PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul

Ciência da Computação

Fundamentos de Desenvolvimento de Software

Trabalho Final

Nomes: Eduardo Barbosa, Miguel dos Santos, Kamilla Borges e Pedro Riva.

Prof. Julio Henrique A P Machado

Turma: 33

25/11/2024

Introdução

Este projeto foi desenvolvido com base nos princípios da Arquitetura Limpa, garantindo uma separação clara entre as responsabilidades das camadas do sistema, facilitando a manutenção, extensibilidade e testabilidade. Além disso, os princípios SOLID e padrões de projeto foram aplicados para estruturar e modularizar as classes, promovendo um design robusto e escalável.

O que é a Arquitetura Limpa?

A Arquitetura Limpa, proposta por Robert C. Martin, é um conjunto de princípios para organização de sistemas que visa:

- Separação de responsabilidades: Cada camada do sistema deve ter uma responsabilidade específica, facilitando manutenção e testes.
- Independência de frameworks: O sistema não deve ser acoplado a frameworks ou tecnologias externas, permitindo alterações sem grandes impactos.
- Foco nas regras de negócio: A lógica central (regras de negócio) deve ser o núcleo do sistema, isolada das camadas externas, como banco de dados, interfaces de usuário e frameworks.
- Testabilidade: Por meio da separação de responsabilidades, é mais fácil testar as diferentes partes do sistema, tanto isoladamente quanto de forma integrada.

Como a Arquitetura Limpa foi aplicada no trabalho

- Princípios Adotados:
  - Camadas bem definidas: O trabalho é dividido em camadas específicas (domínio, aplicação, interface adaptadora e infraestrutura), cada uma com responsabilidades claras.
  - Independência de frameworks: As entidades e casos de uso não dependem diretamente de frameworks como Spring ou JPA, promovendo reutilização e facilidade de manutenção.
  - Isolamento das regras de negócio: As regras de negócio são centralizadas nas entidades do domínio e nos serviços, enquanto os controladores e repositórios cuidam das interações externas.
- Como o trabalho foi dividido:
  - Domínio: Centraliza as regras e conceitos principais do sistema (entidades e interfaces de repositórios).
  - Casos de Uso: Gerenciam o fluxo das operações e interações entre as camadas.
  - Interface Adaptadora: Faz a comunicação entre a interface externa (como APIs) e as camadas internas.
  - Testes: Validações robustas garantem que as funcionalidades atendam aos requisitos esperados.

Componentes do trabalho

1. Casos de Uso (UCs - Use Cases)

- Os casos de uso encapsulam a lógica de aplicação específica para atender aos requisitos do negócio. Representam funcionalidades específicas que o sistema deve realizar e se concentram em orquestrar operações entre as entidades do domínio e os serviços. Centralizam a lógica específica de cada funcionalidade do sistema, interagindo com os repositórios e serviços para realizar operações e garantir que os requisitos sejam atendidos.

- Casos de uso do programa:
  - AtualizaCustoAplicativoUC.java: Atualiza o custo mensal de um aplicativo no sistema.
  - CadastraAssinaturaUC.java: Realiza o registro de novas assinaturas, associando um cliente a um aplicativo.
  - ListaAssinaturasClienteUC.java: Lista todas as assinaturas ativas de um cliente específico.
  - ListaAssinaturasUC.java: Lista todas as assinaturas cadastradas no sistema.
  - ListaClientesPorAplicativoUC.java: Retorna todos os clientes que possuem assinaturas de um determinado aplicativo.
  - ListaTodosAplicativosUC.java: Lista todos os aplicativos disponíveis no sistema.
  - ListaTodosClientesUC.java: Lista todos os clientes cadastrados no sistema.
  - RegistrarPagamentoUC.java: Processa pagamentos de assinaturas, aplicando promoções e reativando assinaturas, quando necessário.
  - ValidaAssinaturaUC.java: Verifica a validade de uma assinatura, identificando se ela ainda está ativa ou expirada.

2. DTOs (Data Transfer Objects)

- Objetos projetados para transportar dados entre camadas do sistema. Simplificam a transferência de informações e evitam o uso direto de entidades do domínio. Facilitam a comunicação entre controladores, casos de uso e camadas externas, promovendo um design limpo e desacoplado.

- DTOs do programa:
  - AplicativoDTO.java: Representa as informações de um aplicativo (nome, custo mensal) a serem enviadas ou recebidas pela interface do usuário ou outros serviços.
  - AssinaturaDTO.java: Contém dados de uma assinatura, como cliente, aplicativo e vigência.
  - ClienteDTO.java: Representa informações de clientes, como nome, email e código.
  - CriarAssinaturaDTO.java: Estrutura os dados necessários para criar uma nova assinatura, incluindo identificadores de cliente e aplicativo.
  - CustoDTO.java: Transporta informações relacionadas ao custo de aplicativos, permitindo atualizações ou consultas específicas.
  - PagamentoDTO.java: Contém informações sobre pagamentos, incluindo valor pago, data de pagamento e detalhes sobre descontos ou promoções aplicadas.

