Relatório Final da Aplicação e Refatoração

Projeto: Gerenciador de Campeonatos de Futebol (GCF)
Disciplina: Gestão e Qualidade de Software

Grupo:
- Artur Rosa Correia - RA: 824135943
- Gustavo Silveira Benicio - RA: 824134160
- Luan Bernardo Alves - RA: 824134204
- Victor Hugo Santos Nunes - RA: 825163477

1. Introdução

Este relatório documenta o processo completo de refatoração do sistema GCF (Gerenciador de Campeonatos de Futebol), que teve como objetivo transformar uma aplicação funcional, mas sem boas práticas, em um projeto com código limpo, bem estruturado e seguindo os princípios da engenharia de software moderna.
O sistema permite gerenciar times, campeonatos, jogos e calcular classificações automaticamente. Na primeira versão, o foco era apenas fazer funcionar. Nesta segunda fase, nosso objetivo foi refatorar e garantir a qualidade da aplicação.

2. Estado Inicial do Projeto

2.1 Contexto

Quando iniciamos a refatoração, o projeto estava funcional mas apresentava diversos problemas que dificultavam a manutenção e expansão do código:

Problemas Identificados:

- Banco de dados despadronizado: Nomes de colunas inconsistentes e dados desorganizados
- Arquitetura desorganizada: Sem separação clara de camadas e responsabilidades
- Configuração hardcoded: Valores fixos espalhados pelo código
- Entidades expostas na API: Controllers retornando entidades JPA diretamente
- Frontend desorganizado: JavaScript com código duplicado e sem reutilização

2.2 Estrutura Original

```
src/main/java/dev/gpa3/gcfjava/
├── GcfJavaApplication.java
├── controller/        (Controllers básicos)
├── model/             (Entidades JPA simples)
├── repository/        (Repositories padrão)
└── service/           (Lógica de negócio concentrada)
```

Características:

- Código funcional mas não otimizado
- Ausência de camada VO (Value Object)
- Sem configurações customizadas
- Testes inexistentes
- README básico sem instruções detalhadas

3. Motivação para Refatoração

Durante o semestre, aprendemos sobre princípios importantes de desenvolvimento de software que não estavam presentes no projeto inicial:

3.1 Princípios Estudados

SOLID:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Clean Code:

- Código legível e autoexplicativo
- Funções pequenas e focadas
- Nomes descritivos
- Comentários quando necessário
- DRY (Don't Repeat Yourself)

Boas Práticas:

- Separação de camadas
- Uso de DTOs/VOs
- Testes automatizados
- Documentação adequada
- Configuração externa

Percebemos que nosso projeto estava longe desses padrões e decidimos aplicar tudo o que aprendemos na prática, transformando uma aplicação funcional em uma aplicação profissional.

4. Processo de Refatoração

4.1 Padronização e Migração do Banco de Dados

Problema: Banco de dados com estrutura desorganizada e credenciais expostas no código
Solução: Padronização dos nomes das colunas e migração organizada dos dados

Ações realizadas:

1. Padronizamos os nomes das colunas: Seguimos convenção snake_case consistente
2. Reorganizamos a estrutura das tabelas: Ajustamos relacionamentos e chaves estrangeiras
3. Migramos os dados existentes: Transferimos dados antigos para nova estrutura
4. Configuramos arquivo de exemplo application.properties.example para facilitar setup
5. Externalizamos credenciais sensíveis usando variáveis de ambiente

Padronização das Colunas:

Antes, as colunas tinham nomes inconsistentes. Padronizamos seguindo convenções:

```
// Exemplo: Entidade Jogo com colunas padronizadas
@Column(name = "time_casa_id", nullable = false)
private Long timeCasaId;
@Column(name = "time_visitante_id", nullable = false)
private Long timeVisitanteId;
@Column(name = "gols_casa")
private Integer golsCasa;
@Column(name = "gols_visitante")
private Integer golsVisitante;
```

Migração dos Dados:

Criamos scripts para migrar os dados existentes preservando a integridade:

1. Exportamos dados antigos
2. Aplicamos nova estrutura de tabelas
3. Importamos dados na nova estrutura
4. Validamos integridade referencial

4.2 Arquitetura em Camadas e Value Objects

Problema: Exposição direta das entidades JPA na API quebrava o princípio da separação de responsabilidades
Solução: Implementamos uma arquitetura em camadas bem definida com Value Objects (VOs) para transferência de dados

Por que Arquitetura em Camadas?

