

PROGRAMAÇÃO 2
CIÊNCIA DA COMPUTAÇÃO
MÁSIO CÉSAR DE CARVALHO MORAES

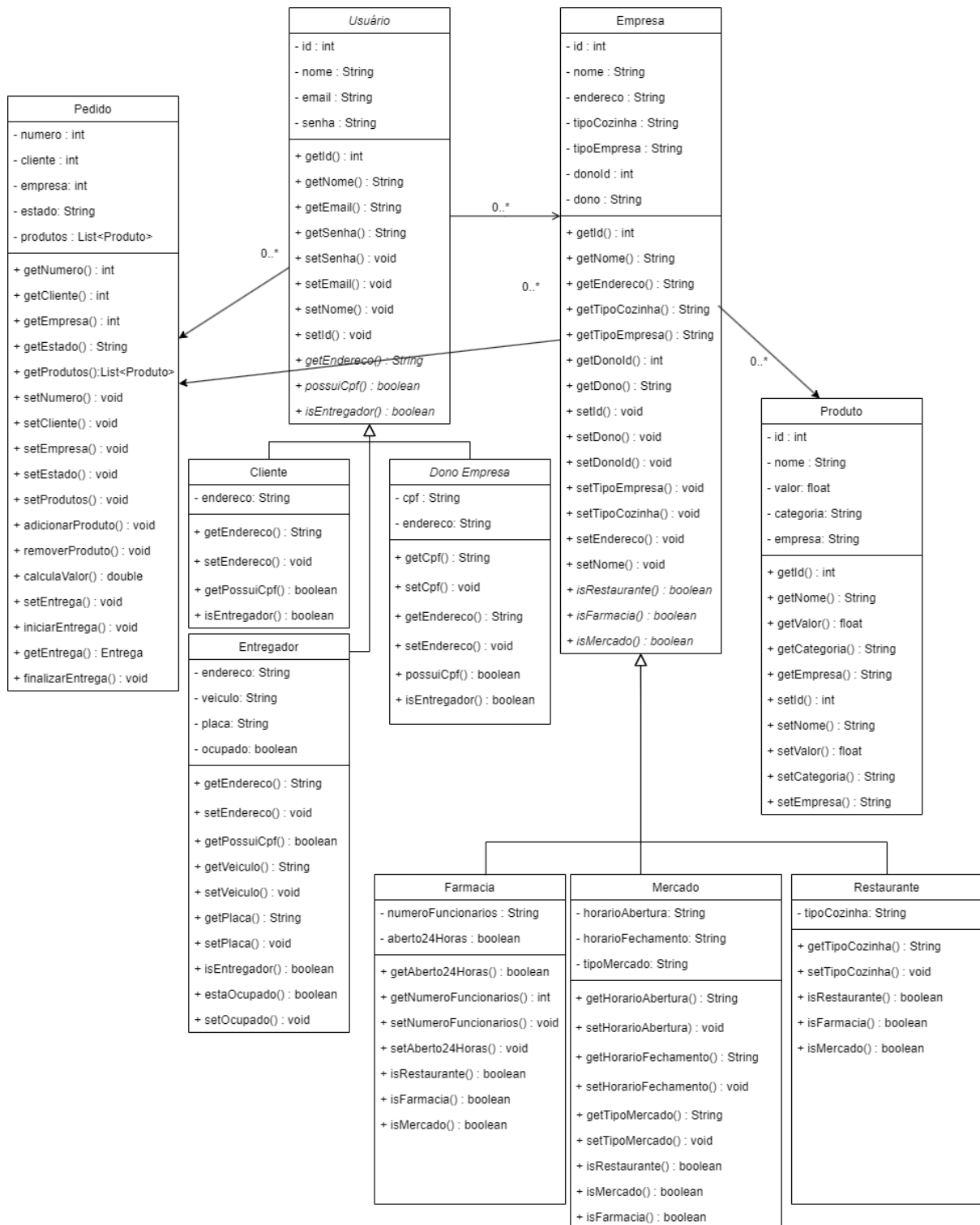
RELATÓRIO DE PROGRAMAÇÃO 2 - MYFOOD

SUMÁRIO

1. INTRODUÇÃO	3
2. DESIGN ARQUITETURAL DO SISTEMA	4
• 2.1. Exceptions	4
• 2.2. Models	4
• 2.3. Services	4
3. PRINCIPAIS COMPONENTES	4
3.1. Usuario	4
3.2. Empresa	6
3.3. EmpresaManager	7
3.4. PedidoManager	7
3.5. ProdutoManager	8
3.6. UsuarioManager	9
3.7. DataPersistenceManager	10
4. PADRÕES DE PROJETO	10
4.1. FACADE	10
4.2. MEDIATOR	12
4.3. SINGLETON	14

1. INTRODUÇÃO

Este relatório apresenta o projeto do sistema MyFood, uma aplicação desenvolvida para gerenciar empresas, pedidos, produtos e usuários em um sistema simples de delivery. O sistema inclui a gestão de diferentes tipos de usuários, produtos oferecidos, pedidos realizados, entregas desse pedidos e informações sobre as empresas.