3. Entidades Model (Domínio - Models)

- São as representações principais do domínio de negócio. Cada entidade reflete um conceito importante do sistema e contém as regras e atributos associados. Servem como base do domínio do sistema, centralizando as regras de negócio específicas de cada entidade.

- Entidades Model do programa:
    - AplicativoModel.java: Representa os aplicativos disponíveis, contendo atributos como código, nome e custo mensal.
    - AssinaturaModel.java: Reflete as assinaturas realizadas pelos clientes, incluindo informações sobre vigência, cliente associado e aplicativo relacionado.
    - ClienteModel.java: Representa os clientes cadastrados, com atributos como nome, email e código.
    - PagamentoModel.java: Gerencia informações sobre pagamentos realizados, como valor pago, data e detalhes de promoções aplicadas.
    - UsuarioModel.java: Representa os usuários do sistema, geralmente para fins de autenticação e autorização.

4. Repositórios (Repos)

- Interfaces que abstraem a lógica de acesso ao banco de dados, permitindo operações como criar, ler, atualizar e deletar (CRUD). Oferecem uma interface simples para acessar o banco de dados, promovendo o desacoplamento entre as camadas de negócio e de persistência.

- Repositórios do programa:
    - AplicativoRepository.java: Gerencia a persistência de aplicativos no banco de dados.
    - AssinaturaRepository.java: Controla a persistência de assinaturas, permitindo operações como criação, consulta e atualização.
    - ClienteRepository.java: Cuida da persistência de informações sobre clientes cadastrados.
    - PagamentoRepository.java: Lida com a persistência de dados de pagamentos realizados no sistema.
    - UsuarioRepository.java: Gerencia os dados relacionados aos usuários do sistema.

5. Serviços

- Classes que encapsulam a lógica de negócio compartilhada, centralizando operações reutilizáveis entre diferentes casos de uso. Garantem que a lógica de negócio seja consistente e isolada, permitindo que os casos de uso a reutilizem sem duplicação de código.

- Serviços do programa:

- ServicoDePagamentos.java: Lida com a lógica de processamento de pagamentos, incluindo cálculo de promoções, validação de valores pagos e extensão de vigências.
- ServicoDeAssinaturas.java: Gerencia operações relacionadas às assinaturas, como consulta de status ou listagem de assinaturas de um cliente.
- ServicoDeClientes.java: Centraliza operações sobre os clientes, como consulta ou registro de informações.
- ServicoDeAplicativos.java: Lida com os aplicativos, permitindo listar, atualizar custos e outras operações relacionadas.

6. Interface Adaptadora

- Essa camada faz a ponte entre o mundo externo (como APIs ou interfaces do usuário) e as camadas internas (domínio e casos de uso). Traduz as solicitações externas em comandos que o sistema interno pode processar, além de expor os resultados em um formato apropriado.

- Controladores do programa:
  - Controladores:
    - AssinaturaController.java: Gerencia requisições relacionadas às assinaturas, como criação e validação.
    - AssinaturaStatusController.java: Lida com endpoints para verificar o status de uma assinatura.
  - Repositórios Implementados:
    - AplicativoRepJPA.java, AssinaturaRepJPA.java, ClienteRepJPA.java, PagamentoRepJPA.java, UsuarioRepJPA.java: Implementações específicas dos repositórios, usando o JPA para gerenciar a persistência no banco de dados.
  - Entidades: Contém mapeamentos diretos entre as entidades do domínio e o banco de dados.

7. Testes

- Testes Unitários:
  - Localizados no pacote testesUnitarios, verificam a lógica de negócio de forma isolada.
  - Exemplo: ServicoDePagamentosTest.java valida o comportamento do sistema ao processar pagamentos.
- Testes de Integração:
  - Localizados no pacote testerIntegrados, testam a interação entre serviços e repositórios, garantindo que as camadas estejam bem integradas.
  - Exemplo: ServicoDePagamentosIntegrationTest.java verifica se as operações de pagamento funcionam corretamente com o banco de dados.
- Testes do Sistema:
  - Localizados no pacote adapter, esses testes validam o funcionamento dos endpoints da API. Eles simulam chamadas HTTP para garantir que os controladores retornem as respostas corretas e que o sistema lide adequadamente com diferentes cenários e erros.