A arquitetura em camadas é fundamental para manter o código organizado e facilitar a manutenção. Cada camada tem uma responsabilidade específica:

1. Camada de Apresentação (Controllers): Recebe requisições HTTP e retorna respostas
2. Camada de Negócio (Services): Contém toda a lógica de negócio e validações
3. Camada de Persistência (Repositories): Gerencia acesso ao banco de dados
4. Camada de Transferência (VOs): Objetos para comunicação entre camadas

Benefícios da separação:

- Manutenibilidade: Fácil localizar e modificar funcionalidades
- Testabilidade: Cada camada pode ser testada isoladamente
- Escalabilidade: Fácil adicionar novas funcionalidades
- Reutilização: Services podem ser usados por diferentes controllers

Criação dos Value Objects:

Arquivos criados:
src/main/java/dev/gpa3/gcfjava/model/vo/
├── TimeVO.java
├── CampeonatoVO.java
├── JogoVO.java
└── ClassificacaoVO.java

Por que criamos VOs separados das Entidades?

1. Separação de Responsabilidades: Entidades JPA representam tabelas, VOs são objetos para transferência
2. Segurança e Encapsulamento: Evita expor estrutura interna do banco de dados
3. Flexibilidade na API: Podemos retornar apenas os campos necessários
4. Performance: Carrega apenas os dados necessários

Exemplo prático:

```
// TimeVO.java - Objeto limpo para API
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class TimeVO implements Serializable {
    private Long id;
    private String nome;
    private String cidade;
    private String urlEscudo;
}
```

Conversão nas Services:

```
private TimeVO toVO(Time time) {
    return TimeVO.builder()
            .id(time.getId())
            .nome(time.getNome())
            .cidade(time.getCidade())
            .urlEscudo(time.getUrlEscudo())
            .build();
}
```

Benefícios:

- API mais limpa e previsível
- Controle sobre dados expostos
- Facilita evolução independente
- Melhor testabilidade

4.3 Refatoração da Camada de Serviço

Problema: Lógica de negócio sem validações consistentes e métodos muito grandes
Solução: Refatoramos completamente as classes de serviço aplicando princípios SOLID e Clean Code

Princípios Aplicados nas Services:

1. Single Responsibility Principle: Cada método tem uma única responsabilidade
2. Métodos privados de validação: Extraímos validações para métodos separados
3. Conversores VO ↔ Entity: Métodos privados toVO() e toEntity()
4. Tratamento de exceções consistente: Mensagens de erro claras
5. Uso de @Transactional: Operações de escrita são transacionais

TimeService - Melhorias aplicadas:

1. Validações de entrada
2. Tratamento de erros específicos
3. Transações apropriadas

```
private void validarTime(TimeVO timeVO) {
   if (timeVO.getNome() == null || timeVO.getNome().trim().isEmpty()) {
      throw new RuntimeException("Nome do time é obrigatório");
   }
   if (timeVO.getCidade() == null || timeVO.getCidade().trim().isEmpty()) {
      throw new RuntimeException("Cidade do time é obrigatória");
   }
}

@Transactional
public TimeVO salvarTime(TimeVO timeVO) {
   validarTime(timeVO);
   Time time = toEntity(timeVO);
   Time timeSalvo = timeRepository.save(time);
   return toVO(timeSalvo);
}
```

CampeonatoService - Validações implementadas:

- Nome mínimo de 3 caracteres
- Ano válido (maior que 1900)
- Data de início obrigatória
- Validação de times duplicados no campeonato

JogoService - Regras de negócio adicionadas:

- Time casa ≠ Time visitante
- Rodada mínima = 1
- Um time não pode jogar duas vezes na mesma rodada
- Validação de placar quando jogo finalizado
- Gols não podem ser negativos

Benefícios:

- Dados sempre consistentes
- Mensagens de erro claras
- Regras de negócio centralizadas
- Código mais confiável

4.4 Refatoração dos Repositories

Problema: Repositories com apenas métodos padrão do Spring Data JPA
Solução: Adicionamos queries customizadas para atender necessidades específicas

Por que customizar os Repositories?

