UNIVERSIDADE FEDERAL DO CEARÁ

Luis Felippe Morais de Lima - 538605

Pedro Emanuel Santana - 537386

Refatoração com Padrões de Projeto em Sistema Legado - Bookify

Quixadá

2025

O sistema

O sistema Bookify, selecionado para análise e refatoração neste trabalho, foi inicialmente desenvolvido em 2023 para a cadeira de Projeto Integrado I. Sua arquitetura de implementação se baseia na utilização de Java para a lógica de back-end e as regras de negócio. O front-end e a interface gráfica do usuário foram construídos com JavaFX, com o apoio de CSS para estilização. Para a persistência de dados e o gerenciamento do acervo, foi adotado o sistema gerenciador de banco de dados PostgreSQL.

A ideação desse projeto surgiu da observação de que o controle de acervo e empréstimo de livros da biblioteca da **Escola Estadual de Ensino Profissional Dr. José Alves da Silveira** de Quixeramobim, era realizado de maneira inteiramente manual. Esse método antiquado envolvia o uso de planilhas impressas em papel, nas quais se registravam as informações básicas dos leitores e os números de tombamento dos livros. Esse procedimento resultava em um controle de empréstimos e devoluções ineficiente, o que, em alguns casos, contribuía para a perda de itens do acervo. Além disso, tarefas bibliotecárias essenciais, como o tombamento de novos livros, a gestão de pedidos de aquisição e o controle geral do acervo, também eram executadas manualmente.

Desse modo, o Bookify surgiu visando implementar um sistema que unificasse, de forma prática e simples, o gerenciamento de todos os processos da biblioteca. Ele teve como principais objetivos:

- Facilitar e controlar de maneira mais eficiente o processo de empréstimo e devolução;
- Automatizar o acervo local;
- Reduzir o número de processos manuais;

Para atingir esses fins, o sistema foi desenhado para oferecer funcionalidades essenciais, como CRUD de livros, alunos, professores, registrar o *check-in* e *check-out* de volumes e empréstimos, além de permitir visualizar o histórico dos empréstimos feitos anteriormente, o que facilita o contato com as pessoas que podem estar em débito com a biblioteca.

Problemas Identificados

A análise do código do sistema Bookify antes da refatoração mostrou problemas sérios na arquitetura. Eles são típicos de um sistema legado, indicando que ele pode ser difícil de manter e caro de evoluir. Os pontos fracos encontrados se concentram em dois grandes temas: a falta de um domínio claro e o excesso de tarefas nas classes de controle.

Falta de Classes de Entidade

O problema mais visível e que mais afeta a orientação a objetos do sistema é a ausência de classes de Entidade. O projeto não usa classes para representar conceitos importantes, como `Livro`, `Aluno`, `Professor` ou `Empréstimo`, que são a base de qualquer biblioteca.

Em vez de criar objetos que combinam dados e comportamentos, o sistema trata a persistência de forma muito simples. Os dados necessários são capturados diretamente nos campos da interface (o *front-end* FXML) e, em seguida, são passados para funções que se comunicam diretamente com o banco de dados **PostgreSQL** para salvar, atualizar ou buscar informações.

Essa forma de trabalhar ignora totalmente os princípios da Programação Orientada a Objetos e cria uma **ligação muito forte (alto acoplamento)** entre a tela do sistema e a parte que salva os dados. Sem um modelo de domínio claro, o código fica complexo de entender, difícil de evoluir, e as regras de negócio não podem ser reutilizadas, pois estão misturadas à interface.

Controladores com Muitas Responsabilidades (*God Controllers*)

Outro problema importante é o excesso de responsabilidade nas classes Controller. Essas classes, que deveriam apenas receber um clique ou um comando do usuário e coordenar o que acontece em seguida, estão, na verdade, fazendo uma grande parte da lógica de negócio e das operações que deveriam estar em outras camadas.

Esse problema, é uma clara violação do Princípio da Responsabilidade Única. Além de gerenciar eventos de tela, os Controllers também:

- **Contêm as Regras de Negócio:** Eles fazem validações e executam a lógica central.
- **Fazem a persistência:** Em vez de delegar, eles chamam diretamente as funções que interagem com o banco de dados.

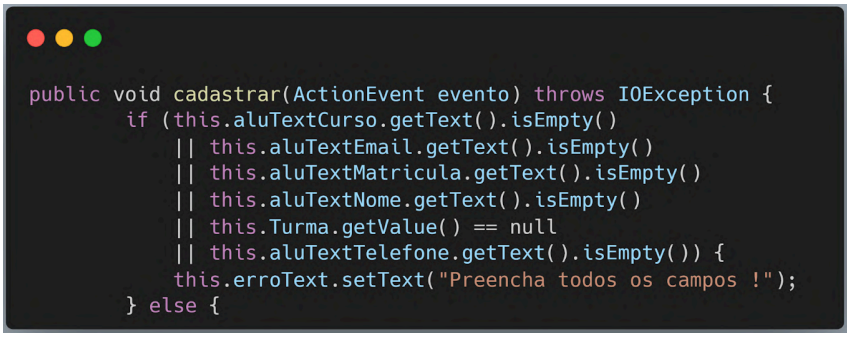
Essa concentração de tarefas resulta em classes grandes, pouco flexíveis e com baixa coesão. Isso torna difícil entender o que está acontecendo e, principalmente, impede que a lógica de negócio seja testada de forma isolada, pois ela está colada à interface. Mudar uma regra de negócio acaba exigindo alterar um Controller já complicado, o que aumenta o risco de criar novos erros. Isolar a lógica do Controller é essencial para facilitar a manutenção futura do sistema.