2. DESIGN ARQUITETURAL DO SISTEMA

O design arquitetural do sistema foi estruturado para garantir uma organização clara e eficiente através da implementação do padrão Facade. Esse padrão possibilitou a centralização das operações, simplificando e unificando a interação entre os diversos subsistemas do projeto. A arquitetura foi planejada para assegurar manutenção e reutilização do código, organizando-o com base em suas funções e responsabilidades. As principais camadas que interagem com a Facade incluem: Exceptions, que gerencia o tratamento de erros; Models, que define as estruturas dos principais objetos; e Services, que coordena as operações do sistema e a persistência de dados.

- **2.1. Exceptions**

Para centralizar o tratamento de erros e simplificar a manutenção, a classe de Exceções foi criada. Ela é responsável por capturar e gerenciar todas as exceções geradas durante a execução do sistema, garantindo que problemas inesperados sejam tratados de forma consistente e eficiente.

- **2.2. Models**

Os modelos representam as principais entidades do sistema, estruturando dados e comportamentos relacionados aos usuários, empresas, produtos e pedidos. Eles foram desenvolvidos com base no diagrama de classes inicial e ajustados conforme as necessidades do projeto evoluíram, refletindo as User Stories e as funcionalidades do sistema.

- **2.3. Services**

Os serviços gerenciam as operações principais do sistema, incluindo configurações e histórico de estados. Eles são responsáveis pela criação e manipulação dos dados, controle da persistência através de arquivos XML, e execução de funcionalidades essenciais, como verificação e processamento de dados. Os serviços garantem a integração e o funcionamento adequado entre os diferentes componentes do sistema.

