Prof. Evandro César Freiberger

Estudo de Caso Proposto para a Disciplina de Testes Automatizados de Software

Brasil

10 de novembro de 2023

Sumário

1 Introdução

Este documento tem o objetivo descrever e especificar o Estudo de Caso prático da disciplina de Testes Automatizados de Software do curso de Pós-Graduação *Lato Sensu* em Engenharia DevOps.

O estudo de caso tem como objetivo principal a implementação de testes automatizado para um sistema hipotético proposto. Para isso, serão fornecidos os seguintes elementos do software: documentos de especificação de requisitos, diagramas em UML, especificação de estrutura de dados das entidades do software, regras de negócio e código-fonte implementado em linguagem Java.

A partir dos elementos de software fornecidos, serão produzidos testes automatizados que cobrirão os níveis: testes unitários, testes de integração, testes de sistema e testes de performance.

Este documento está organizado nas seguintes seções, além desta introdução: 2 - Requisitos do software fornecido - que apresenta uma visão geral do sistema, o Backlog do produto, as especifições de entidades de dados e modelos UML; 3 - Modelo Arquitetural do Software Fornecido - apresenta o modelo arquitetural utilizado na implementação do software fornecido, descrevendo os elementos e as informações mais relevantes para o entendimento do software; 4 - Execução do Estudo de Caso - especifica o que deve ser implementado de testes automatizados no software fornecido; 5 - Entrega e Apresentação - especifica o que deve ser entregue, prazo, formas de entrega e critérios de avaliação.

2 Requisitos do Software Fornecido

Nesta seção serão apresentados os requisitos do software fornecido e que serão utilizados como base para o desenvolvimento do estudo de caso.

2.1 Visão Geral

Esta subseção descreve a visão geral de um sistema de gestão de vendas e entregas de refeições de um restaurante fictício. Para que os pedidos de refeições possam ser realizados é necessário uma série de informações iniciais que dão suporte ao processo de pedido, produção e entrega.

Devem existir funcionalidades no sistema que permitam manutenção do cadastro de produtos, manutenção de grupos alimentares e tipos de preparos dos alimentos. Os clientes terão acesso as funcionalidades que lhes permitirão gerir os dados do seu próprio perfil de usuário do sistema, por meio de um módulo *mobile* que ele pode baixar em seu celular. Perfis de usuário internos do sistema serão geridos pelo administrador do sistema, cadastrando e configurando suas respectivas credenciais.

Os pedidos são realizados diretamente pelo cliente, por meio do aplicativo *mobile*, onde este escolhe uma refeição a partir de um cardápio diário oferecido pelo restaurante. O cardápio é criado e preenchido pelo chefe de cozinha, contemplando mais de uma opção de grupos alimentares, tais como: proteínas, cereais, legumes, verduras. As porções de cada item do cardápio sempre serão oferecidas em unidades de 50 gramas e o cliente pode escolher qualquer combinação e quantidades de frações, para compor seu pedido, contidas no cardápio do dia.

Os pedidos realizados pelos clientes devem ser inseridos em uma fila cronológica com o status pendente. Essa fila será visualizada pelos agentes de produção, que escolhem o pedido da vez e muda seu estado para produção, indicando que o pedido está em produção. Quando o agente de produção termina a produção de um pedido, este mudará seu estado para pronto, liberando o pedido para a entrega.

O gerente de produção deve ter acesso a uma fila de pedidos organizados de forma cronológica de produção e que estejam com estado pronto. Assim, o gerente de produção organizará as entregas dos pedidos prontos, mudando seu estado para entrega, quando esses forem despachados. Os pedidos são despachados para entregadores cadastrados no sistema, para que haja um controle de qualidade e pontualidade das entregas. Quando o entregador finaliza a entrega, este deve sinalizar em módulo *mobile* de entrega que o pedido foi entregue. O cliente deve possuir uma funcionalidade em seu módulo *mobile* que lhe permite avaliar o pedido recebido, em questões relacionadas ao pedido (conteúdo, sabor, temperatura, embalagem, etc) e em relação à entrega.

O chefe de cozinha deve planejar e definir os cardápios diários do restaurante. No dia que um cardápio vai ser colocado em produção, o chefe de cozinha deve fazer um procedimento de retirada de unidades de estoque dos produtos, considerando que o controle de estoque será realizado por frações de 50 gramas. Esse procedimento subtrairá essas unidades do estoque de cada produto. Isso significa que o chefe de cozinha pode criar os cardápios com antecedência, mas deve realizar um procedimento no sistema que indicará que determinado cardápio será colocado em produção naquele dia, indicando a quantidade de unidades de cada item que será produzida. Isso fará com que as respectivas unidades sejam subtraídas dos estoques dos produtos.

O gerente de estoque deve consultar o estoque dos produtos e obter uma relação de produtos que estejam com estoque abaixo da margem de segurança, para que este possa realizar pedidos aos fornecedores. Quando uma compra de produtos é recebida pelo gerente de

estoque, este deverá registrar a entradas das unidades/frações adquiridas para que o estoque seja atualizado.

O gerente de estoque também registrará baixas de estoque de produtos decorrentes de motivos que inviabilizem a utilização dos produtos, tais como: vencimento, quebra, mal estado de conservação. Esse procedimento deverá dar baixa às respectivas quantidades de estoque, e registrar a data e motivo pelo qual essas quantidades foram baixadas.

O valor cobrado em um pedido será calculado pelo valor de custo de cada item inserido no pedido, acrescido da taxa de entrega, obtida de uma planilha de valores de custo de entrega por bairro. O custo de cada item será formado pelo valor de custo do item, mais o valor de preparo, conforme o tipo de preparo estabelecido no cardápio, além de um percentual de valores fixos que representam custos como: água, energia, mão de obra, impostos, etc.

O gerente-geral do restaurante precisa de uma série de consultas para auxiliar em suas tomas de decisões, tais como:

- Quantidade de refeições produzidas por dia e com totalização mensal;
- Quantidade de produtos descartados;
- Produtos mais consumidos na produção das refeições;
- Produtos menos consumidos na produção de refeições;
- Tempo médio entre o recebimento de pedidos e o término da produção;
- Tempo médio entre a finalização da produção de um pedido e sua entrega ao cliente;

Como visão inicial do sistema, espera-se ter um módulo *mobile* para que o cliente e o entregador usem para realizar o pedido e as entregas respectivamente. Também vislumbra-se um módulo *desktop* para ser usado em um monitor *touch screen* para ser usado na produção e um módulo Web para ser usado nas configurações e consultas.

2.2 Backlog do Software

Nesta subseção são apresentadas as Users Stories produzidas para o software fornecido, que será usado como base do estudo de caso.

US01 - Como Gerente de Produção, quero fazer a gestão do cadastro de produtos que são usados para produzir as refeições, para que esses possam ser inseridos como itens de futuros cardápios, produções e pedidos.

US02 - Como Gerente de Produção, quero fazer a gestão do cadastro de grupos alimentares, para que esses possam ser usados para categorizar os produtos.

US03 - Como Gerente de Produção, quero fazer a gestão do cadastro de tipos de preparo de alimentos, para que esses possam ser usados para caracterizar o custo de preparo de cada item de cardápio.

