Universidade Federal do Rio Grande do Sul - Instituto de Informática
Departamento de Informática Aplicada - INF01120 - Técnicas de Construção de Programas Prof. Karina Kohl

Trabalho Prático - Etapa 2: Implementação e teste

Artur Webber de Oliveira - 316091
Bruna Rosa Bragança de Lima - 588681
Elano Tavares do Nascimento - 588513
João Victor Serpa - 590536

1.2 Mudanças em relação à Etapa anterior

1. **Implementação de Camada de Persistência em Memória (Context e Repository):**
   - **Mudança:** Originalmente, o projeto previa a leitura de dados, mas não especificava detalhadamente como esses dados seriam gerenciados durante a execução. Foi introduzida a classe genérica `Repository<T>` e a classe `Context`.
   - **Justificativa:** Para simular o comportamento de um banco de dados em memória e facilitar as operações de *CRUD* (especialmente busca e inserção) utilizadas pela interface gráfica e pelos serviços, centralizando o acesso aos dados carregados via CSV.
2. **Refinamento do Modelo de Autenticação e Segurança:**
   - **Mudança:** A classe abstrata `Pessoa` recebeu um novo atributo `senhaHash`. Além disso, foi criada a interface `Autenticacao` e sua implementação `ServicoAutenticacao`.
   - **Justificativa:** A especificação original (RF-1) tratava do cadastro de forma abstrata. Para garantir a integridade e segurança mínima exigida, optou-se por não armazenar senhas em texto plano e isolar a lógica de login em um serviço dedicado, separando da interface de usuário.
3. **Ajuste nos Construtores das Entidades (Models):**
   - **Mudança:** As classes de domínio (`Filme`, `Serie`, `Jogo`, `Livro`, `Pessoa`) receberam sobrecarga de construtores.
   - **Justificativa:** Foi necessário distinguir a criação de novos objetos (onde o ID é gerado automaticamente via `UUID.randomUUID()`) da criação de objetos carregados via persistência (onde o ID já existe e provém dos arquivos CSV ou do banco em memória).
4. **Inclusão de Biblioteca de Logging:**
   - **Mudança:** Integração da biblioteca externa *tinylog* em substituição ao uso exclusivo de `System.out.println`.
   - **Justificativa:** Atender aos requisitos não funcionais de manutenibilidade e rastreabilidade (RNF-5), permitindo um monitoramento mais profissional do fluxo da aplicação e tratamento de erros sem poluir a saída padrão.
5. **Lógica Dinâmica de Carregamento de Séries:**
   - **Mudança:** A classe `CarregadorDeDados` implementou uma lógica robusta para vincular episódios às temporadas. Se um episódio referencia uma temporada ainda não existente, o sistema agora a cria automaticamente.

- **Justificativa:** Garantir a consistência dos dados relacionais complexos (Série -> Temporada -> Episódio) ao ler arquivos planos (CSV), evitando erros de referência nula durante a inicialização.

1.3 Implementação

O software "ArigóFlix" foi implementado utilizando a linguagem de programação Java (versão 17+), aderindo estritamente ao paradigma de Orientação a Objetos. A arquitetura do sistema foi organizada em camadas lógicas para separar responsabilidades: Modelo (*Models*), Serviço (*Services*) e Interface (*UI*).

Bibliotecas e Ferramentas

A implementação priorizou o uso das bibliotecas padrão do Java (**Java Standard Library**), garantindo portabilidade e reduzindo dependências. Foram amplamente utilizados os pacotes:

- **java.io**: Para manipulação de arquivos CSV e serialização de objetos (persistência).
- **java.util**: Para uso de coleções (List, Map, ArrayList, HashMap) e utilitários como `UUID`.
- **java.time**: Para manipulação moderna de datas (`LocalDate`).
- **javax.swing** e **java.awt**: Para construção da interface gráfica.

Como exceção e melhoria técnica (ver seção 1.2), foi incluída a **biblioteca externa Tinylog (versão 2.7.0)**. Esta biblioteca foi escolhida por ser leve, rápida e de fácil configuração estática, permitindo o registro de logs de operação e erros (`Logger.info`, `Logger.error`) sem a complexidade de frameworks maiores como Log4j.

Estrutura e Detalhamento das Classes

O código foi organizado nos seguintes pacotes principais:

1. Pacote main.models (Domínio)

Este pacote contém as classes que representam o núcleo do negócio. A implementação reflete o diagrama de classes com o uso de herança, abstração e polimorfismo.

- **Conteudo (Abstrata):** Classe base para todos os itens do catálogo.
  - ***Atributos:*** `UUID id`, `String titulo`, `LocalDate dataLanc`, `List<Avaliacao> avaliacoes`.
  - ***Método Principal:*** `calcularMediaPonderada()` - Implementa a lógica do RF-5, calculando a média das notas considerando o peso de cada avaliação. Lança `IllegalStateException` se a soma dos pesos for zero.
- **Filme, Livro, Jogo (Concretas):** Subclasses de `Conteudo` que adicionam atributos específicos (ex: diretor para Filme, editora para Livro, plataforma para Jogo).
- **Serie (Concreta):** Estende `Conteudo` e possui uma relação de composição com `Temporada`.
  - ***Atributos:*** `List<Temporada> temporadas`.