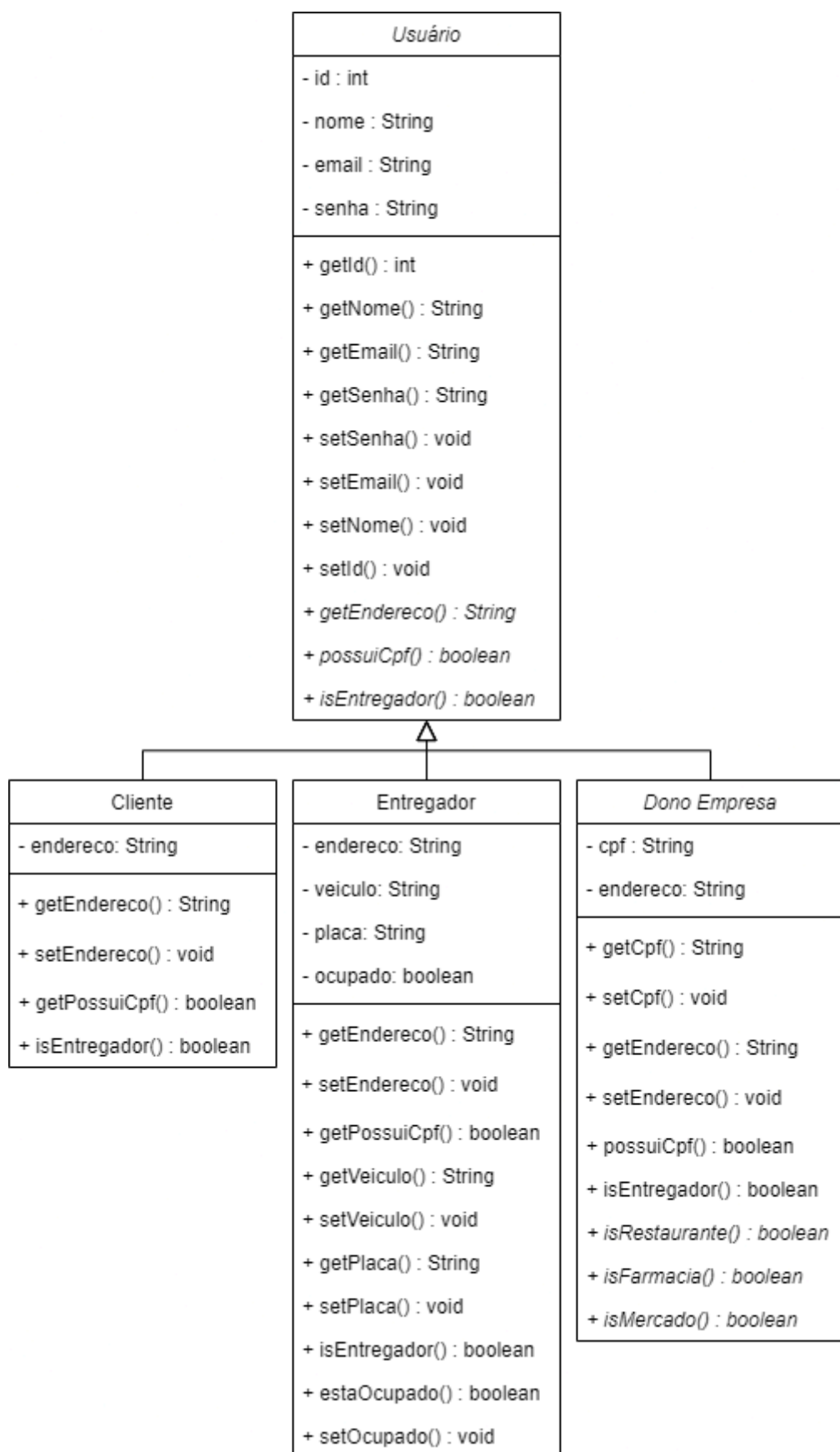
3. PRINCIPAIS COMPONENTES

A seguir, detalhamos os principais componentes do sistema.

3.1. Usuario

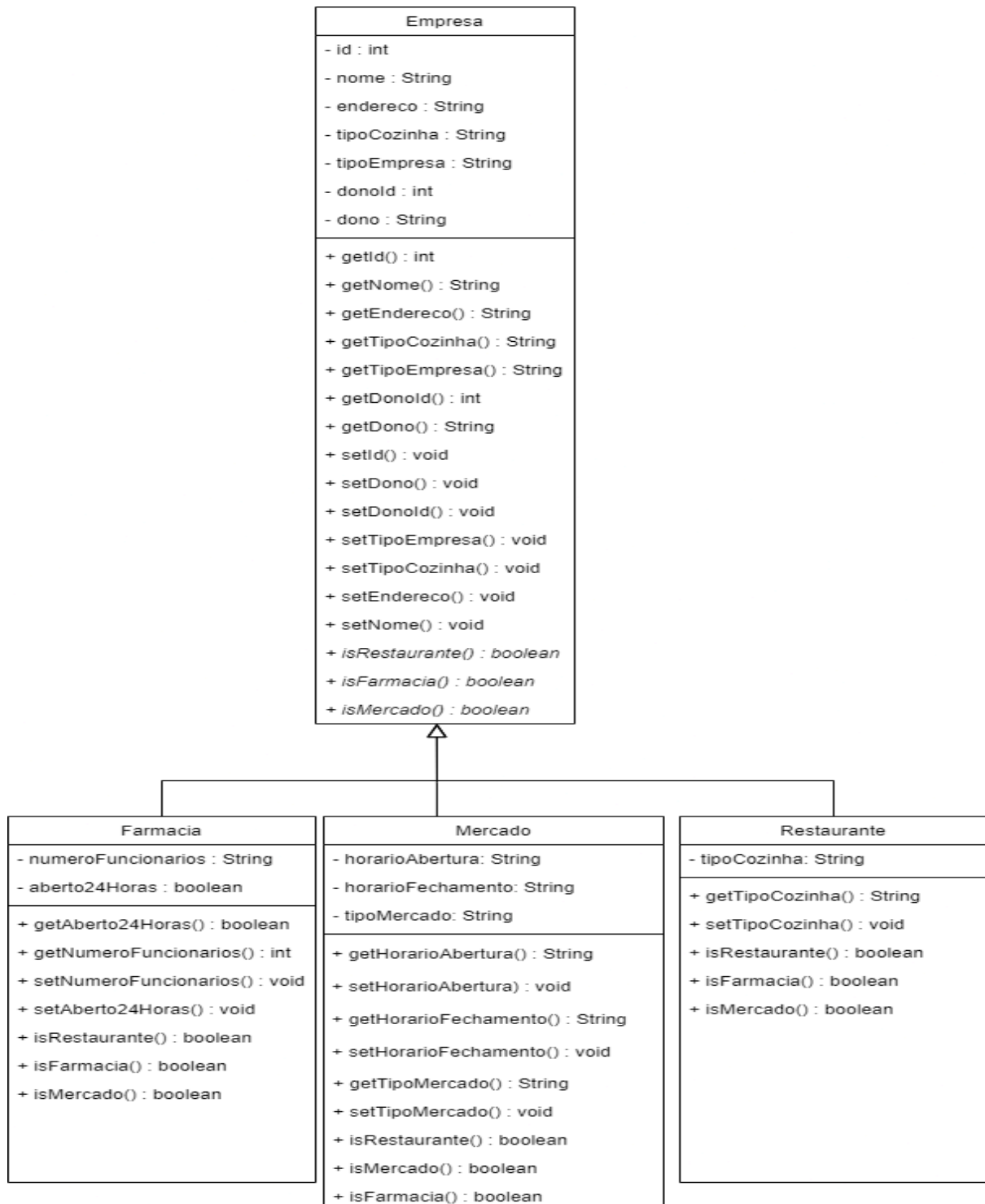
A classe **Usuario** é a base para diferentes tipos de usuários no sistema. Inclui atributos como ID, nome, email, senha. Sendo esta classe herdada por outras 3 classes: **Cliente**, **DonoRestaurante** e **Entregador**. Além do mais, também é uma