US04 - Como Chefe de Cozinha, quero fazer gestão de cardápios de refeições, para que estes sejam usados na produção de refeições.

US05 - Como Cliente, quero realizar a gestão dos dados do meu perfil no aplicativo, para que eu possa realizar pedidos de refeições.

US06 - Como Cliente, quero realizar um pedido de refeição a partir do cardápio do dia, para que eu possa receber a refeição no meu endereço de entrega.

US07 - Como Cliente, quero acompanhar o status do meu pedido, para saber em que situação está.

US08 - Como Agente de Produção, quero consultar os pedidos do dia em ordem cronológica de solicitação e status em Pendente, para que eu possa colocá-lo em produção.

US09 - Como Agente de Produção, quero mudar o status de um pedido de Produção para Pronto, para que possa ser escalado para entrega.

US10 - Como Gerente de Produção, quero consultar os pedidos do dia que estejam com status em Pronto, para que eu possa despachar para entrega.

US11 - Como Chefe de Cozinha, quero escolher um cardápio para entrar em produção no dia, para a retirada de unidades/frações de produtos do estoque.

US12 - Como Gerente de Estoque, quero consultar produtos que estejam abaixo do estoque de segurança, para que eu possa fazer pedido aos fornecedores.

US13 - Como Gerente de Estoque, quero registrar entrada de unidades/frações de produto ao estoque.

US14 - Como Gerente de Estoque, quero registrar a baixa de unidades/frações de produtos do estoque que serão descartados, para que eu possa manter o estoque de produtos atualizado.

US15 - Como Gerente-Geral, quero consultar a produção de refeições por dia com totalização mensal, para que eu possa fazer estimativas e projeções futuras.

US16 - Como Gerente-Geral, quero consultar as quantidades de produtos descartados por um período, para que eu possa tomar decisões para a diminuição dos descartes.

US17 - Como Gerente-Geral, quero consultar quais são os produtos que mais foram produzidos em unidades, para que possa criar estratégias de compras mais eficientes.

US18 - Como Gerente-Geral, quero consultar quais são os produtos que menos foram produzidos em unidades, para que possa analisar a possibilidade de deixar de produzir tais produtos.

US19 - Como Gerente-Geral, quero consultar o tempo médio entre o recebimento de um pedido do cliente e o término da produção, para que eu possa criar estimativas de tempo de produção.

US20 - Como Gerente-Geral, quero configurar um percentual de custo fixo de produção, para ser aplicado na composição dos valores dos pedidos produzidos.

US21 - Como Gerente-Geral, quero consultar o tempo médio entre o término da produção de uma refeição e a finalização da entrega ao cliente, para que eu possa criar estimativas de tempo de entrega.

US22 - Como Gerente-Geral, quero configurar uma tabela de custo de entrega por bairro, para que seja usado na composição do valor dos pedidos produzidos.

US23 - Como Administrador, quero gerir o cadastro dos entregadores de pedidos, para que as entregas sejam rastreáveis por entregador.

US24 - Como Administrador, quero gerir o cadastro dos colaboradores internos do restaurante, para estabelecer o perfil de usuário de cada papel a ser desempenhado.

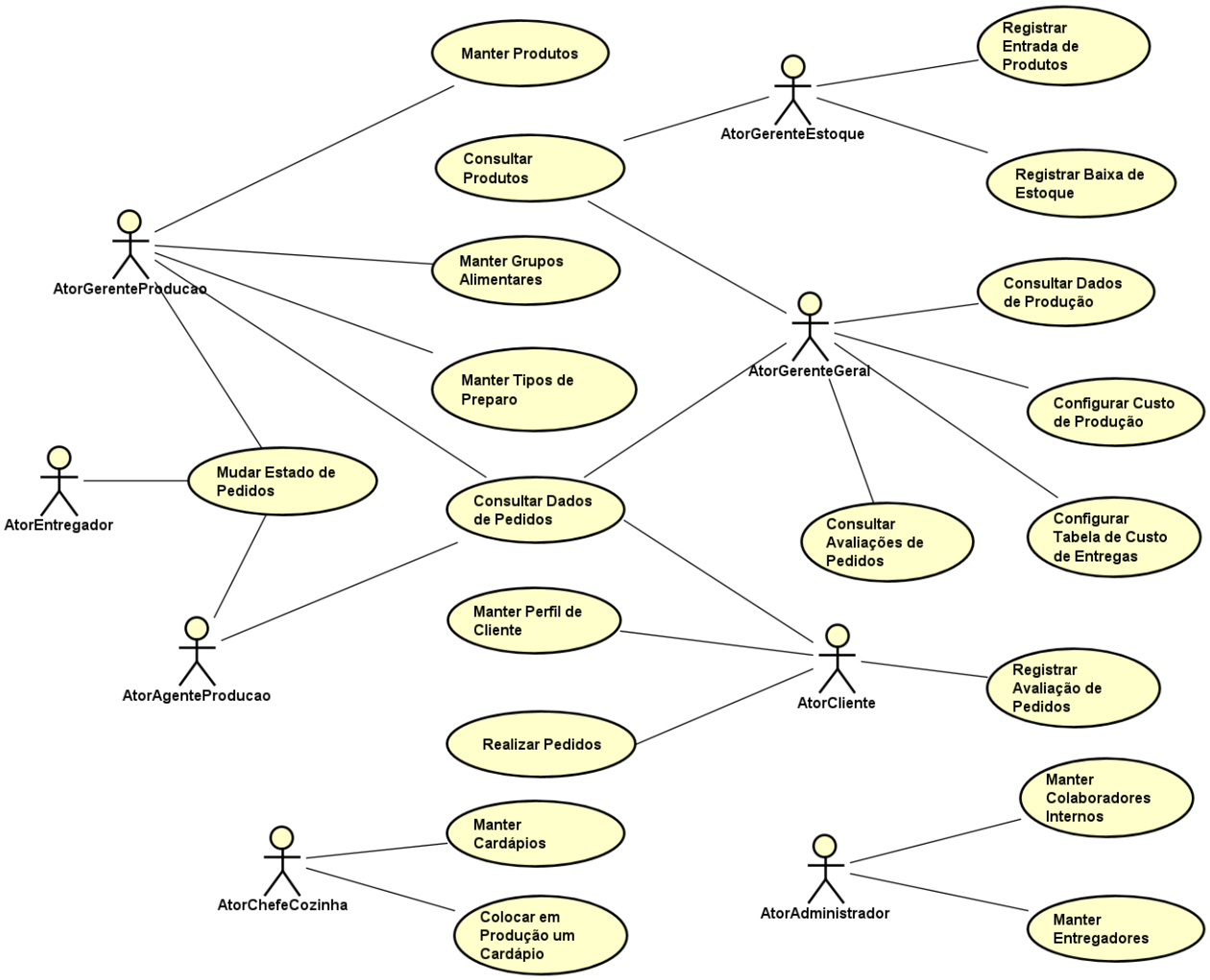
US25 - Como Entregador, quero registrar a finalização da entrega de um pedido, para que haja um controle de tempo de entrega de cada pedido.

US26 - Como Cliente, quero uma funcionalidade que permita o registro de avaliação do pedido recebido, para que o restaurante tenha dados para as avaliações de qualidade da produção e entrega das refeições fornecidas.