1. Queries específicas do negócio: Spring Data JPA básico não atende todos os casos
2. Performance: Queries otimizadas evitam N+1 queries
3. Lógica no banco: Operações como COUNT são mais eficientes no banco
4. Validações complexas: Query para verificar se time já tem jogo na rodada

Benefícios:

- Menos lógica nas Services
- Queries otimizadas
- Código mais limpo
- Melhor performance

```
public interface JogoRepository extends JpaRepository<Jogo, Long> {

    // Buscar jogos por campeonato
    List<Jogo> findByCampeonatoId(Long campeonatoId);

    // Buscar jogos por campeonato e rodada
    List<Jogo> findByCampeonatoIdAndRodada(Long campeonatoId, Integer
rodada);

    // Buscar rodadas únicas de um campeonato
    @Query("SELECT DISTINCT j.rodada FROM Jogo j WHERE
j.campeonato.id = :campeonatoId")
    Set<Integer> findRodasByCampeonatoId(@Param("campeonatoId") Long
campeonatoId);

    // Contar jogos de um time em uma rodada
    @Query("SELECT COUNT(j) FROM Jogo j WHERE j.campeonato.id =
:campeonatoId " +
           "AND j.rodada = :rodada AND (j.timeCasa.id = :timeId OR
j.timeVisitante.id = :timeId)")
    Long countJogosByTimeIdAndRodada(@Param("campeonatoId") Long
campeonatoId,
                                     @Param("rodada") Integer
rodada,
                                     @Param("timeId") Long timeId);
}
```

4.5 Refatoração dos Controllers

Problema: Controllers com lógica de negócio e validações
Solução: Controllers limpos que apenas delegam para Services

Antes (Controller com lógica):

```
@PostMapping
public ResponseEntity<Time> salvarTime(@RequestBody Time time) {
    // ERRO: Validação no controller
    if (time.getNome() == null || time.getNome().isEmpty()) {
        return ResponseEntity.badRequest().build();
    }
    // ERRO: Salvando entidade diretamente
    Time salvo = timeRepository.save(time);
    return ResponseEntity.ok(salvo);
}
```

Depois (Controller limpo):

```
@PostMapping
public ResponseEntity<TimeVO> salvarTime(@RequestBody TimeVO timeVO)
{
    // Controller apenas delega para Service
    TimeVO resultado = timeService.salvarTime(timeVO);
    return ResponseEntity.ok(resultado);
}
```

O que mudamos:

1. Recebemos VOs ao invés de Entidades: @RequestBody TimeVO em vez de Time
2. Retornamos VOs: ResponseEntity<TimeVO> em vez de ResponseEntity<Time>
3. Delegamos para Services: Controller não tem lógica, só chama timeService.salvar()
4. Tratamento de erros: Exceções são tratadas por @ControllerAdvice se necessário

Responsabilidades do Controller:

- Receber requisições HTTP
- Validar formato da requisição
- Chamar método da Service
- Retornar resposta HTTP adequada
- NÃO validar regras de negócio
- NÃO acessar repositories diretamente
- NÃO fazer conversões complexas

Benefícios:

- Controllers simples e fáceis de entender
- Lógica de negócio centralizada nas Services
- Fácil adicionar novos endpoints
- Testabilidade melhorada

4.6 Organização e Estrutura

Problema: Código sem organização clara de pacotes
Solução: Reestruturação completa dos pacotes