classe abstrata, então há somente 3 tipos de usuário, cliente, dono empresa ou entregador.



3.2. Empresa

A classe **Empresa** é a base para diferentes tipos de empresas no sistema. Inclui atributos como ID, nome, email, senha e endereço. Sendo esta classe herdada por outras 3 classes: **Farmacia**, **Restaurante** e **Mercado**. Além do mais, também é uma classe abstrata, então há somente 3 tipos de empresa, farmácia, restaurante ou mercado.



3.3. EmpresaManager

EmpresaManager é responsável por administrar todas as operações relacionadas às empresas no sistema. Suas responsabilidades incluem a criação de novas empresas, a verificação das empresas associadas a um usuário específico e a validação de dados de entrada para garantir que não ocorram duplicidades. Ele assegura que cada empresa seja única e corretamente associada ao seu respectivo dono.

```
12 public class EmpresaManager { ① MasioCesar
13     private Map<Integer, Empresa> empresas = new HashMap<>();
14     private final UsuarioManager usuarioManager;
15     private int nextEmpresaId = 0;
16
17     > public EmpresaManager(UsuarioManager usuarioManager) {...}
21
22     > public int criarEmpresa(String nome, int donoId, String endereco, String tipoCozinha, String tipoEmpresa) throws Exception {...}
26
27     > public String getEmpresasDoUsuario(int idDono) throws Exception {...}
33
34     > public int getIdEmpresa(int idDono, String nome, String indice) throws Exception {...}
40
41     > public String getAtributoEmpresa(int empresaId, String atributo) throws Exception {...}
47
48     > public Empresa getEmpresa(int empresaId) { return empresas.get(empresaId); }
54
55     > public Map<Integer, Empresa> getEmpresas() { return empresas; }
61
62     > public void setEmpresas(Map<Integer, Empresa> empresas) { this.empresas = empresas; }
68
69     > public void zerarSistema() {...}
75
76     > public boolean isDonoEmpresa(int donoId, int empresaId) throws Exception {...}
82
83     public void salvarDados() { ① MasioCesar
84         XMLEmpresa.save(empresas);
85     }
86 }
87 }
```