2.3 Diagrama de Casos de Uso

Nesta subseção é apresentado um diagrama de Casos de Uso do software fornecido, embora não foram usadas as especificações de casos de uso, o diagrama foi produzido para complementar, de forma gráfica, a relações entre os atores do sistema e as funcionalidades descritas nas Users Stories. Veja o diagrama na Figura 1.

Figura 1 – Diagrama de Casos de Uso



2.4 Refinamento das Entidades do Modelo de Domínio

Nesta subseção são apresentados os detalhes das entidades do domínio de problema. Para cada entidade são especificados os atributos, tipo dos atributos, formato, validação e a informação se o atributo é chave primário ou chave estrangeira. Além dessas informações também foi especificado para cada entidade as regras de negócio a ela associada.

Nome da Entidade: Cardapio
Nome da Tabela: cardapio

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
nome	String (50)	-	Não pode ser nulo	
descricao	String (250)	-		
listaPreparoProduto	PreparoProduto[]	-	Não pode ser vazia	FK (Mx1)

Regras de Negócio:

- Não deve existir mais de um cardápio com o mesmo nome
- Um cardápio válido deve ter, pelo menos, um item vinculado

Nome da Entidade: Cliente
Nome da Tabela: cliente

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
nome	String	-	Não pode ser nulo e deve ser menor três caracteres	
RG	String	-	Não pode ser nulo e nem vazio	
CPF	String	-	Não pode ser nulo e pelo menos 11 caracteres	
telefone	String	-	Não pode ser nulo e pelo menos 8 caracteres	
logradouro	String	-	Não pode ser nulo e nem vazio	
numero	String	-	Não pode ser nulo e nem vazio	
bairro	String	-	Não pode ser nulo e nem vazio	
pontoReferencia	String	-	Não pode ser nulo	

Regras de Negócio:

- Não pode existir dois clientes com o mesmo CPF
- Não pode ser excluído, caso tenha pedidos em seu nome

Nome da Entidade: Entregador
Nome da Tabela: entregador

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
nome	String	-	Não pode ser nulo e deve ser menor três caracteres	
RG	String	-	Não pode ser nulo e nem vazio	
CPF	String	-	Não pode ser nulo e pelo menos 11 caracteres	
telefone	String	-	Não pode ser nulo e pelo menos 8 caracteres	

Regras de Negócio:

- Não pode existir dois entregadores com o mesmo CPF
- Não pode ser excluído, caso tenha pedidos entregues por ele

Nome da Entidade: GrupoAlimentar
Nome da Tabela: grupo_alimentar

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
nome	Caractere (40)	Letras	Tamanho >= 3 caracteres	

Regras de Negócio:

- Não deve existir dois grupos alimentares com o mesmo nome.
- Não deve ser excluído um grupo referenciado por algum produto.

Nome da Entidade: ItemOrdemProducao
Nome da Tabela: item_ordem_producao

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
quantidadePorcao	Inteiro	-	> 0	
preparoProduto	PreparoProduto	-	Não pode ser nulo	FK (Mx1)

Regras de Negócio:

- Deve ser excluído juntamente a OrdemProducao

Nome da Entidade: ItemPedido
Nome da Tabela: item_pedido

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
quantidadePorcao	Inteiro	-	> 0	
preparoProduto	PreparoProduto	-	Não pode ser nulo	FK (Mx1)

Regras de Negócio:

- Deve ser excluído juntamente ao Pedido

Nome da Entidade: OrdemProducao
Nome da Tabela: ordem_producao

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
dataProducao	Data	-	Não pode ser nulo e deve ser menor ou igual a data corrente	
cardapio	Cardapio		Não pode ser nulo	FK(Mx1)
listaItens	ItemOrdemProducao[]	-	Não pode ser nulo	FK (1xM)
estado	EstadoOrdemProducao		Valores: REGISTRADA, PROCESSADA	

Regras de Negócio:

- Inicia no estado REGISTRADA
- Processamento muda o estado para PROCESSADA e retira as respectivas quantidades de unidades do estoque dos produtos relacionados.
- Não pode ser excluída depois de processada.

Nome da Entidade: Pedido
Nome da Tabela: pedido

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
dataPedido	Data	-	Não pode ser nulo e deve ser igual a data corrente	
horaPedido	Hora (time)		Não pode ser nulo e deve ser menor que a hora corrente	
cliente	Cliente	-	Não pode ser nulo	FK(Mx1)
horaProducao	Hora (time)		Se não for nulo, deve ser menor que a hora do pedido	
horaPronto	Hora (time)		Se não for nulo, deve ser menor que a hora de	

			producao	
horaEntrega	Hora (time)		Se não for nulo, deve ser menor que a hora de pronto	
horaFinalizado	Hora (time)		Se não for nulo, deve ser menor que a hora de entrega	
listaItens	ItemPedido[]	-	Não pode ser nulo	FK (1xM)
entregador	Entregador		Nos estados ENTREGA E CONCLUIDO, não pode ser nulo	FK (Mx1)
estado	EstadoOrdemProducao		Valores: REGISTRADO, PRODUCAO, PRONTO, ENTREGA, CONCLUIDO	

Regras de Negócio:

- Inicia no estado REGISTRADO
- Ir para produção muda o estado para PRODUCAO
- Término da produção muda o estado para PRONTO
- Sair para entrega muda o estado para ENTREGA
- Finalizar a entrega muda o estado para CONCLUIDO
- Não pode ser excluído depois que estiver no estado PRODUCAO

Nome da Entidade: PreparoProduto
Nome da Tabela: preparo_produto

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
produto	Produto	-	Não pode ser nulo	FK (Mx1)
tipoPreparo	TipoPreparo	-	Não pode ser nulo	FK (Mx1)
tempoPreparo	Inteiro	-	> 0	
valorPreparo	Float	-	> 0	

Regras de Negócio:

- Não deve existir mais de um PreparoProduto (Produto/TipoPreparo).

Nome da Entidade: Produto
Nome da Tabela: produto

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro		> 0	PK
nome	Caractere (50)		Tamanho > 3 caracteres	
custoUnidade	Float		> 0	
valorEnergetico	Inteiro		>= 0	
estoque	Inteiro		>= 0	
grupoAlimentar	Inteiro		Não nulo	FK (Mx1)

Regras de Negócio:

- Não deve existir dois produtos com o mesmo nome.