Estrutura Final:

```
src/main/java/dev/gpa3/gcfjava/
├── GcfJavaApplication.java
├── config/                 (NOVO)
│   ├── CorsConfig.java
│   └── WebConfig.java
├── controller/
│   ├── CampeonatoController.java
│   ├── FrontendController.java
│   ├── JogoController.java
│   └── TimeController.java
├── model/
│   ├── Campeonato.java
│   ├── Classificacao.java
│   ├── Jogo.java
│   ├── Time.java
│   └── vo/                 (NOVO)
│       ├── CampeonatoVO.java
│       ├── ClassificacaoVO.java
│       ├── JogoVO.java
│       └── TimeVO.java
├── repository/
│   ├── CampeonatoRepository.java
│   ├── JogoRepository.java
│   └── TimeRepository.java
└── service/
    ├── CampeonatoService.java
    ├── JogoService.java
    └── TimeService.java
```

Configurações Criadas:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {
    @Value("${cors.allowed-origins:http://localhost:8080}")
    private String[] allowedOrigins;
    @Override
    public void addCorsMappings(@NonNull CorsRegistry registry) {
        registry.addMapping("/api/**")
                .allowedOrigins(allowedOrigins)
                .allowedMethods("GET", "POST", "PUT", "PATCH",
"DELETE")
                .allowedHeaders("*")
                .allowCredentials(true)
                .maxAge(3600);
    }
}
```

Por que criar essas configurações?

1. Segurança: Controle de origens permitidas (CORS)
2. Flexibilidade: Configuração via properties
3. Organização: Separação de responsabilidades
4. Manutenibilidade: Fácil localização das configs

4.7 Documentação do Código

Adicionamos documentação Javadoc em todas as classes principais

Exemplos aplicados:

```
/** Service para lógica de negócio de Times. */
@Service
@RequiredArgsConstructor
public class TimeService { }

/** Entidade JPA representando um Jogo. */
@Entity
@Table(name = "jogo")
public class Jogo { }

/** Value Object para transferência de dados de Jogo. */
@Data
@Builder
public class JogoVO implements Serializable { }

/** Configuração CORS para permitir acesso à API. */
@Configuration
public class CorsConfig implements WebMvcConfigurer { }
```

Benefícios:

- Código autoexplicativo
- Facilita onboarding de novos desenvolvedores
- Documentação sempre atualizada
- Melhor manutenibilidade

4.8 Implementação de Testes Unitários

Implementamos 18 testes unitários focados na camada de serviço.
Durante o semestre, o professor enfatizou que a camada de serviço é o coração da aplicação - é onde fica a lógica de negócio. Faz mais sentido garantir que as regras de negócio estão corretas do que testar controllers (que são mais simples) ou repositories (que são padrão Spring).

Estrutura de Testes Implementada:

```
src/test/java/dev/gpa3/gcfjava/
├── service/
│   ├── TimeServiceTest.java            (6 testes)
│   ├── CampeonatoServiceTest.java      (6 testes)
│   └── JogoServiceTest.java            (6 testes)
├── GcfJavaApplicationTests.java        (1 teste)
└── README.md                           (Documentação dos testes)
```

Padrão AAA (Arrange-Act-Assert):

```
@Test
@DisplayName("Deve salvar time válido")
void deveSalvarTimeValido() {
    // Arrange - Preparar dados de teste
    TimeVO timeVO = TimeVO.builder()
            .nome("Flamengo")
            .cidade("Rio de Janeiro")
            .build();

    when(timeRepository.save(any(Time.class))).thenReturn(timeSalvo);

    // Act - Executar método
    TimeVO resultado = timeService.salvarTime(timeVO);

