteca (SGB) acadêmico específico para a MTI Adelino Cunha Alcântara. O sistema nasceu como Projeto de Extensão para a cadeira de Projeto Integrado em Engenharia de Software I, desenvolvido em Java com Spring Boot, React e PostgreSQL, o sistema é responsável por gerenciar o acervo de livros, alunos, turmas, empréstimos, ocorrências disciplinares e geração de relatórios em PDF.

Sua arquitetura atual é em camadas (Controller, Service, Repository), mas com 17 classes de serviço e 16 repositórios, o sistema começou a apresentar problemas significativos de manutenibilidade e coesão, motivando esta refatoração. Vale ressaltar também, que neste presente documento contemplamos apenas a camada do Back-End, ou seja, as refatorações ocorreram na linguagem Java, no framework Spring Boot.

2. Problemas Identificados (Antes da Refatoração)

A análise inicial do código-fonte revelou cinco problemas críticos de design que violam princípios fundamentais de reuso e SOLID, foram eles:

1. **Classe Deus (God Class):** A classe PdfExportService possuía mais de 300 linhas e centralizava a lógica de criação de *todos* os relatórios do sistema (frequência, acervo, ocorrências, etc.). Isso violava o Princípio da Responsabilidade Única (SRP), gerava alta duplicação de código e tornava a manutenção ou adição de novos relatórios extremamente difícil.
2. **Duplicação de Estrutura:** Mesmo que a "God Class" fosse quebrada, a lógica de *estrutura* do PDF (criação de cabeçalhos, rodapés, configuração de tabelas, abertura/fechamento de documento) era repetida em todos os métodos de geração, violando o princípio DRY (Don't Repeat Yourself).
3. **Acoplamento Excessivo:** A classe EmprestimosService era um segundo ponto de

centralização de regras. Ela possuía 10 dependências diretas injetadas (incluindo repositórios de Aluno, Exemplar, Usuário, além de serviços de token e email), tornando-a frágil, difícil de testar e com baixa coesão.

4. **Ações Secundárias Acopladas:** Lógicas de negócio secundárias (como enviar email de notificação, registrar log de auditoria ou atualizar estatísticas) estavam diretamente acopladas dentro dos métodos de negócio principais (ex: realizarEmprestimo). Isso tornava impossível adicionar uma nova ação (ex: "enviar SMS") sem modificar o método principal, violando o Princípio Aberto/Fechado (OCP).
5. **Falta de Abstração (Sistema de Notificação):** O serviço de empréstimos possuía uma dependência "hardcoded" do EmailSend, tornando impossível trocar o canal de notificação (para SMS, Push, etc.) sem alterar a lógica de negócio, violando o OCP.

3. Descrição das Refatorações Realizadas

Para solucionar os problemas identificados, aplicamos **cinco** padrões de projeto GoF, organizados em frentes de refatoração:

3.1 Módulo de Notificações (Padrão Strategy)

Justificativa

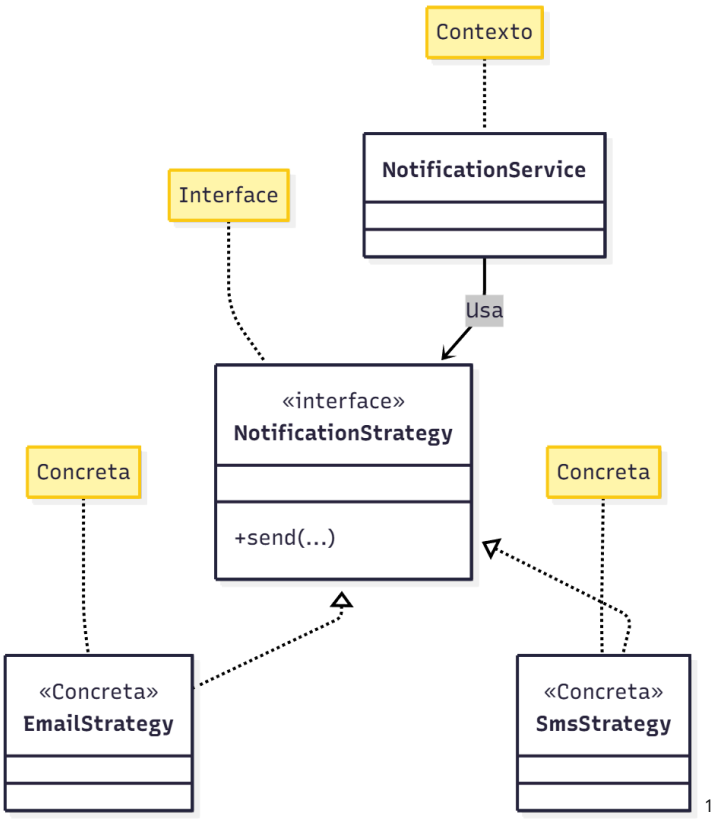
O **Strategy Pattern** permite definir uma família de algoritmos (diferentes canais de notificação), encapsulá-los e torná-los intercambiáveis. Isso resolve o problema de dependência rígida do EmailSend e permite adicionar novos canais (SMS, Push, WhatsApp) sem modificar o código existente.