Nome da Entidade: TipoPreparo
Nome da Tabela: tipo_preparo

Campo	Tipo	Formato	Validação	Chave
codigo	Inteiro	-	> 0	PK
descricao	Caractere (50)	Letras	Tamanho >= 3 caracteres	

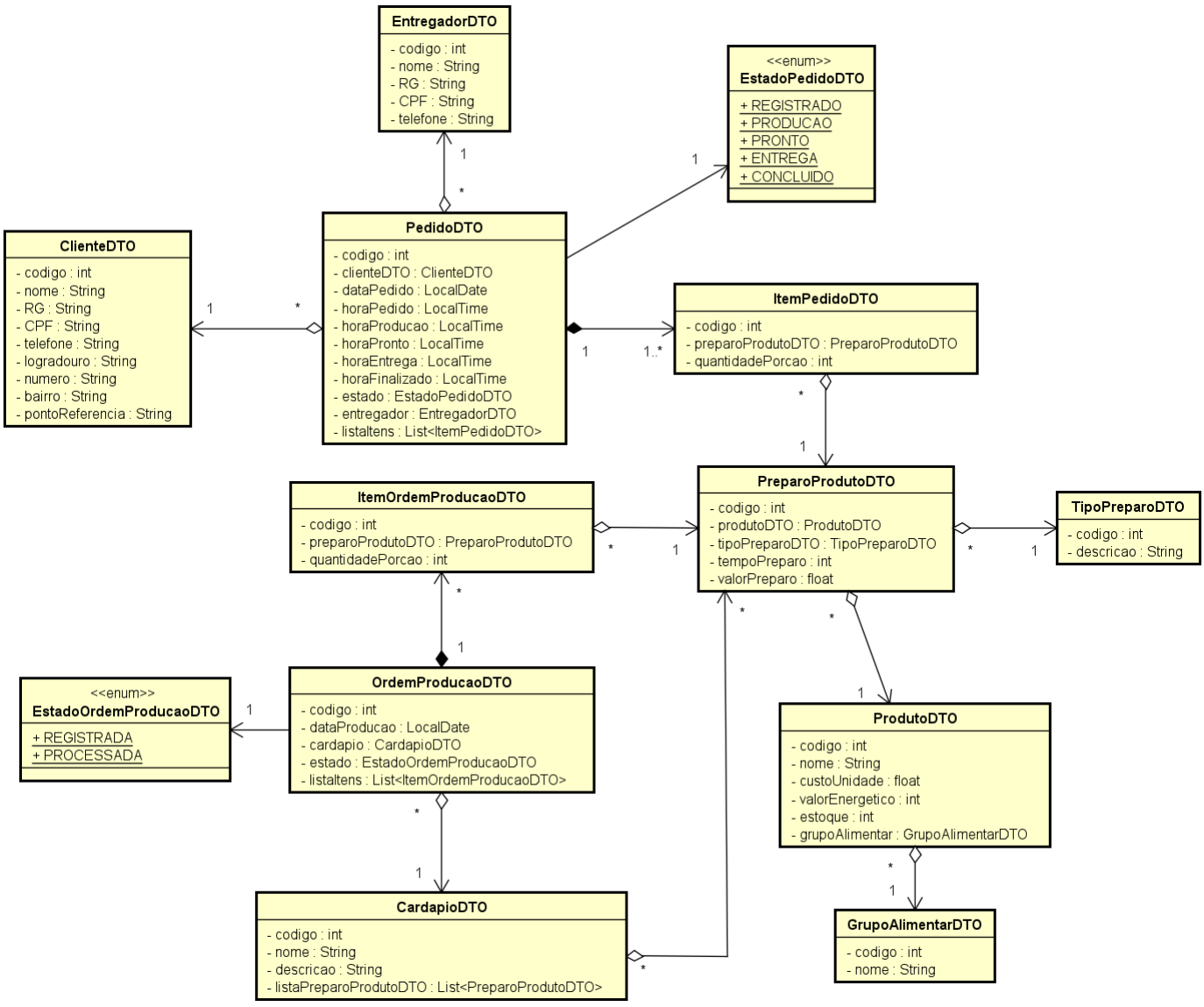
Regras de Negócio:

- Não deve existir mais de um TipoPreparo com a mesma descrição.
- Não deve ser excluído um TipoPreparo referenciado por algum PreparoProduto.

2.5 Entidades do Modelo de Domínio

A Figura 2 apresenta um modelo de classes que representam o modelo de domínio do software fornecido. Também é possível obter as relações entre as entidades e suas respectivas multiplicidades.

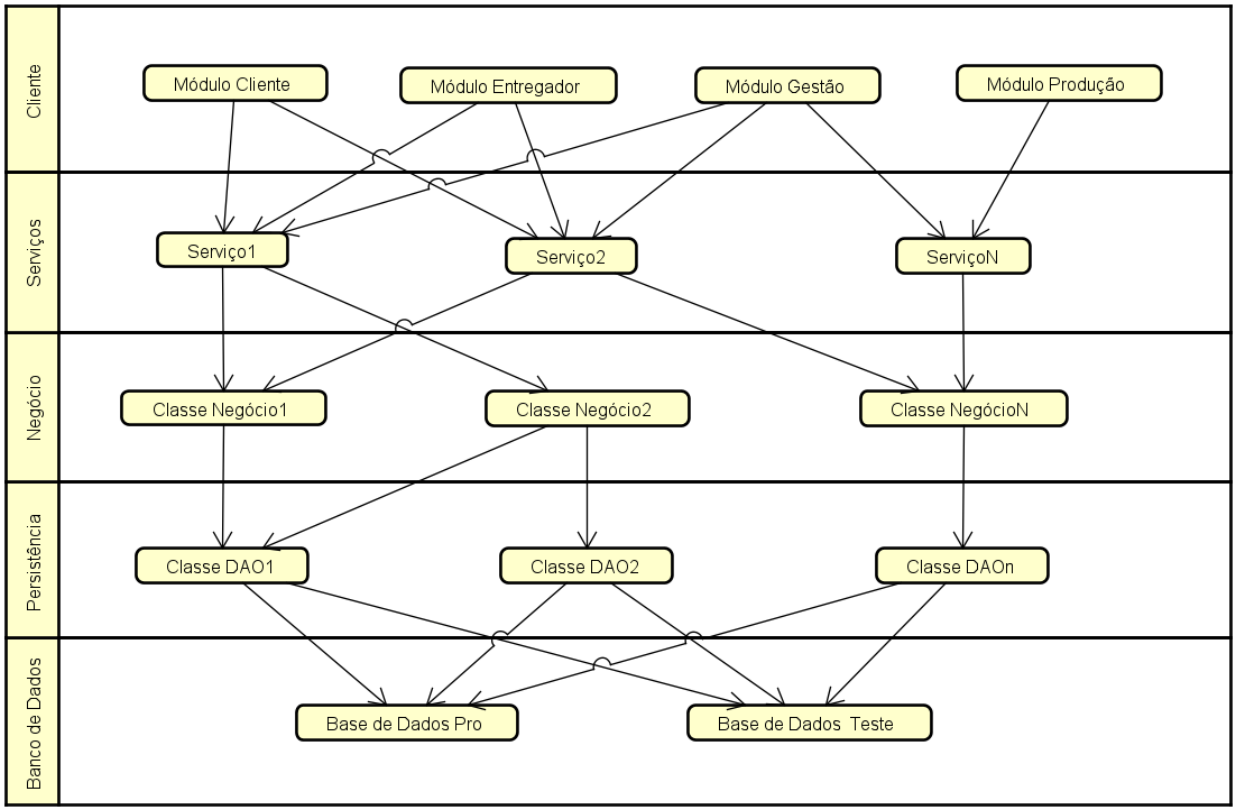
Figura 2 – Modelo Conceitual do Software



3 Modelo Arquitetural do Software Fornecido

Nesta seção são apresentadas informações que subsidiam o entendimento sob a arquitetura do software fornecido. A principal informação é descrever as responsabilidades dos grupo de elementos de software encontrados no projeto implementado. A Figura 3 apresenta uma representação abstrata dos elementos que compõem a arquitetura do software em termos de abstrações (camadas).