É importante deixar claro que os problemas maiores da arquitetura — como não ter classes de entidade e ter *Controllers* muito grandes — **não foram resolvidos diretamente pelos padrões de projeto**. Na verdade, esses problemas estruturais foram a causa de outros problemas mais práticos e imediatos no código. Foi nisso que a nossa refatoração focou. Os problemas que resolvemos incluem: código duplicado, cheques condicionais enormes e parecidos em várias partes do sistema, e a criação repetida da mesma classe em lugares diferentes. Os padrões de projeto foram a ferramenta para combater essa duplicação e rigidez, trazendo mais ordem e controle.

O único problema que não possui muita relação com o que foi citado anteriormente é a instanciação da classe responsável pela troca de tela em diferentes arquivos, sendo que isso poderia ser resolvido ao pegar a mesma instância utilizada em uma classe, para as outras.

Padrão Strategy

Durante a etapa de cadastro de Livros, Alunos e Professores, é realizada uma validação dos campos preenchidos pelo usuário. Cada classe possui uma validação própria, mas executam a mesma ação, apenas de forma diferente.

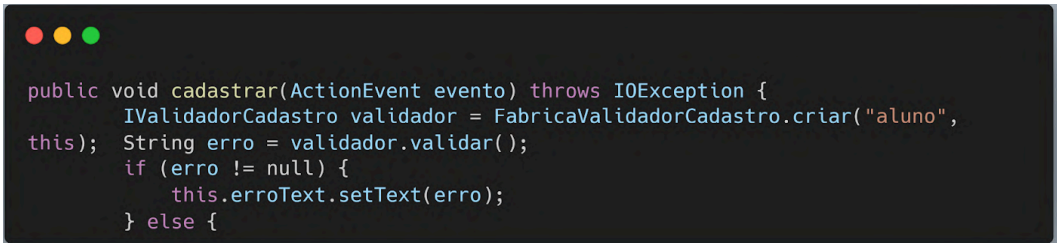


```
public void cadastrar(ActionEvent evento) throws IOException {
        if (this.aluTextCurso.getText().isEmpty()
            || this.aluTextEmail.getText().isEmpty()
            || this.aluTextMatricula.getText().isEmpty()
            || this.aluTextNome.getText().isEmpty()
            || this.Turma.getValue() == null
            || this.aluTextTelefone.getText().isEmpty()) {
            this.erroText.setText("Preencha todos os campos !");
        } else {
```

Validação na classe Aluno, algo parecido se repete nas outras duas.

O padrão Strategy se encaixa como solução por termos essa variação de um mesmo algoritmo. O código se duplica em cada classe, cada uma com seu ‘if/else’. Essa tarefa pode ser delegada a uma estratégia de validação, sem que a implementação fique explícita a quem consome.

O conceito do Strategy é aplicado com a criação de uma interface comum **IValidadorCadastro**, que define a ação de validar em forma de método. Assim, as três implementações concretas ficam restritas ao definido no contrato.



```
public void cadastrar(ActionEvent evento) throws IOException {
        IValidadorCadastro validador = FabricaValidadorCadastro.criar("aluno",
this);  String erro = validador.validar();
        if (erro != null) {
            this.erroText.setText(erro);
        } else {
```

AlunosCadastroController usando validador concreto dos campos do aluno.

Antes, a lógica de validação em cada controller de cadastro (AlunosCadastroController, LivrosCadastroController e ProfessorCadastroController) estava duplicada, resultando em código repetitivo e difícil manutenção. Mudanças nas regras de validação agora podem ser feitas apenas na classe validadora correspondente (ValidadorAluno, ValidadorLivro ou ValidadorProfessor) e novas validações podem ser mais facilmente adicionadas, basta que implementem

IValidadorCadastro. Dessa forma, os controllers focam na ordenação do fluxo e delegam o trabalho de validação a outra camada.

Padrão Factory

Antes da aplicação do padrão Factory, cada controller teria que instanciar manualmente seu validador correspondente, criando acoplamento direto entre controllers e implementações concretas de validadores. O padrão busca ajudar a centralizar e simplificar a criação desses validadores, melhorando o acoplamento e delegando a responsabilidade para uma classe criadora, que lida com os diferentes tipos.

```java
public class FabricaValidadorCadastro {

    public static IValidadorCadastro criar(String tipo, Object controller) {
        return switch (tipo.toLowerCase()) {
            case "aluno" -> new ValidadorAluno((AlunosCadastroController) controller);
            case "livro" -> new ValidadorLivro((LivrosCadastroController) controller);
            case "professor" -> new ValidadorProfessor((ProfessorCadastroController) controller);
            default -> throw new IllegalArgumentException("Tipo de validador desconhecido: " + tipo);
        };
    }
}
```

Centralização no criar, método estático que identifica qual validador tratar.