Arquitetura (Antes/Depois)

Antes: O NotificationService (ou EmprestimoService) chamava diretamente a classe EmailSend.

Depois: O NotificationService (Context) agora depende apenas da interface NotificationStrategy. Diferentes implementações (Email, SMS) podem ser injetadas e trocadas em tempo de execução.

Diagrama da Solução (Strategy):



Benefícios

- Desacoplamento entre lógica de notificação e canal
- Facilita adição de novos canais
- Permite configurar canal por tipo de notificação
- Testabilidade melhorada

3.2 Módulo de Relatórios (Padrões Factory Method e Template Method)

Justificativa

- **Factory Method:** Foi escolhido para resolver o problema da "God Class" (PdfExportService). O padrão permite delegar a responsabilidade de *criação* de cada relatório (produto) para uma classe "fábrica" específica, descentralizando a lógica.

[1] Embora se tenha alguns elementos do diagrama “tortos”, como algumas setas, o diagrama foi desenvolvido exclusivamente para esse trabalho utilizando a ferramenta Mermaid.

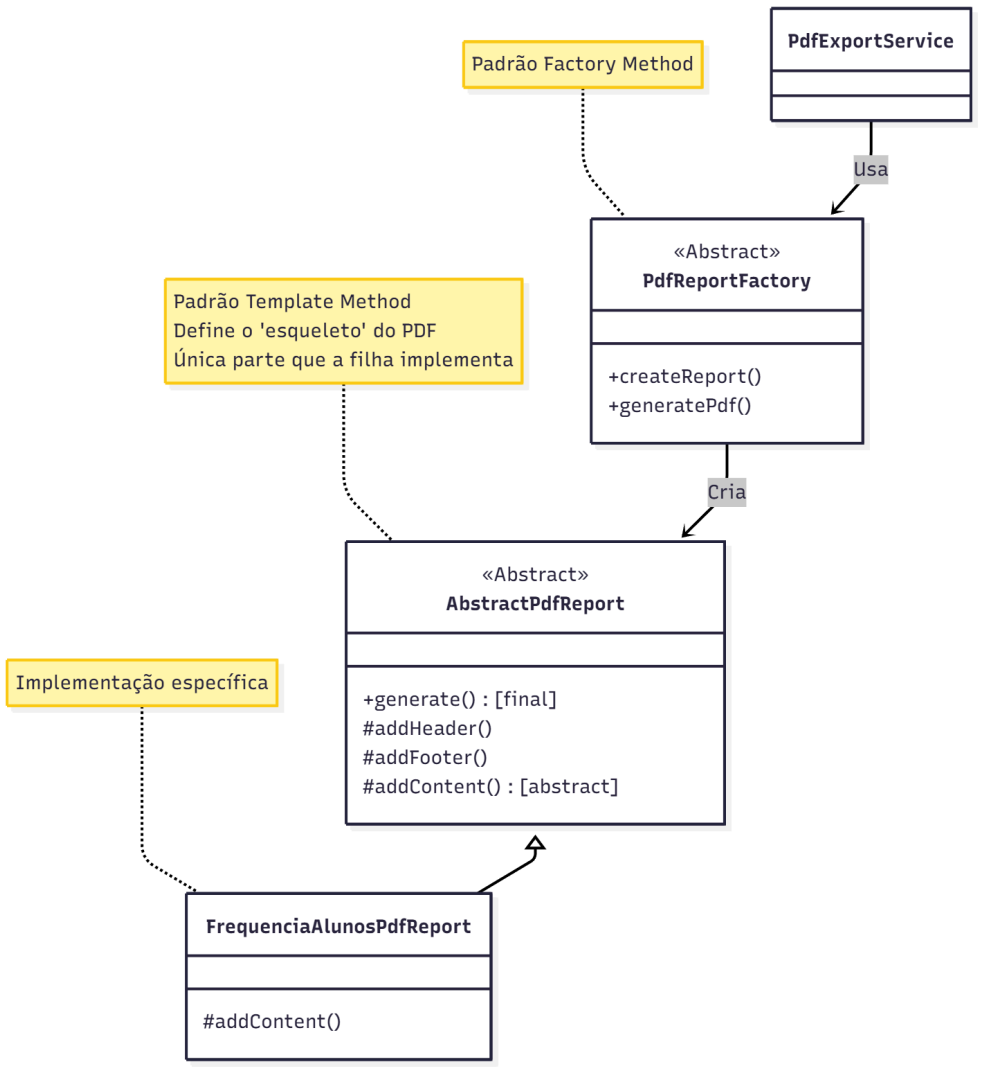
- **Template Method:** Foi escolhido para resolver o problema da *duplicação de estrutura* entre os relatórios. O padrão define um "esqueleto" de algoritmo (a estrutura do PDF) na classe pai, permitindo que as classes filhas redefinam apenas o *conteúdo* específico.