Figura 3 – Representação Abstrata da Arquitetura do Software



3.1 Classes DTOs

As classes DTOs (Data Transfer Object) tem como responsabilidade produzir objetos que serão usados como tranferência de dados entre as abstrações (camadas) do software.

A comunicação entre a camada do Cliente e a camada de serviço transfere e recebe dados no formato JSON (JavaScript Object Notation) que contemplem a estrutura de dados dos DTOs. Uma alternativa é fornecer a implementação/especificação dos DTOs aos Clientes para que possam lidar diretamente com istâncias dessas classes.

A comunicação entre a camada de Serviços e a camada de Negócio é feita por meio de instâncias de DTOs. Veja o exemplo de uma classe DTO a seguir:

```
1 package ifmt.cba.dto;
2
3 import org.apache.commons.lang3.builder.ToStringBuilder;
4 import org.apache.commons.lang3.builder.ToStringStyle;
5
6 public class ProdutoDTO {
7
8     private int codigo;
9     private String nome;
```

```
10      private float custoUnidade;
11      private int valorEnergetico;
12      private int estoque;
13      private GrupoAlimentarDTO grupoAlimentar;
14
15      public int getCodigo() {
16          return codigo;
17      }
18
19      public void setCodigo(int codigo) {
20          this.codigo = codigo;
21      }
22
23      public String getNome() {
24          return nome;
25      }
26
27      public void setNome(String nome) {
28          this.nome = nome;
29      }
30
31      public float getCustoUnidade() {
32          return custoUnidade;
33      }
34
35      public void setCustoUnidade(float custoUnidade) {
36          this.custoUnidade = custoUnidade;
37      }
38
39      public int getValorEnergetico() {
40          return valorEnergetico;
41      }
42
43      public void setValorEnergetico(int valorEnergetico) {
44          this.valorEnergetico = valorEnergetico;
45      }
46
47      public GrupoAlimentarDTO getGrupoAlimentar() {
48          return grupoAlimentar;
49      }
50
51      public void setGrupoAlimentar(GrupoAlimentarDTO grupoAlimentar) {
52          this.grupoAlimentar = grupoAlimentar;
53      }
54
55      public int getEstoque() {
56          return estoque;
57      }
58
59      public void setEstoque(int estoque) {
60          this.estoque = estoque;
61      }
62
63      @Override
64      public String toString() {
65          return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
66      }
67  }
```

Código 1 – ProdutoDTO.java

As classes DTOs são extremamente simples, representam apenas a estrutura de dados de entidades do modelo de domínio do software, não possuindo nenhuma lógica associada. Apenas a sobrescrita do método toString() é implementada nessa classe, além dos atributos e métodos de acesso aos atributos.

3.2 Classes de Entidades

As classes de Entidades tem como responsabilidade produzir objetos que serão usados como tranferência de dados entre as camadas de negócio e persistência.

Quando as classes da camada de negócio recebem instâncias de DTOs, estes são convertidos para instâncias de Entidades. As classes de entidades possuem duas características que as diferenciam dos DTOs: 1 - possuem o MOR (mapeamento objeto-relacional); 2 - possuem lógica de validação dos dados presentes nos atributos. Veja o exemplo de uma classe de Entidade a seguir:

```java
package ifmt.cba.entity;

import org.apache.commons.lang3.builder.ToStringBuilder;
import org.apache.commons.lang3.builder.ToStringStyle;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.JoinColumn;
import jakarta.persistence.ManyToOne;
import jakarta.persistence.Table;

@Entity
@Table(name = "produto")
public class Produto {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int codigo;

    @Column(name = "nome")
    private String nome;

    @Column(name = "custo_unidade")
    private float custoUnidade;

    @Column(name = "valor_energetico")
    private int valorEnergetico;

    @Column(name = "estoque")
    private int estoque;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "id_grupo")
    private GrupoAlimentar grupoAlimentar;

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public float getCustoUnidade() {
        return custoUnidade;
    }

    public void setCustoUnidade(float custoUnidade) {
        this.custoUnidade = custoUnidade;
    }

    public int getValorEnergetico() {
        return valorEnergetico;
    }

    public void setValorEnergetico(int valorEnergetico) {
```

```
69          this.valorEnergetico = valorEnergetico;
70      }
71
72      public GrupoAlimentar getGrupoAlimentar() {
73          return grupoAlimentar;
74      }
75
76      public void setGrupoAlimentar(GrupoAlimentar grupoAlimentar) {
77          this.grupoAlimentar = grupoAlimentar;
78      }
79
80      public int getEstoque() {
81          return estoque;
82      }
83
84      public void setEstoque(int estoque) {
85          this.estoque = estoque;
86      }
87
88      @Override
89      public String toString() {
90          return ToStringBuilder.reflectionToString(this, ToStringStyle.JSON_STYLE);
91      }
92
93      @Override
94      public int hashCode() {
95          final int prime = 31;
96          int result = 1;
97          result = prime * result + codigo;
98          return result;
99      }
100
101     @Override
102     public boolean equals(Object obj) {
103         if (this == obj)
104             return true;
105         if (obj == null)
106             return false;
107         if (getClass() != obj.getClass())
108             return false;
109         Produto other = (Produto) obj;
110         if (codigo != other.codigo)
111             return false;
112         return true;
113     }
114
115     public String validar() {
116         String retorno = "";
117
118         if (this.nome == null || this.nome.length() < 3) {
119             retorno += "Nome invalido";
120         }
121
122         if (this.custoUnidade <= 0) {
123             retorno += "Custo por unidade invalido";
124         }
125
126         if (this.valorEnergetico < 0) {
127             retorno += "Valor energetico invalido";
128         }
129
130         if (estoque < 0) {
131             retorno += "Estoque invalido";
132         }
133
134         if (this.grupoAlimentar == null) {
135             retorno += "Grupo alimentar nao pode ser nulo";
136         }
137
138         return retorno;
139     }
140 }
```

Código 2 – Produto.java

Pode-se observar que a classe entidade tem a mesma estrutura de dados da classe DTO correspondente, acrescida das anotações de MOR da JPA. Além disso sobrescreve os métodos toString(), hashCode() e equals().

Por fim, acrescenta-se um método validar(), cuja finalidade é verificar se os valores presentes nos atributos do objeto satisfazem as condições estabelecidas. Este método, cujo retorno é uma String, pode representar duas situações: 1 - se os dados estiverem corretos a String devolvida será vazia; 2 - para cada valor de atributo inválido, será concatenado uma mensagem de inconsistência, sendo devolvida ao final da validação a String contendo todas as mensagem de insconsistência concatenadas.