    - ***Métodos:*** `adicionarTemporada(Temporada t)`, `getTotalTemporadas()`.
- **Temporada** e **Episodio:** Classes auxiliares que estruturam a hierarquia de uma série. `Temporada` contém uma lista de `Episodio`.
- **Pessoa (Abstrata):** Classe base para usuários. Implementa `Serializable` para persistência.
    - ***Atributos:*** `UUID id`, `String nome`, `String email`, `String senhaHash`, `List<Avaliacao> listaAvaliacoes`.
    - ***Segurança:*** O construtor utiliza o método estático `hashSenha(String senha)` para armazenar apenas o hash SHA-256 da senha, nunca o texto plano.
- **Arigo** e **Critico (Concretas):** Estendem `Pessoa` e implementam a interface `Avaliador`.
    - ***Diferencial:*** A classe `Critico` define a constante `PESO_AVALIACAO = 2`, enquanto `Arigo` define como 1.
- **Avaliador (Interface):** Define o contrato `avaliar(Conteudo c, int nota, String comentario)`. Ambas as classes de usuário implementam este método, criando o objeto `Avaliacao` e vinculando-o ao conteúdo e ao histórico do usuário.

2. Pacote main.service (Lógica de Negócio e Persistência)

Responsável por orquestrar as operações que não pertencem unicamente a uma entidade.

- **CarregadorDeDados:** Classe especialista em I/O. Lê os arquivos CSV localizados em `src/resources/data`.
    - ***Lógica Complexa:*** Possui o método `carregarSeriesCompletas()`, que realiza o ***parsing*** de dois arquivos (`series.csv` e `episodios.csv`). Ela utiliza um `Map<Integer, Serie>` para vincular episódios às suas séries e cria objetos `Temporada` dinamicamente se eles ainda não existirem, garantindo a integridade da árvore de objetos.
- **Context** e **Repository<T>:** Implementam um padrão de repositório em memória.
    - **Repository<T>:** Classe genérica que fornece métodos CRUD (`add`, `getById`, `findAll`, `delete`) para qualquer entidade.
    - **Context:** Mantém as instâncias estáticas dos repositórios (pessoas, filmes, etc.) e gerencia a persistência global do estado da aplicação via serialização (save/load em arquivo `.dat`).
- **ServicoPromocao:** Contém a regra de negócio para promover um `Arigo` a `Critico`. Verifica se o usuário atingiu o `LIMITE_LIKES` em suas avaliações.

3. Pacote main.service.autenticacao (Segurança)

Isola a lógica de login do restante do sistema.

- **Autenticacao (Interface):** Define os métodos `registrar`, `autenticar`, `sair` e `getPessoaAutenticada`.
- **ServicoAutenticacao:** Implementação que valida credenciais. Verifica se o email existe no repositório e se o hash da senha fornecida coincide com o armazenado. Lança exceções de domínio como `CredenciaisInvalidasException`.

4. Pacote main.ui (Interface Gráfica)

Implementação visual utilizando Java Swing.

- **Main:** Ponto de entrada. Inicializa o `Context`, carrega dados iniciais se necessário, configura o `Autenticacao` e inicia a Thread do Swing.
- **Navegação:** Utiliza um `CardLayout` para alternar entre painéis (`TelaLogin`, `TelaCadastro`, `TelaInicial`, `TelaDetalhes`) na mesma janela, proporcionando uma experiência de SPA (Single Page Application) desktop.

1.4 Teste

Para garantir a qualidade e a conformidade do software com os requisitos funcionais, foram desenvolvidos testes unitários automatizados utilizando a estrutura **JUnit 5**. A estratégia de testes focou na validação da lógica de negócio (*Domain Layer*) e nas regras críticas do sistema, assegurando que as classes principais se comportem conforme o esperado antes da integração com a interface gráfica.

Cobertura e Detalhamento dos Casos de Teste

Os testes cobriram todas as classes do pacote `main.models`, além de serviços essenciais de lógica de dados. Abaixo, detalhamos os cenários validados:

- **Testes de Conteúdo (`FilmeTest`, `LivroTest`, `JogoTest`, `SerieTest`):**
  - **Cálculo de Média Ponderada (RF-5):** Validamos se o método `calcularMediaPonderada()` retorna o valor correto considerando os diferentes pesos dos usuários (Arigo = 1, Crítico = 2).
  - **Validação de Construtores:** Verificamos se o sistema impede a criação de conteúdos com dados inválidos (ex: título nulo, duração negativa), garantindo a robustez do modelo ("Defensive Programming").
  - **Hierarquia de Séries:** Em `SerieTest`, testamos especificamente a adição de temporadas e a correta contagem de episódios, assegurando que a estrutura de composição (Série -> Temporada -> Episódio) funcione.
- **Testes de Usuário (`PessoaTest`):**
  - **Regras de Avaliação (RF-3):** Testamos o método `avaliar()`. O sistema deve lançar uma exceção (`IllegalArgumentException`) caso a nota atribuída esteja fora do intervalo permitido (1 a 5).
  - **Vínculo de Avaliação:** Verificamos se, ao avaliar, a avaliação é corretamente adicionada à lista de avaliações do conteúdo e ao histórico do usuário.
- **Testes de Estrutura Auxiliar (`TemporadaTest`, `EpisodioTest`):**
  - Validamos a integridade dos objetos menores, como a impossibilidade de adicionar um episódio nulo a uma temporada e a validação de títulos e durações.
- **Teste de Integração de Dados (`CarregadorDeDadosTest`):**
  - Embora a especificação exigisse apenas testes de domínio, optamos por testar também a classe `CarregadorDeDados`.
  - **Lógica Complexa:** Validamos se o algoritmo de leitura de CSV consegue reconstruir corretamente a árvore de objetos de uma Série (vinculando episódios às temporadas corretas dinamicamente) e se diferencia