Arquitetura (Antes/Depois)

Antes: Uma única classe PdfExportService com alta complexidade e baixo reuso.

Depois: A refatoração combinada resultou na seguinte arquitetura, onde o PdfExportService apenas invoca a *Factory* correta, que por sua vez cria um *Produto* que segue o *Template* estrutural.

Diagrama da Solução (Factory + Template):



Benefícios (Factory Method + Template Method)

- Separação de Responsabilidades:
    - Descentraliza a lógica da "God Class". Cada relatório passa a ter uma responsabilidade única.
- Reutilização Máxima:
    - O **Template Method** eliminou **83%** do código duplicado ao centralizar a estrutura comum (cabeçalho, tabelas) na classe `AbstractPdfReport`.
- Extensibilidade:
    - Adicionar um novo relatório ao sistema agora requer apenas a criação de duas novas classes (Factory e Produto), sem modificar o código existente.
- Testabilidade:
    - Cada relatório pode ser testado isoladamente, validando apenas sua implementação de `addContent()`.
- Manutenibilidade:
    - A lógica de criação (Factory) está separada da lógica de estrutura (Template), facilitando correções e evoluções.

3.3 Refatoração 2: Módulo de Empréstimos (Padrões Facade e Observer)

Justificativa

- **Facade:** Foi escolhido para resolver o *acoplamento excessivo* do EmprestimosService. O padrão provê uma interface unificada e simplificada (EmprestimoFacade) que encapsula a complexidade de múltiplos subsistemas (ex: validadores de Aluno, Exemplar, Empréstimo, e múltiplos repositórios).
- **Observer:** Foi escolhido para *desacoplar ações secundárias* da lógica principal. O padrão permite que o EmprestimoFacade (Subject) apenas "publique" um evento (ex: EMPRESTIMO_CRIADO) sem conhecer seus assinantes (Observers).

Arquitetura (Antes/Depois)

Antes: EmprestimosService com 10 dependências diretas, misturando validação, orquestração e ações secundárias (notificação, logs).

Depois: O EmprestimosService (cliente) agora depende apenas do EmprestimoFacade. O Facade orquestra os validadores e, ao final, publica um evento para os Observers, que reagem de forma independente.

Diagrama da Solução (Facade):