- Exemplo: AssinaturaControllerSystemTest.java testa operações como listagem de clientes, aplicativos e assinaturas, além do cadastro de assinaturas e atualização de custos e a AssinaturaStatusControllerTest.java valida o registro de pagamentos e o status de assinaturas, verificando cenários de sucesso, valores inválidos e assinaturas inexistentes.

Conjunto de Testes Tabulados

Classes para Testes Unitários (Classes de Negócio):

As Classes de Negócio são componentes do sistema responsáveis por implementar a lógica central ou "regras de negócio" do domínio da aplicação. Elas encapsulam as operações e decisões que o sistema deve executar para atingir seus objetivos. No contexto de testes unitários, essas classes são os principais alvos porque representam a essência funcional do sistema.

ServicoDePagamentosTest: É responsável por validar a lógica central implementada na classe de negócios ServicoDePagamentos, que gerencia o processamento de pagamentos, promoções e revalidação de assinaturas.

Teste	Explicação
processarPagamento_Mensal_ComDescontoDe5Porcento	Verifica se, ao realizar um pagamento mensal de uma assinatura ativa, o sistema adiciona 30 dias de validade e calcula o estorno de 5% do valor pago.
processarPagamento_Anual_ComDescontoDe10Porcento	Valida se, ao realizar um pagamento anual de uma assinatura ativa, o sistema adiciona 1 ano de validade e calcula o estorno de 10% do valor pago.
processarPagamento_Reativacao_ComPagamentoMensal	Testa se, ao reativar uma assinatura cancelada com um pagamento mensal, o sistema adiciona 45 dias à validade sem calcular estorno.
processarPagamento_Reativacao_ComPagamentoAnual	Verifica se, ao reativar uma assinatura cancelada com um pagamento anual, o sistema adiciona 1 ano de validade sem calcular estorno.
processarPagamento_ValorIncorreto	Valida se o sistema identifica um pagamento incorreto e retorna o valor pago como estorno.
processarPagamento_Reativacao_ComPagamentoAnual_CorrigeValidade	Verifica se, ao reativar uma assinatura cancelada com um pagamento anual, o sistema adiciona 1 ano à validade da assinatura corretamente e sem calcular estorno.

Classes para Testes Integrados (Classes que fazem uso de repositórios):

As classes para testes integrados são aquelas destinadas a verificar o funcionamento do sistema em um contexto mais amplo, testando a integração entre diferentes partes do código e o banco de dados. Em vez de testar apenas a lógica de negócio de forma isolada, como ocorre nos testes unitários, os testes integrados avaliam se os componentes estão interagindo corretamente.

No caso das classes que fazem uso de repositórios, os testes integrados se concentram em verificar se as operações realizadas no código (como salvar, buscar, atualizar ou deletar dados) funcionam corretamente com o banco de dados ou qualquer outro sistema de persistência.

AplicativoIntegrationTest: Testa as funcionalidades relacionadas ao repositório de aplicativos para verificar operações básicas de persistência e busca.

Teste	Explicação
salvarAplicativo_ComSucesso	Verifica se o repositório salva corretamente um aplicativo no banco de dados.
buscarAplicativo_PeloCodigo	Valida se o repositório consegue recuperar corretamente um aplicativo pelo seu identificador.

ClienteIntegrationTest: Testa as funcionalidades relacionadas ao repositório de clientes, garantindo a integridade das operações CRUD.

Teste	Explicação
salvarCliente_ComSucesso	Verifica se o repositório salva corretamente um cliente no banco de dados.
buscarCliente_PeloCodigo	Valida se o repositório consegue recuperar corretamente um cliente pelo seu identificador.

ServicoDePagamentosIntegrationTest: Integra a lógica de negócio da classe ServicoDePagamentos com o repositório de assinaturas, verificando o funcionamento completo de cenários de pagamento.

Teste	Explicação
processarPagamento_Sucesso	Testa se um pagamento válido atualiza corretamente a validade da assinatura e registra o pagamento no sistema.
processarPagamento_ValorIncorreto	Valida se um pagamento com valor incorreto não ativa promoções e retorna corretamente os dados esperados.

Testes do Sistema (Endpoints de controle):

Os testes de controle do sistema verificam o funcionamento dos endpoints expostos pelos controladores. Eles simulam interações com a API para garantir que as respostas sejam corretas e que os fluxos principais do sistema funcionem como esperado.

AssinaturaControllerSystemTest: Esta classe é responsável por testar os endpoints do controlador de assinaturas, garantindo que os serviços expostos estejam funcionando corretamente e retornando os dados esperados.