corretamente os tipos de usuários (`Arigo` vs `Critico`) ao ler do arquivo.

Relato de Experiência

A adoção de testes unitários durante o desenvolvimento provou-se fundamental para a estabilidade do ArigóFlix. Inicialmente, a equipe encontrou dificuldades em configurar o ambiente de testes (dependências do JUnit e *classpath*), mas, uma vez superada essa barreira, os benefícios foram imediatos.

Os testes nos permitiram identificar e corrigir erros lógicos precocemente, como um problema na fórmula da média ponderada e a questão da criação dinâmica de temporadas na leitura do CSV, que falhava silenciosamente antes da implementação dos testes. Além disso, a prática proporcionou segurança para refatorar o código (melhoria de nomes de variáveis e métodos) sem medo de quebrar funcionalidades já implementadas (*Regression Testing*). A experiência reforçou a importância de validar "pequenas partes" do software isoladamente antes de tentar executar o sistema completo.

1.5 Executável (aplicação, interface)

Nesta seção, o grupo deve apresentar detalhes sobre a **aplicação (executável)**, em particular apresentando as funcionalidades desenvolvidas com imagens do software em execução, mostrando suas interfaces e interações sobre ela. Apesar de não ser a prioridade do trabalho, é fundamental que haja pelo menos algum tipo de interface gráfica (mesmo que simples), evitando que a aplicação rode apenas por linha de comando, por exemplo.

O relatório deve incluir um "**manual de uso**" simplificado, mostrando detalhes de como executar o aplicativo, acompanhado de **imagens da interface** seguida de **explicações gerais de cada funcionalidade**. Além disso, deve haver uma **discussão em linhas gerais confrontando a interface atual com a prototipada na etapa anterior**.

1.6 Integração e configuração

- **Qual biblioteca foi escolhida e por quê?**
  - Escolhemos o tinylog 2.7.0 porque é leve, muito simples de implementar e já oferece o que precisamos para um projeto demonstrativo: níveis de log, saída em console e em arquivo, com configuração bem direta.

- **Quais foram as principais dificuldades na integração?**
  - As principais dificuldades foram montar corretamente o classpath (tanto no script local quanto no GitHub Actions) e garantir que os JARs do tinylog fossem carregados também na pipeline de testes, evitando erros de NoClassDefFoundError.

- **Como foi feita a configuração (arquivo tinylog.properties, etc.)?**
  - A configuração foi feita via arquivo tinylog.properties, definindo dois writers (console e arquivo app.log), o formato das mensagens (data, nível, classe, método, mensagem) e o nível mínimo de log (info). O tinylog lê esse arquivo automaticamente a partir do classpath.

1.7 Pontos de reuso

- **O que foi reutilizado?**
  - Reutilizamos a biblioteca tinylog (API de logging), uma configuração centralizada via tinylog.properties e um padrão de mensagem de log aplicado em várias camadas (models, serviços e interface gráfica).

- **Que benefícios foram percebidos (tempo, qualidade, clareza, manutenção)?**
  - Ganhamos mais clareza sobre o comportamento em tempo de execução, principalmente em erros de dados e carregamento de arquivos. Isso facilita a manutenção e acelera o diagnóstico de problemas, tanto durante o desenvolvimento quanto em execuções futuras.

1.8 Boas práticas

- **Como a equipe definiu o nível adequado de log?**
  - Usamos INFO para eventos normais e relevantes (inicialização, carregamento de dados, promoções), WARN para situações anômalas mas recuperáveis (entrada inválida, usuário sem critério para promoção) e ERROR para falhas que disparam exceções ou interrompem funcionalidades.

- **Que informações não devem ser logadas (segurança, privacidade)?**
  - Evitamos registrar dados sensíveis como senhas, tokens ou detalhes excessivos de identificação do usuário. Também tomamos cuidado para não logar informações que não contribuam para o diagnóstico, mas possam expor o comportamento interno desnecessariamente.

- **Como o log pode ajudar na manutenção ou teste do sistema?**
  - Os logs ajudam a enxergar a ordem das chamadas, confirmar se determinados caminhos foram executados e reproduzir cenários de erro. Isso complementa os testes automatizados e facilita a depuração, tanto em ambiente local quanto em execuções futuras do sistema.