3.3 Classes de Negócio

As classes de Negócio reúnem as operações CRUD das Entidades, além de operações de processamento e consulta aos dados da Entidade relacionada. Essas recebem os DTOs vindo da camada de Serviços, convertê-los para Entidades, e verificar a validade dos dados contidos, as regras de negócio e executar as ações de persistência. Quando a resposta de operações das Negócio devolvem instâncias de objetos de Entidades, estas serão convertidas para objetos DTOs antes de serem devolvidos. A seguir, veja um exemplo de classe de Negócio:

```
1  package ifmt.cba.negocio;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.modelmapper.ModelMapper;
7
8  import ifmt.cba.dto.ProdutoDTO;
9  import ifmt.cba.entity.Produto;
10 import ifmt.cba.persistencia.PersistenciaException;
11 import ifmt.cba.persistencia.ProdutoDAO;
12
13 public class ProdutoNegocio {
14  private ModelMapper modelMapper;
15  private ProdutoDAO produtoDAO;
16
17  public ProdutoNegocio(ProdutoDAO produtoDAO) {
18   this.produtoDAO = produtoDAO;
19   this.modelMapper = new ModelMapper();
20  }
21
22  public void inserir(ProdutoDTO produtoDTO) throws NegocioException {
23
24   Produto produto = this.toEntity(produtoDTO);
25   String mensagemErros = produto.validar();
26
27   if (!mensagemErros.isEmpty()) {
28    throw new NegocioException(mensagemErros);
29   }
30
31   try {
32    if (!produtoDAO.buscarPorParteNome(produto.getNome()).isEmpty()) {
33     throw new NegocioException("Ja existe esse produto");
34    }
35    produtoDAO.beginTransaction();
36    produtoDAO.incluir(produto);
37    produtoDAO.commitTransaction();
38   } catch (PersistenciaException ex) {
39    produtoDAO.rollbackTransaction();
40    throw new NegocioException("Erro ao incluir o produto - " + ex.getMessage());
41   }
42  }
43
44  public void alterar(ProdutoDTO produtoDTO) throws NegocioException {
45
46   Produto produto = this.toEntity(produtoDTO);
47   String mensagemErros = produto.validar();
```

```
48    if (!mensagemErros.isEmpty()) {
49     throw new NegocioException(mensagemErros);
50    }
51    try {
52     if (produtoDAO.buscarPorCodigo(produto.getCodigo()) == null) {
53      throw new NegocioException("Nao existe esse produto");
54     }
55     produtoDAO.beginTransaction();
56     produtoDAO.alterar(produto);
57     produtoDAO.commitTransaction();
58    } catch (PersistenciaException ex) {
59     produtoDAO.rollbackTransaction();
60     throw new NegocioException("Erro ao alterar o produto - " + ex.getMessage());
61    }
62   }
63
64   public void excluir(ProdutoDTO produtoDTO) throws NegocioException {
65
66    Produto produto = this.toEntity(produtoDTO);
67    try {
68     if (produtoDAO.buscarPorCodigo(produto.getCodigo()) == null) {
69      throw new NegocioException("Nao existe esse produto");
70     }
71     produtoDAO.beginTransaction();
72     produtoDAO.excluir(produto);
73     produtoDAO.commitTransaction();
74    } catch (PersistenciaException ex) {
75     produtoDAO.rollbackTransaction();
76     throw new NegocioException("Erro ao excluir o produto - " + ex.getMessage());
77    }
78   }
79
80
81   public List<ProdutoDTO> pesquisaParteNome(String parteNome) throws NegocioException {
82    try {
83     return this.toDTOAll(produtoDAO.buscarPorParteNome(parteNome));
84    } catch (PersistenciaException ex) {
85     throw new NegocioException("Erro ao pesquisar produto pelo nome - " + ex.getMessage());
86    }
87   }
88
89   public ProdutoDTO pesquisaCodigo(int codigo) throws NegocioException {
90    try {
91     return this.toDTO(produtoDAO.buscarPorCodigo(codigo));
92    } catch (PersistenciaException ex) {
93     throw new NegocioException("Erro ao pesquisar produto pelo codigo - " + ex.getMessage());
94    }
95   }
96
97   public List<ProdutoDTO> toDTOAll(List<Produto> listaProduto) {
98    List<ProdutoDTO> listaDTO = new ArrayList<ProdutoDTO>();
99
100   for (Produto produto : listaProduto) {
101    listaDTO.add(this.toDTO(produto));
102   }
103   return listaDTO;
104  }
105
106  public ProdutoDTO toDTO(Produto produto) {
107   return this.modelMapper.map(produto, ProdutoDTO.class);
108  }
109
110  public Produto toEntity(ProdutoDTO produtoDTO) {
111   return this.modelMapper.map(produtoDTO, Produto.class);
112  }
113 }
```

Código 3 – ProdutoNegocio.java

As operações de persistência invocadas pelas classes de Negócio são executadas pelos objetos de classes DAO, injetados pelo construtor dessas classes.

3.4 Classes de Persistência (DAOs)

Essas classe tem o objetivo de executarem as operações de persistência e recuperação de informações das entidades do domínio do software. As classes DAOs foram implementadas por meio da API JPA do Java.

Como forma de reutilizar um código comum entre os DAOs, foi codificada uma superclasse DAO, que reúne as operações de inclusão, alteração e exclusão, além do controle de trasação com o banco de dados. Veja a seguir a codificação da superclasse DAO:

```
1  package ifmt.cba.persistencia;
2
3  import jakarta.persistence.EntityManager;
4
5  public class DAO<VO> {
6
7   protected EntityManager entityManager;
8
9   public DAO(EntityManager entityManager) throws PersistenciaException {
10   this.entityManager = entityManager;
11  }
12
13  public void incluir(VO vo) throws PersistenciaException {
14   try {
15    this.entityManager.persist(vo);
16   } catch (Exception e) {
17    throw new PersistenciaException("Erro ao incluir " + vo.getClass() + " - " + e.getMessage()
          );
18   }
19  }
20
21  public void alterar(VO vo) throws PersistenciaException {
22   try {
23    this.entityManager.merge(vo);
24   } catch (Exception e) {
25    throw new PersistenciaException("Erro ao alterar " + vo.getClass() + " - " + e.getMessage()
          );
26   }
27  }
28
29  public void excluir(VO vo) throws PersistenciaException {
30   try {
31    this.entityManager.remove(vo);
32   } catch (Exception e) {
33    throw new PersistenciaException("Erro ao excluir " + vo.getClass() + " - " + e.getMessage()
          );
34   }
35  }
36
37  public EntityManager getEntityManager() {
38   return entityManager;
39  }
40
41  public void setEntityManager(EntityManager entityManager) {
42   this.entityManager = entityManager;
43  }
44
45  public void beginTransaction() {
46   this.entityManager.getTransaction().begin();
47  }
48
49  public void commitTransaction() {
50   this.entityManager.getTransaction().commit();
51  }
52
53  public void rollbackTransaction() {
54   this.entityManager.getTransaction().rollback();
55  }
56
57 }
```

