Documentação da Refatoração do Projeto

Nome dos Alunos:

Genilson C. da Silva Junior - 822161753

2025

ÍNDICE DETALHADO

1. Introdução

O sistema escolhido para refatoração é um gerenciador de reservas de livros de biblioteca, originalmente estruturado como um script Python único, utilizando variáveis globais, entrada de usuário direta e acesso a arquivo de forma acoplada à lógica de negócio.

O código original apresentava diversos problemas típicos de software legado: baixa modularidade, dificuldade de manutenção, ausência de testes, duplicação de lógica e forte dependência entre camadas.

O repositório refatorado final encontra-se disponível em:

https://github.com/JuniorCSilva/A3CALVETTI

1.1. Objetivo e Contexto

O objetivo principal foi refatorar o código legado para:

- Melhorar legibilidade
- Aplicar Clean Code
- Reduzir complexidade
- Adicionar testes
- Organizar o sistema em camadas
- Criar um histórico completo de commits
- Configurar pipeline de CI com pytest

Essa refatoração atende integralmente às exigências da atividade A3 da disciplina.

1.2. Benefícios e Adequação aos Critérios

A refatoração trouxe melhorias claras:

Legibilidade

Melhoria de nomes, separação de responsabilidades e organização.

Estrutura

Implementação de camadas claras:

- `domain` — modelos
- `repository` — persistência
- `service` — regras de negócio
- `app` — interface

Clean Code

Remoção de duplicações e excesso de responsabilidades.

Testes

Criação de testes unitários com pytest garantindo funcionamento correto.

CI

Pipeline GitHub Actions executando testes automaticamente.

Git e Versionamento

Commits incrementais simulando a prática profissional de refatoração.

1.3. Tecnologias e Patterns Utilizados

Python 3.11

pytest (testes unitários)

GitHub Actions (CI)

Princípios de Clean Code

Arquitetura em camadas

Inversão de dependência (DIP) entre serviços e repositórios

Domain-driven design básico

2. Metodologia

A refatoração seguiu a abordagem incremental:

1. Análise estática do código legado
2. Identificação de code smells
3. Implementação da nova arquitetura
4. Migração da lógica por partes
5. Criação de testes
6. Ajuste e estabilização
7. Documentação
8. Automação (CI)

Todos os commits foram registrados no GitHub seguindo boa prática de versionamento.

3. Principais Deficiências no Código Legado

3.1. Acoplamento excessivo

O código misturava lógica de regras, entrada de usuário, impressão e escrita em arquivos.

3.2. Ausência de camadas

O código não possuía nenhuma separação entre domínio, regras de negócio e persistência.

3.3 Código monolítico

Todas as funções estavam no mesmo arquivo e compartilhavam variáveis globais.

3.4 Ausência de testes

Nenhuma parte era testável de forma isolada devido ao acoplamento e globalização do estado.

3.5 Baixa legibilidade

Funções grandes, nomes ruins e regras duplicadas.

4. Historico de commit

4.1. Commit 1 — Adição do código legado

Adicionado o arquivo `library_original.py` contendo todo o sistema acoplado.

4.2. Commit 2 — Criação da estrutura base

Criadas pastas: `src/`, `domain/`, `repository/`, `service/`, `tests/`, `.github/workflows`.

4.3. Commit 3 — Implementação das entidades

Criadas classes `User`, `Book`, `Reservation`.

4.4. Commit 4 — Repositórios

Isolada camada de acesso a arquivos.

4.5. Commit 5 — Serviços

Regras de negócio migradas do legado para classes:

- **UserService**
- **BookService**
- **ReservationService**

4.6. Commit 6 — Implementação da CLI

Arquivo `app.py` estruturado separando apresentação e regras.

4.7. Commit 7 — Testes

Criação do arquivo `test_reservation_service.py`.

4.8. Commit 8 — CI

Criação do arquivo `.github/workflows/ci.yml`.

4.9. Commit 9 — Limpeza e ajustes finais

Ajustes finais de validação, organização e README.

5. Justificativa das Mudanças

5.1. Legibilidade

Nomes de classes, métodos e variáveis foram padronizados para refletir propósito claro.

5.2. Estrutura

A divisão em camadas reduz acoplamento e facilita manutenção e evolução futura.

5.3. Comentários e Documentação

Documentação apenas quando necessário, com código limpo sendo autoexplicativo.

5.4. Princípios Aplicados

- **SRP**: cada classe possui apenas uma responsabilidade
- **OCP**: regras novas podem ser adicionadas sem alterar código existente
- **DRY**: eliminação de duplicações
- **KISS**: código simples e direto
- **YAGNI**: nada foi implementado além do necessário

5.5. Testabilidade

A separação adequada permitiu criação de testes confiáveis isolando regras de negócio.

6. Testes Unitários Implementados

Testes do ReservationService

Casos testados:

1. Reserva bem-sucedida
2. Detecção de livro já reservado
3. Cancelamento bem-sucedido
4. Erro ao cancelar inexistente

Os testes garantem conformidade com a regra original.

7. Conclusão

O processo de refatoração demonstrou como código legível, modular e testável reduz custos de manutenção e aumenta robustez.
A transformação do sistema mostrou na prática que Clean Code:

- reduz bugs
- facilita evolução
- melhora colaboração
- aumenta vida útil do software

O sistema final é sustentável, escalável e pronto para futuras melhorias.