Teste	Explicação
listarClientes_DeveRetornarListaDeClientes	Valida se o endpoint de listagem de clientes retorna uma lista com pelo menos um cliente, contendo informações básicas como nome.
listarAplicativos_DeveRetornarListaDeAplicativos	Verifica se o endpoint retorna corretamente uma lista de aplicativos cadastrados, com ao menos um registro presente.
cadastrarAssinatura_DeveRetornarNovaAssinatura	Garante que ao realizar uma requisição de cadastro de assinatura, o sistema retorna o código e o status "ATIVA" da nova assinatura.
atualizarCustoAplicativo_DeveRetornarAplicativoAtualizado	Verifica se a atualização do custo de um aplicativo retorna corretamente o novo valor configurado para o custo.

listarAssinaturasPorTipo_DeveRetornarListaDeAssinaturas	Testa se o endpoint de listagem de assinaturas por tipo (mensal ou anual) retorna uma lista válida com ao menos um registro contendo o status da assinatura.
listarAssinaturasCliente_DeveRetornarListaDeAssinaturasDoCliente	Valida se o endpoint retorna todas as assinaturas associadas a um cliente específico, verificando o identificador correto do cliente no retorno.
listarAssinaturasPorAplicativo_DeveRetornarListaDeAssinaturasDoAplicativo	Garante que o endpoint retorna uma lista de assinaturas associadas a um aplicativo específico, validando o identificador do aplicativo no retorno.

AssinaturaStatusControllerTest: Esta classe valida os endpoints relacionados ao status e ao registro de pagamentos de assinaturas. Os testes verificam condições de valor limite, particionamento e casos inválidos.

Teste	Explicação
registrarPagamento_PagamentoExato_DeveRetornarPagamentoOk	Verifica se o sistema processa corretamente um pagamento no valor exato do custo, retornando o status "PAGAMENTO_OK" e sem valores estornados.
registrarPagamento_ValorSuperior_DeveRetornarValorIncorreto	Valida se um pagamento superior ao custo da assinatura é tratado como incorreto, retornando o valor pago como estorno.
registrarPagamento_ValorInferior_DeveRetornarValorIncorreto	Garante que pagamentos com valores inferiores ao custo são identificados como incorretos, com o valor pago retornado como estorno.
verificarAssinaturaValida_AssinaturaAtiva_DeveRetornarTrue	Testa se o endpoint retorna "true" para uma assinatura que está ativa.
verificarAssinaturaValida_AssinaturaInativa_DeveRetornarFalse	Valida se o sistema retorna "false" ao verificar o status de uma assinatura que está inativa.

registrarPagamento_DataInvalida_DeveRetornarBadRequest	Testa o comportamento do sistema ao receber uma data inválida em um registro de pagamento, garantindo que o sistema responda com o status HTTP "BAD_REQUEST".
verificarAssinaturaValida_AssinaturaInexistente_DeveRetornarFalse	Valida se o sistema retorna "false" para assinaturas inexistentes ao consultar o status de validade.

Falhas observadas na execução de testes:

1.a A falha observada;
Erro
AssinaturaStatusControllerTest.registrarPagamento_DataInvalida_DeveRetornarBadRequest
expected: 400 BAD_REQUEST
but was: 500 INTERNAL_SERVER_ERROR
java.time.DateTimeException: Invalid date 'FEBRUARY 31'
Causa: O erro foi causado pelo programa não informar bad request ao informar uma data inválida, 21 de fevereiro.

1.b O nome do teste que a detectou;
AssinaturaStatusControllerTest.registrarPagamento_DataInvalida_DeveRetornarBadRequest

1.c De que maneira foi corrigido: Na Classe RegistrarPagamentoUC foi convertida a data para formato Date, para lançar uma exceção caso a data seja inválida. Utiliza os blocos Try, Catch e IllegalArgumentException. Na Classe Controller (AssinaturaStatusController) foi adicionado tratamento de exceção IllegalArgumentException que se a data for inválida irá retornar uma resposta HTTP 400 – bad request.

2.a A falha observada; org.opentest4j.AssertionFailedError:
Expecting actual:
  1701398400000L (Sat, 30 Nov 2024 00:00:00 GMT)
to be equal to:
  1732934400000L (Sat, 29 Nov 2025 00:00:00 GMT)
but was not.
A data deveria ter sido recalculada para 1 ano e não 45 dias.

2.b O nome do teste que a detectou;
void processarPagamento_Reativacao_ComPagamentoAnual_CorrigeValidade

2.c De que maneira foi corrigido: A correção foi implementada dentro do método calcularNovaValidade, responsável por definir a nova data de validade da assinatura com base no tipo de pagamento e na situação (reativação ou pagamento regular). Antes da correção, o código tratava pagamentos de reativação (independente de serem mensais ou anuais) com 45 dias de validade, ignorando o fato de que pagamentos anuais deveriam oferecer 1 ano de validade.