Código 4 – DAO.java

A classe DAO executa as operações de persistência por meio de um objeto EntityManager, injetado pelo construtor dessa classe. Veja a seguir a codificação de uma subclasse DAO:

```java
package ifmt.cba.persistencia;

import java.util.List;
import ifmt.cba.entity.Produto;
import jakarta.persistence.EntityManager;
import jakarta.persistence.Query;

public class ProdutoDAO extends DAO<Produto>{

    public ProdutoDAO(EntityManager entityManager) throws PersistenciaException {
  super(entityManager);
 }

 public Produto buscarPorCodigo(int codigo) throws PersistenciaException {

  Produto produto = null;

  try {
   produto = this.entityManager.find(Produto.class, codigo);
  } catch (Exception ex) {
   throw new PersistenciaException("Erro na selecao por codigo - " + ex.getMessage());
  }
  return produto;
 }

 @SuppressWarnings("unchecked")
 public List<Produto> buscarPorGrupoAlimentar(int codigoGrupo) throws PersistenciaException {
  List<Produto> listaProduto;
  try {
   Query query = this.entityManager
     .createQuery("SELECT p FROM Produto p WHERE p.grupoAlimentar.codigo = :codgrupo ORDER BY
         p.nome");
   query.setParameter("codgrupo", codigoGrupo);
   listaProduto = query.getResultList();
  } catch (Exception ex) {
   throw new PersistenciaException("Erro na selecao por grupo alimentar - " + ex.getMessage())
       ;
  }
  return listaProduto;
 }

 @SuppressWarnings("unchecked")
 public List<Produto> buscarPorParteNome(String nome) throws PersistenciaException {
  List<Produto> listaProduto;
  try {
   Query query = this.entityManager
     .createQuery("SELECT p FROM Produto p WHERE UPPER(p.nome) LIKE :pNome ORDER BY p.nome");
   query.setParameter("pNome", "%" + nome.toUpperCase().trim() + "%");
   listaProduto = query.getResultList();
  } catch (Exception ex) {
   throw new PersistenciaException("Erro na selecao por parte do nome - " + ex.getMessage());
  }
  return listaProduto;
 }
}
```

Código 5 – ProdutoDAO.java

As subclasses de DAO reúnem operações particulares de cada Entidade, principalmente operações de recuperação de dados.

Os objetos recebidos e devoldidos pelas classes DAO são de classes do tipo Entidade, que serão tratados e convertidos para DTOS nas classes de negócio.

3.5 Classes Utilitária FabricaEntityManager

A classe utilitária FabricaEntityManager tem como objetivo gerar instâncias de objetos EntityManager que são usados pelas classes DAOs para executarem as operações de persistência via JPA.

```
1  package ifmt.cba.persistencia;
2
3  import jakarta.persistence.EntityManager;
4  import jakarta.persistence.EntityManagerFactory;
5  import jakarta.persistence.Persistence;
6
7  public class FabricaEntityManager {
8
9   private static EntityManagerFactory emf_producao;
10  private static EntityManagerFactory emf_teste;
11  public static final String UNIDADE_PRODUCAO = "restaurante_producao";
12  public static final String UNIDADE_TESTE = "restaurante_teste";
13
14  private FabricaEntityManager() {
15
16  }
17
18  public static EntityManager getEntityManagerProducao() {
19   if (emf_producao == null) {
20    emf_producao = Persistence.createEntityManagerFactory(UNIDADE_PRODUCAO);
21   }
22   return emf_producao.createEntityManager();
23  }
24
25  public static EntityManager getEntityManagerTeste() {
26   if (emf_teste == null) {
27    emf_teste = Persistence.createEntityManagerFactory(UNIDADE_TESTE);
28   }
29   return emf_teste.createEntityManager();
30  }
31 }
```

Código 6 – FabricaEntityManager.java

Pode-se observar que em sua codificação existem dois métodos que produzem objetos EntityManager: getEntityManagerProducao() e getEntityManagerTeste(). Essa implementação permite que o código de classes de negócio e persistência trabalhem com uma base de dados diferente da base de dados dos testes automatizados. No arquivo de configurações da JPA do projeto, foram especificadas duas unidades de persistência, uma para ser usada pelo código que irá entrar em produção e outra pelos códigos dos testes automatizados, permitindo o isolamento dessas duas ações. Veja o código do arquivo persistence.xml:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/
         persistence/persistence_2_1.xsd"
5    version="2.1">
6
7   <persistence-unit name="restaurante_producao" transaction-type="RESOURCE_LOCAL">
8    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
9    <exclude-unlisted-classes>false</exclude-unlisted-classes>
10   <properties>
11
12    <property name="javax.persistence.jdbc.url" value="jdbc:postgresql://localhost:5432/
         restaurante_producao" />
13    <property name="javax.persistence.jdbc.user" value="postgres" />
14    <property name="javax.persistence.jdbc.driver" value="org.postgresql.Driver" />
15    <property name="javax.persistence.jdbc.password" value="postgres" />
16    <property name="javax.persistence.schema-generation.database.action" value="none" /> <!--
         none / create / create-drop / update -->
17    <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
18    <property name="hibernate.show_sql" value="true" /> <!-- Show SQL in console -->
```