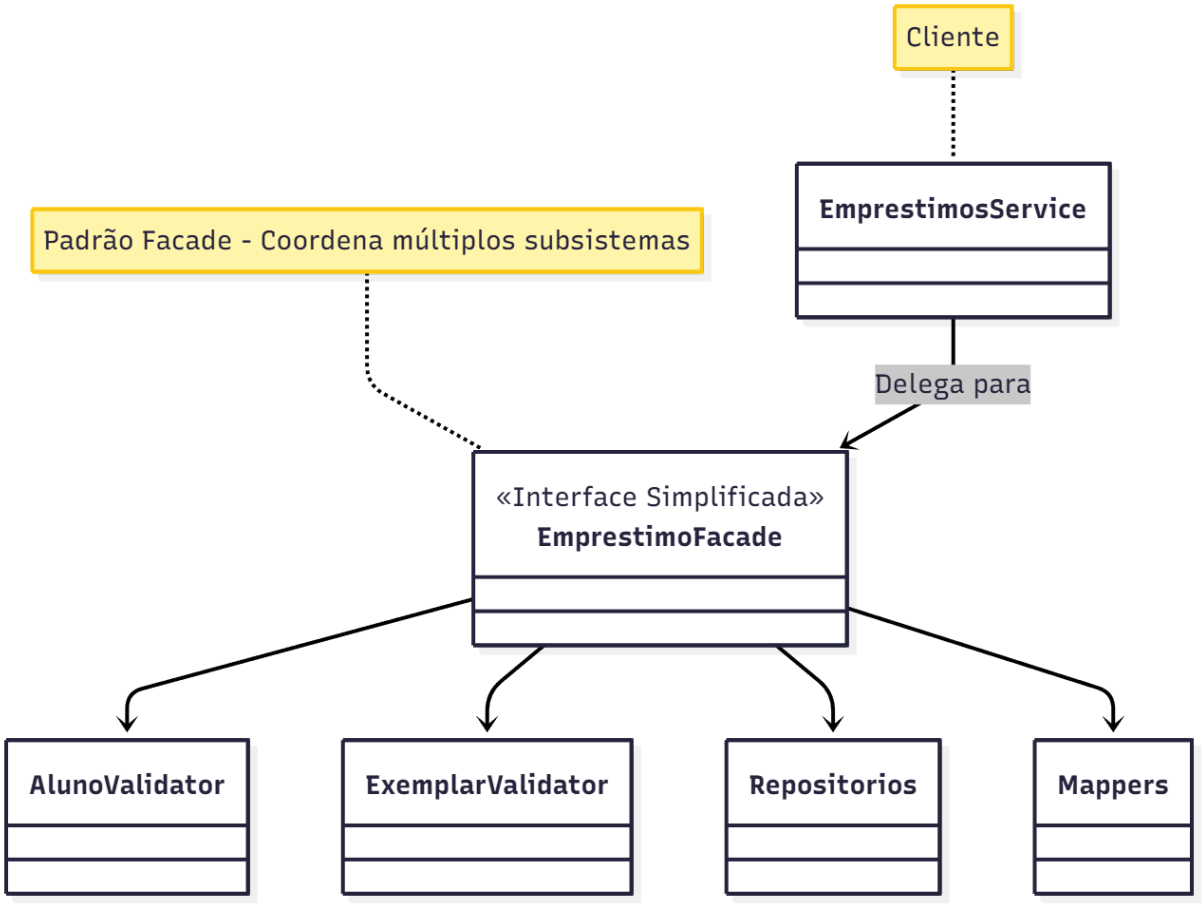
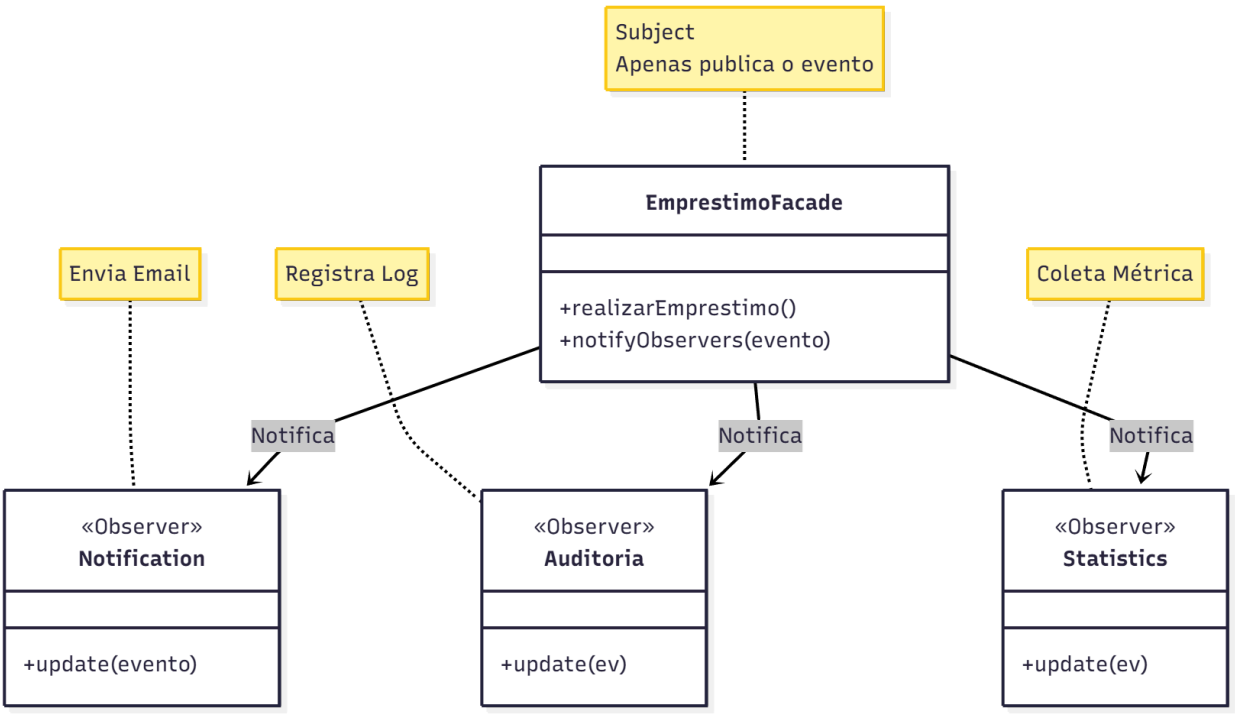