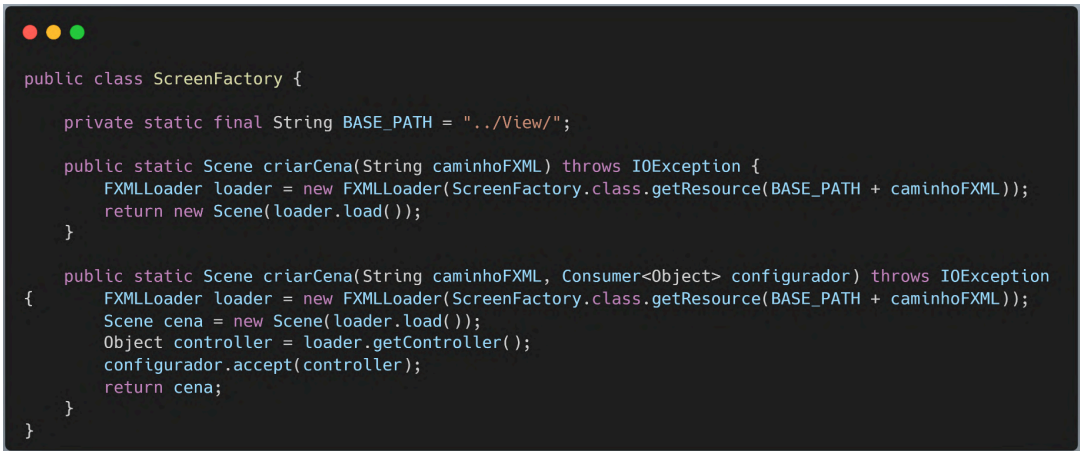
Um comportamento semelhante foi identificado para a criação de telas, antes tratadas por um gigantesco *switch-case* potencialmente problemático. Uma segunda fábrica foi implementada para tratar da manipulação exclusiva da navegação de telas. Em conjunto com o Strategy, o controller deixa de conhecer regras de negócio e construção que antes faziam parte de sua responsabilidade.

```java
public class TelasView {

    private static TelasView instance;
    private final Map<String, String> telas = new HashMap<>();

    private TelasView() {
        telas.put("alunos/cadastro", "Alunos-cadastro-window.fxml");
        telas.put("alunos/menu", "Alunos-window.fxml");
        telas.put("professores/menu", "Professor-window.fxml");
        telas.put("professores/cadastro", "Professor-cadastro-window.fxml");
        telas.put("home", "Home-window.fxml");
        telas.put("login", "Login-window.fxml");
        telas.put("emprestimos/realizar", "Realizar-emprestimo.fxml");
        telas.put("livros/menu", "Livros-window2.fxml");
        telas.put("livros/cadastro", "Livros-cadastro-window.fxml");
        telas.put("alunos/listagem", "Aluno-listagem-window.fxml");
        telas.put("professores/listagem", "Professor-listagem-window.fxml");
        telas.put("livros/listagem", "Livro-listagem-window.fxml");
        telas.put("emprestimos/listagem", "Emprestimo-listagem-
window.fxml");telas.put("professores/edicao", "Professor-edicao-window.fxml");
        telas.put("alunos/edicao", "Aluno-edicao-window.fxml");
        telas.put("livros/edicao", "Livros-edicao-window.fxml");
    }
```

Gerenciamento de telas desacoplado dos controllers e uso de estruturas para tornar o código mais conciso.



```
public class ScreenFactory {

    private static final String BASE_PATH = "../View/";

    public static Scene criarCena(String caminhoFXML) throws IOException {
        FXMLLoader loader = new FXMLLoader(ScreenFactory.class.getResource(BASE_PATH + caminhoFXML));
        return new Scene(loader.load());
    }

    public static Scene criarCena(String caminhoFXML, Consumer<Object> configurador) throws IOException
{       FXMLLoader loader = new FXMLLoader(ScreenFactory.class.getResource(BASE_PATH + caminhoFXML));
        Scene cena = new Scene(loader.load());
        Object controller = loader.getController();
        configurador.accept(controller);
        return cena;
    }
}
```

Fábrica para criação de telas, permitindo diferentes consumos.

Padrão Singleton

Por último, notamos a múltipla instanciação do navegador de telas do JavaFX em diversos arquivos, criando uma redundância e perdendo a informação sobre o estado atual da aplicação. O acesso global e único garantido pelo padrão Singleton auxiliou na criação de um ponto único e consistente, sem perder o Stage atual internamente.

Assim, qualquer controller pode solicitar mudanças sem se preocupar em gerenciar o contexto e ainda permite que se possa incrementar com etapas antes e depois da troca, como transições. Ter um único objeto criado em memória também possui claras vantagens de memória e desempenho.

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

Restrição ao uso de única instância de navegação.

Repositório Github: https://github.com/Luis-Felippe/bookify_refactor