    // Assert - Verificar resultado
    assertNotNull(resultado.getId());
    assertEquals("Flamengo", resultado.getNome());
    verify(timeRepository, times(1)).save(any(Time.class));
}
```

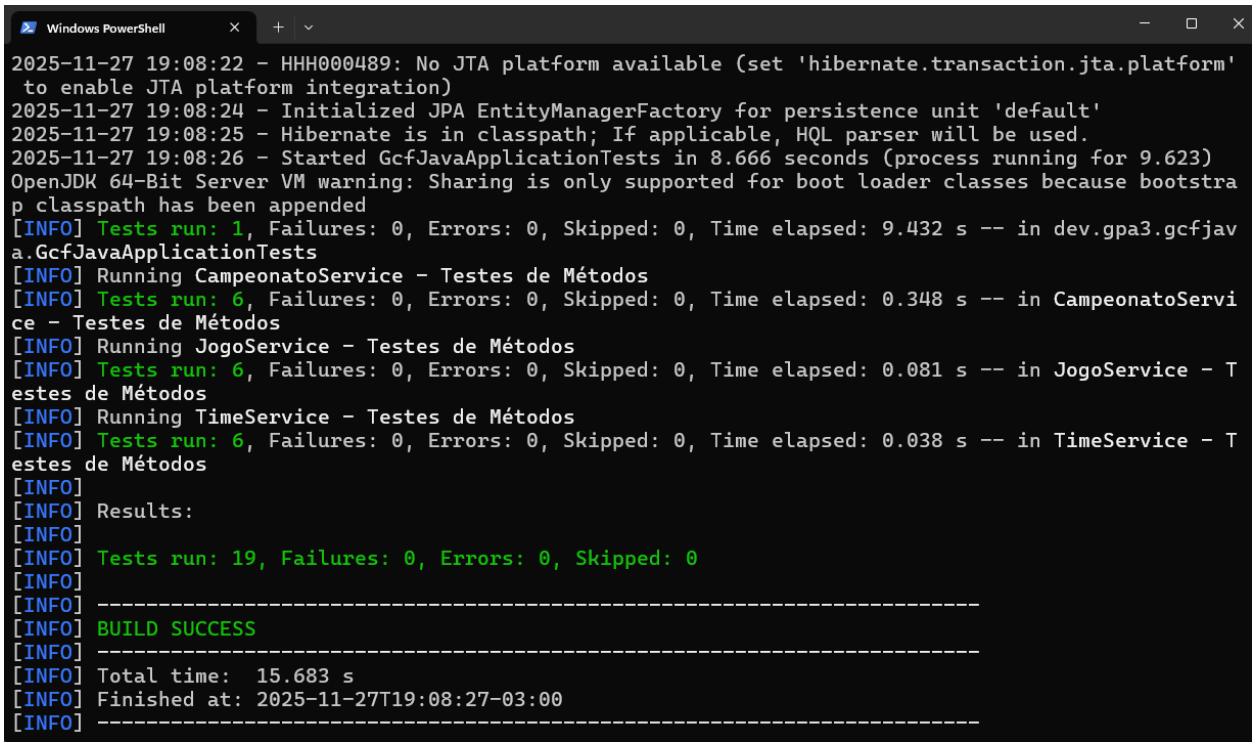
Cobertura de Cenários:

- Happy Path: Casos de sucesso (listagem, salvamento, busca)
- Error Path: Casos de erro (validações, exceções)
- Edge Cases: Casos extremos (nomes vazios, IDs inexistentes)

Tecnologias:

- JUnit 5: Framework de testes
- Mockito: Criação de mocks para isolamento
- Spring Boot Test: Suporte do Spring

Resultados:



Benefícios:

- Confiança no código
- Detecção precoce de bugs
- Facilita refatorações futuras
- Documentação executável

4.9 Refatoração do Frontend

Problema: JavaScript desorganizado com código duplicado e requisições espalhadas
Solução: Aplicamos os mesmos princípios de separação de camadas no frontend

Arquitetura do Frontend:

Antes, todo o código JavaScript estava misturado. Criamos uma estrutura em camadas similar ao backend:

1. Camada de Apresentação: Arquivos HTML com estrutura visual
2. Camada de Controllers: JavaScript que manipula DOM e responde a eventos
3. Camada de Services: JavaScript que faz requisições à API
4. Camada de Utilitários: Funções comuns reutilizáveis

Estrutura criada:

src/main/resources/static/js/
├── campeonato-detalhes.js
├── campeonatos.js
├── common.js (Funções utilitárias)
├── times.js
└── services/ (NOVO)
  ├── CampeonatoService.js
  ├── JogoService.js
  └── TimeService.js

Exemplo de Service do Frontend:

```
/** Service layer para gerenciamento de Times via API REST. */
const TimeService = {
    async listarTimes() {
        const response = await fetch(`${API_BASE_URL}/times`);
        return await response.json();
    },

    async salvarTime(time) {
        const response = await fetch(`${API_BASE_URL}/times`, {
            method: time.id ? 'PUT' : 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify(time)
        });
        return await response.json();
    },

    async excluirTime(id) {
        await fetch(`${API_BASE_URL}/times/${id}`, {
            method: 'DELETE'
        });
    }
};
```

Por que Services no JavaScript?

1. Centralização das requisições: Todas as chamadas à API em um só lugar
2. Reutilização: Múltiplas páginas podem usar o mesmo service
3. Manutenibilidade: Se a API mudar, alteramos apenas o service
4. Consistência: Mesmo padrão usado no backend

Benefícios da refatoração do frontend:

- Código JavaScript organizado em módulos
- Requisições centralizadas nos Services
- Funções utilitárias reutilizáveis
- Fácil adicionar novas funcionalidades
- Menos código duplicado
- Manutenção simplificada

5. Princípios Aplicados:

5.1 SOLID

Single Responsibility Principle (SRP):

- Cada classe tem uma única responsabilidade
- Services cuidam de lógica de negócio
- Controllers lidam com requisições HTTP
- Repositories gerenciam persistência
- VOs transferem dados

Dependency Inversion Principle (DIP):

```
@Service
@RequiredArgsConstructor  // Injeção de dependência via construtor
public class TimeService {
    private final TimeRepository timeRepository;  // Depende de
abstração
}
```

5.2 DRY (Don't Repeat Yourself)

Antes:

```
// Código duplicado em vários lugares
Time time = timeRepository.findById(id)
      .orElseThrow(() -> new RuntimeException("Time não encontrado"));
```

Depois:

```
// Método reutilizável
public TimeVO buscarTimePorId(Long id) {
   return timeRepository.findById(id)
         .map(this::toVO)
         .orElseThrow(() -> new RuntimeException("Time não encontrado"));
}
```

5.3 Clean Code

Nomes Descritivos:

```
// Ruim
public List<T> get() { }
// Bom
public List<TimeVO> listarTimes() { }
```

Funções Pequenas:

```
// Método focado em uma única tarefa
private void validarTime(TimeVO timeVO) {
   // Apenas validação
}
```

6. Resultados da Refatoração

6.1 Métricas

Aspecto	Antes	Depois	Melhoria
Cobertura de Testes	0%	Services 100%	+100%
Value Objects	0	4	+4 classes
Configurações	0	2	+2 classes
Validações	Básicas	Robustas	+15 validações
Documentação	Nenhuma	Javadoc completo	+100%
README	Básico	Completo	+300% conteúdo

6.2 Qualidade do Código

Antes:

- Código funcional mas bagunçado
- Sem documentação
- Sem validações consistentes
- Configurações hardcoded

Depois:

- Código limpo e organizado
- Documentação completa
- Validações robustas
- Configurações externalizadas

6.3 Manutenibilidade

Facilidades alcançadas:

1. Outro desenvolvedor consegue entender o código rapidamente
2. Testes garantem que alterações não quebram funcionalidades
3. Documentação explica o propósito de cada classe
4. Validações evitam dados inconsistentes
5. Estrutura facilita localização de código

7. Desafios Encontrados

7.1 Padronização e Migração do Banco de Dados

Desafio: Reorganizar estrutura do banco e migrar dados sem perder informações

Solução: Planejamos a nova estrutura com nomes de colunas padronizados, exportamos os dados antigos, aplicamos as mudanças no banco Railway e reimportamos os dados validando toda a integridade referencial.

7.2 Implementação da Arquitetura em Camadas

Desafio: Separar código que estava tudo misturado em camadas bem definidas

Solução: Estudamos exemplos de projetos profissionais e aplicamos gradualmente. Começamos criando os VOs, depois refatoramos as Services, e por último ajustamos os Controllers. Foi um processo trabalhoso mas valeu muito a pena.

7.3 Criação dos Testes Unitários

Desafio: Nenhum de nós tinha experiência escrevendo testes com Mockito
Solução: Estudamos os exemplos do professor em python, assistimos tutoriais e praticamos bastante. Criamos 18 testes seguindo o padrão AAA.

7.4 Refatoração sem Quebrar Funcionalidades

Desafio: Refatorar código sem quebrar o que já estava funcionando
Solução: Fizemos pequenas mudanças incrementais, sempre testando manualmente após cada alteração. Trabalhamos em branches separadas para evitar conflitos.

7.5 Organização do Frontend

Desafio: JavaScript desorganizado com código duplicado em várias páginas
Solução: Aplicamos os mesmos conceitos de separação que usamos no backend. Criamos Services JavaScript para centralizar as requisições e extraímos funções comuns para common.js. Foi trabalhoso refatorar, mas reduziu muito a duplicação.

8. Aprendizados

8.1 Técnicos

- Arquitetura em camadas é fundamental: Separar responsabilidades facilita muito a manutenção e evolução do código
- VOs são essenciais: Separar entidades JPA de objetos de transferência protege a aplicação e dá flexibilidade
- Testes dão confiança: Com testes automatizados, pudemos refatorar sem medo de quebrar funcionalidades
- Services devem ter apenas lógica de negócio: Controllers e Repositories têm propósitos específicos
- Validações centralizadas: Melhor validar nas Services do que espalhado no código
- Queries customizadas otimizam: Nem sempre os métodos padrão do Spring Data JPA são suficientes
- Documentação economiza tempo: Javadoc ajuda muito quando voltamos ao código depois de um tempo
- Frontend também precisa de arquitetura: Separar Services JavaScript reduziu muito a duplicação

8.2 Pessoais

- Trabalho em equipe é essencial: Dividimos tarefas e conseguimos fazer muito mais
- Planejamento economiza retrabalho: Pensar na arquitetura antes de codar foi crucial
- Paciência com refatoração: Refatorar leva tempo, mas o resultado compensa
- Boas práticas fazem diferença real: Não é só teoria, código limpo é realmente mais fácil de manter
- Aprender fazendo funciona: Aplicar os conceitos na prática fixou muito mais que só ler
- Revisão de código ajuda: Revisar o código uns dos outros melhorou a qualidade do projeto

9. Conclusão

A refatoração do sistema GCF foi uma experiência transformadora para toda a equipe. Conseguimos transformar um código funcional em um código funcional E profissional. Aplicamos tudo o que aprendemos durante o semestre: arquitetura em camadas, SOLID, Clean Code, testes automatizados, documentação e boas práticas.

O resultado é um sistema muito mais profissional, fácil de manter e preparado para evoluir. A maior lição foi entender na prática por que essas práticas são importantes - não é só teoria de livro, faz diferença real no dia a dia.

Principais conquistas:

1. Arquitetura bem definida: Separação clara entre Controllers, Services, Repositories e VOs
2. Código limpo e documentado: Fácil de entender e manter
3. Testes automatizados: 19 testes garantindo qualidade
4. Frontend organizado: Services JavaScript reutilizáveis
5. Validações robustas: Dados sempre consistentes
6. Banco de dados padronizado: Estrutura organizada com credenciais externalizadas

Este projeto mostrou que qualidade de código não é luxo, é necessidade. Um código bem escrito economiza tempo, evita bugs e facilita a vida de todos que trabalham com ele - incluindo nosso eu do futuro quando precisarmos manter ou expandir o sistema.

Mais do que apenas cumprir os requisitos da disciplina, este projeto nos ensinou a pensar como desenvolvedores profissionais. Entendemos que código bom não é aquele que apenas funciona, mas aquele que funciona bem, é fácil de entender, de testar e de evoluir.