Diagrama da Solução (Observer):



Benefícios (Facade + Observer)

- **Simplificação e Coesão:**
  - O Facade reduziu drasticamente a complexidade do EmprestimosService. Métodos que antes tinham 40 linhas de lógica misturada (validação, persistência, notificação) agora têm apenas 1 linha de delegação.
- **Redução de Acoplamento:**
  - O serviço principal não depende mais de 10 classes concretas (repositórios, mappers, etc.), apenas da abstração do Facade.
- **Desacoplamento de Ações:**
  - O Observer quebrou a dependência "hardcoded" de ações secundárias. A lógica de negócio (Facade) apenas "publica" um evento, sem saber quais ou quantos sistemas irão reagir a ele.
- **Extensibilidade:**
  - É trivial adicionar novas ações (ex: SmsObserver ou CacheObserver) apenas criando uma nova classe e registrando-a como ouvinte, sem nenhuma alteração na lógica de negócio principal.
- **Tolerância a Falhas:**
  - Falhas em um Observer (ex: erro no envio de email) não interrompem a execução de outros Observers (ex: registro de auditoria).

4. Conclusão: Ganhos de Reutilização e

Manutenibilidade

A aplicação dos quatro padrões de projeto transformou a arquitetura do Bibliotech, gerando ganhos mensuráveis de qualidade, reutilização e manutenibilidade, alinhados aos princípios SOLID.

Ganhos Principais:

- **Manutenibilidade:**
  - **Antes:** Para adicionar um relatório, era preciso modificar a "God Class" PdfExportService. Para adicionar uma notificação, era preciso modificar o EmprestimosService.
  - **Depois:** Para adicionar um relatório, basta criar uma nova *Factory* e um *Produto* (sem alterar classes existentes). Para adicionar uma notificação (ex: SMS), basta criar um SmsObserver (sem alterar o Facade).
- **Reutilização:**
  - O **Template Method** eliminou **83%** do código duplicado na geração de PDFs, centralizando a estrutura comum (cabeçalhos, rodapés, tabelas) na classe AbstractPdfReport.
  - Os **Validators** (parte do Facade) centralizaram regras de negócio (ex: alunoValidator.validarParaEmprestimo()) que antes estavam duplicadas em diferentes serviços.
- **Coesão e Acoplamento:**
  - O EmprestimosService teve sua complexidade e acoplamento drasticamente reduzidos. Métodos que possuíam **40 linhas** de lógica mista foram reduzidos para **1 linha** de delegação ao Facade.
  - O PdfExportService foi reduzido em **77%** (de 300 para 70 linhas).
  - O padrão **Observer** quebrou a dependência direta do negócio para com sistemas secundários (email, log), permitindo que eles evoluam de forma independente.

Métricas de Impacto:

Métrica	Antes	Depois	Redução
Linhas no PdfExportService	~300	~70	**-77%**
Linhas no realizarEmprestimo()	~40	1	**-97.5%**
Dependências do EmprestimosService	10	6 (sendo 1 o Facade)	**-40%**
Duplicação (Estrutura PDF)	~360 linhas	~60 linhas (no Template)	**-83%**

Em suma, o sistema legado, que era monolítico e rígido, evoluiu para uma arquitetura modular, flexível e coesa, onde os componentes possuem responsabilidades únicas e são abertos à extensão, facilitando o reuso e a evolução futura.

5. Referências

1. **GAMMA, Erich et al.** *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
2. **MARTIN, Robert C.** *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.
3. **FOWLER, Martin.** *Refactoring: Improving the Design of Existing Code*. 2nd Edition. Addison-Wesley, 2018.