```
19     <property name="hibernate.format_sql" value="true" /> <!-- Show SQL formatted -->
20    </properties>
21   </persistence-unit>
22
23   <persistence-unit name="restaurante_teste" transaction-type="RESOURCE_LOCAL">
24    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
25    <exclude-unlisted-classes>false</exclude-unlisted-classes>
26    <properties>
27
28     <property name="javax.persistence.jdbc.driver" value="org.h2.Driver" />
29     <property name="javax.persistence.jdbc.url" value="jdbc:h2:mem:restaurante_teste" />
30     <property name="javax.persistence.jdbc.user" value="sa" />
31     <property name="javax.persistence.jdbc.password" value="" />
32
33     <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect"/>
34
35     <property name="javax.persistence.schema-generation.database.action" value="create" /> <!--
           none / create / create-drop / update -->
36     <property name="hibernate.dialect" value="org.hibernate.dialect.PostgreSQLDialect" />
37     <property name="hibernate.show_sql" value="true" /> <!-- Show SQL in console -->
38     <property name="hibernate.format_sql" value="true" /> <!-- Show SQL formatted -->
39    </properties>
40   </persistence-unit>
41 </persistence>
```

Código 7 – Persistence.xml

A unidade de persistência **restaurante_producao** está configurada para acessar uma base de dados com o nome de restaurante_producao em um SGBD PostgresSQL.

A unidade de persistência **restaurante_teste** está configurada para acessar uma base de dados com o nome de restaurante_teste em um SGBD H2 em memória.

4 Execução do Estudo de Caso

Considerando as informações que foram apresentadas nas seções anteriores a essa e o código fonte do software fornecido, esta seção descreve as atividades que deverão ser desenvolvidas, como devem ser desenvolvidas e como devem ser entregues.

4.1 Como Desenvolver as Atividades

As atividades do estudo de caso descrito nesse documento devem ser desenvolvidas por grupos de alunos formados por 3 a 5 integrantes. É importante a formação dos grupos por integrantes com perfis diferentes, por exemplo, não forma grupos com apenas integrantes que dominam a implementação de software/testes ou apenas por integrantes que não dominem. As atividades serão compostas por ações de planejamento, codificação e apuração de resultados de testes, sendo assim, que integrantes que não possuem práticas de implementação desempenhem funções importantes para o grupo.

4.2 Planejamento dos Testes

O grupo deve pesquisar e eleger um modelo de documento para o planejamento de teste, normalmente conhecido como Plano de Testes. O tem liberdade de customizar o modelo adotado para ser suscito e claro, contendo pelo menos as seguintes informações:

- Identificação do software;
- Breve visão geral do software;
- Lista de Users Stories eleitas como críticas para o negócio;
- Lista de Unidades a serem testadas pelos testes automatizados unitários;
- Lista de Unidades a serem testadas pelos testes automatizados de integração;
- Lista de Unidades a serem testadas pelos testes automatizados de sistema;
- Lista de Unidades a serem testadas pelos testes automatizados de performance;

Observação: neste estudo de caso, os testes automatizados de sistemas, que normalmente são realizados por scripts que executam comandos na interface do usuário final, serão automatizados por testes da API de serviços disponibilizada. Sendo assim, a automação de testes de sistema será adaptada para testes de serviços, que irão testar as funcionalidades do sistema por meio da interface de serviços fornecida, gerando requisições REST para o testes ponta-a-ponta.

4.3 Implementação dos Testes

Nesta seção são especificados os critérios para a implementação dos testes automatizados deste estudo de caso. Os critérios foram divididos por tipos/níveis de testes e serão apresentados nas próximas subseções.

Para todos os testes implementados deve existir uma especificação de caso de teste, podendo ser no formato estendido clássico, padrão GivenWhenThen, ou ainda com tabela de decisão. O grupo tem autonomia para decidir quais modelos de casos de testes usará, podendo usar mais de um tipo, caso ache necessário.

Nas subseções a seguir será usado o termo grupo de classe, que significa os agrupamentos e/ou as abstrações criadas no projeto do software fornecido, tais como: classes DTOs, Entidade, Negócio, Persistência e Serviços.

4.3.1 Testes Unitários

Considerando a pirâmide de distribuição de testes, os testes unitários são aqueles que precisam ter um volume maior de automação. Dessa forma, deverá ser implementado um número mínimo de 15 unidades (classes) de testes, sendo que para cada unidade testada, podem existir N casos de teste.

O grupo irá decidir quais grupos de classes serão testados no nível de teste unitário, mas é necessário distribuir os testes em pelo menos dois grupos.

Caso uma unidade escolhida para a implementação de teste unitário dependa de outra unidade, a dependência deve ser isolada (mockada), garantindo que o teste abraja apenas a unidade em questão.

Para cada unidade testada (classe), nomear a classe de teste unitário correspondente da seguinte forma:

<NomeClasse>+_UnitarioTest

Exemplo:

Teste unitário para a classe ProdutoNegocio, a classe de teste deve ser nomeada:

ProdutoNegocio_UnitarioTest

Os nomes dos casos de testes (métodos) da classe de teste unitário podem ser definidos livremente pelo grupo.

4.3.2 Testes de Integração

A implementação de testes de integração envove sempre uma unidade provedora de uma funcionalidade e uma unidade cliente daquela funcionalidade. O objetivo é testar a integração entre esses duas unidades, considerando o sentido cliente para provedor.

Para o nível de teste de integração, será necessário um número mínimo de 15 integrações, sendo considerado uma integração, cada caso de teste implementado. Por exemplo, em um teste de integração entre uma classe de negócio e uma classe de persistência, cada caso de teste será considerado uma integração. Assim, se forem produzidos três casos de testes (métodos) para a integração entre ProdutoNegocio e ProdutoDAO, serão contadas três integrações.

O grupo irá decidir quais grupos de classes serão testados no nível de teste de integração, mas é necessário distribuir os testes em pelo menos três grupos.

Caso exista uma dependência entre a classe provedora e outra unidade de código, está deverá ser isolada (mockada), garantindo que o teste de integração não extrapole os limites entre a unidade cliente e a unidade provedora em teste.

Para cada grupo de integração, nomear a classe de teste correspondente da seguinte forma:

<NomeClasseCliente>+<_>+<NomeClasseProvedor>+_IntegracaoTest

Exemplo:

Teste de integração entre as classes ProdutoNegocio e ProdutoDAO, a classe de teste deve ser nomeada:

ProdutoNegocio_ProdutoDAO_IntegracaoTest

Os nomes dos casos de testes (métodos) da classe de teste de integração podem ser definidos livremente pelo grupo.

4.3.3 Testes de Sistema

Para esse estudo de caso, os testes no nível de sistema não serão realizados na interface de usuário final, mas sim, na interface de serviços. Dessa forma, os testes de sistema devem cobrir as funcionalidades do software por meio de chamadas das operações dos serviços.

As funcionalidades a serem testadas devem ser retiradas da lista de Users Stories classificadas como críticas, no documento de Plano de Testes. Devem ser produzidos pelo menos 5 testes de sistema, ou seja, testar 5 funcionalidades ponta-a-ponta a partir do serviço.

Para cada teste de sistema, nomear a classe de teste correspondente da seguinte forma:

<NomeFuncionalidade>+_SistemaTest

Exemplo:

Teste de sistema para a funcionalidade Mudar Estado de Pedidos, a classe de teste deve ser nomeada:

MudarEstadoPedido_SistemaTest

Os nomes dos casos de testes (métodos) da classe de teste de sistema podem ser definidos livremente pelo grupo.

4.3.4 Testes de Performance

Os testes de performance devem ser produzidos considerando o tempo de resposta de funcionalidades do software. Sendo assim, os testes de performance devem ser produzidos a partir da invocação das operações dos serviços disponibilizados.

A partir dos testes de sistemas produzidos, escolher 2 casos de testes e implementar critérios de performance (timeout) para eles.

O teste de performance deve considerar um volume considerável de dados na base de dados para a realização e verificação do tempo de resposta, assim, o grupo precisará gerar uma massa de dados que contemple pelo menos 500.000 registros na tabela base da funcionalidade. Por exemplo, caso seja realizado um teste de performance da funcionalidade Mudar Estado de Pedidos, deve existir pelo menos 500.000 registros na tabela de Pedidos, e considerar o tempo de resposta o intervalo gasto entre recuperar os dados do pedido a ser alterado, mudar o estado e salvar a alteração.

Para cada teste de performance, nomear a classe de teste correspondente da seguinte forma:

<NomeFuncionalidade>+_PerformanceTest

Exemplo:

Teste de performance para a funcionalidade Mudar Estado de Pedidos, a classe de teste deve ser nomeada:

MudarEstadoPedido_PerformanceTest

Os nomes dos casos de testes (métodos) da classe de teste de performance podem ser definidos livremente pelo grupo.

5 Entrega e Apresentação

Nesta seção serão apresentadas as informações referente as entregas, prazos, formas de entregue e critérios de avaliação.

5.1 Entrega da Implementação dos Testes

Os resultados do desenvolvimento do estudo de caso serão entregues em três arquivos, dois no formato PDF e um no formato Zip. São eles:

1. Plano de Testes - formato PDF;
2. Especificação de Casos de Testes - formato PDF;
3. Pasta do projeto com os testes - formato ZIP;

As entregas desses arquivos serão realizadas no ambiente AVA/Moodle em tarefas específicas para cada arquivo.

Data de Entrega: 15/12/2023.

5.2 Apresentação da Implementação dos Testes

No dia 18/12/2023 cada grupo irá fazer uma apresentação de no máximo 20 minutos, com o objetivo de demonstrar o que o grupo produzir no desenvolvimento do estudo de caso. Também é interessante relatar qual foi a experiência que o grupo teve com o desenvolvimento do estudo de caso.

É interessante que o grupo prepare uma apresentação, com algum software de apresentação, e gere um PDF da apresentação, para evitar possíveis problemas de compatibilidade de software e/ou versões.

5.3 Critérios de Avaliação

A nota da disciplina será calculada seguindo a seguinte distribuição:

1. Plano de Testes - 1,0 ponto;
2. Especificação de Casos de Testes - 2,0 pontos;
3. Implementação dos testes - 5,0 pontos;
4. Apresentação final - 2,0 pontos;