3.4. PedidoManager

PedidoManager gerencia todos os aspectos relacionados aos pedidos dentro do sistema. Suas funções abrangem a criação de novos pedidos, a atualização do status dos pedidos existentes, e a entrega dos pedidos. Ele garante que todos os pedidos sejam corretamente registrados, atualizados e entregues.

```

14 public class PedidoManager {
15     // Instância única da classe PedidoManager
16     private static PedidoManager instance;
17
18     private Map<Integer, Pedido> pedidosPorCliente;
19     private Mediator mediator;
20
21     private int pedidoNumero = 0;
22     private int contadorEntrega = 0;
23
24     > public PedidoManager(Mediator mediator) {...}
28
29     > public static PedidoManager getInstance(Mediator mediator) {...}
35
36     > public void zerarSistema() {...}
42
43     > public int criarPedido(int clienteId, int empresaId) throws Exception {...}
61
62     > public void adicionarProduto(int numeroPedido, int produtoId) throws Exception {...}
88
89     > public String getPedidos(int pedidoId, String atributo) throws Exception {...}
132
133     > public void fecharPedido(int numero) throws Exception {...}
141
142     > public void removerProduto(int numero, String nomeProduto) throws Exception {...}
155
156     > public int getNumeroPedido(int cliente, int empresa, int indice) throws Exception {...}
167
168     > public Pedido getPedidoById(int id) { return pedidosPorCliente.get(id); }
171
172     > public void liberarPedido(int numeroPedido) throws Exception {...}
185
186     > public List<Pedido> getPedidosByEmpresaId(int empresaId) {...}
191
192     > public int obterPedido(int entregadorId) throws Exception {...}
232
233     // FUNCOES DE ENTREGA
234
235     > public int criarEntrega(int pedidoNumero, int entregadorId, String destino) throws Exception {...}
267
268     > public Entrega getEntregaById(int entregaId) throws Exception {...}
276
277     > public Object getAtributoEntrega(int entregaId, String atributo) throws Exception {...}
297
298     > public int getIdEntrega(int pedidoNumero) throws Exception {...}
310
311     > public void entregar(int entregaId) throws Exception {...}
316
317     > public void salvarDados() { XMLPedido.save(pedidosPorCliente); }
320 }

```

3.5. ProdutoManager

ProdutoManager é responsável pela administração dos produtos oferecidos pelas empresas no sistema. Ele permite a criação, edição e listagem de produtos, além de possibilitar a verificação e atualização das informações dos produtos existentes. Ele assegura que todos os produtos estejam corretamente catalogados e atualizados.


```

16 public class ProdutoManager {
17     // Instância única da classe ProdutoManager
18     private static ProdutoManager instance;
19
20     private Map<Integer, Map<Integer, Produto>> produtosPorEmpresa;
21     private Mediator mediator;
22     private int nextProductId = 0;
23
24 >     public ProdutoManager(Mediator mediator) {...}
28
29 >     public static ProdutoManager getInstance(Mediator mediator) {...}
36
37 >     public void setProdutosPorEmpresa(Map<Integer, Map<Integer, Produto>> produtosPorEmpresa) {...}
40
41 >     public int criarProduto(int empresaId, String nome, float valor, String categoria) throws Exception {...}
62
63 >     public void editarProduto(int produtoId, String nome, float valor, String categoria) throws Exception {...}
82
83 >     public String getProduto(String nome, int empresaId, String atributo) throws Exception {...}
108
109 >     public String listarProdutos(int empresaId) throws Exception {...}
135
136 >     public Produto getProdutoPorId(int produtoId) {...}
145
146 >     public void zerarSistema() {...}
151
152 >     public void salvarDados() { XMLProduto.save(produtosPorEmpresa); }
155 }

```

3.6. UsuarioManager

UsuarioManager cuida da gestão dos usuários no sistema. Suas responsabilidades incluem a criação de novos usuários, a autenticação de login e a recuperação de informações dos usuários. Além disso, ele realiza a validação dos dados de entrada para garantir a integridade e a segurança das informações dos usuários.

```

16 public class UsuarioManager {
17     // Instância única da classe UsuarioManager
18     private static UsuarioManager instance;
19
20     private Map<String, Usuario> users;
21     private final Map<Integer, Usuario> usersById;
22     private int nextUserId = 0;
23
24 >     public UsuarioManager() {...}
32
33 >     public static UsuarioManager getInstance() {...}
39
40     // DONO RESTAURANTE
41 >     public void criarUsuario(String nome, String email, String senha, String endereco, String cpf) throws Exception {...}
55
56     // CLIENTE
57 >     public void criarUsuario(String nome, String email, String senha, String endereco) throws Exception {...}
68
69     // ENTREGADOR
70 >     public void criarUsuario(String nome, String email, String senha, String endereco, String veiculo, String placa) throws Exception {...}
88
89 >     public int login(String email, String senha) throws Exception {...}
97
98 >     public String getAtributoUsuario(int id, String atributo) throws Exception {...}
128
129 >     public Usuario getUser(int userId) throws Exception {...}
137
138
139 >     private Usuario findUserById(int id) { return usersById.get(id); }
142
143 >     public void zerarSistema() {...}
149
150     // Método para salvar os dados no XML ao encerrar o sistema
151 >     public void salvarDados() { XMLUsuario.save(users); }
154 }

```

3.7. DataPersistenceManager

A classe **DataPersistenceManager** gerencia a persistência de dados em formato XML. Utiliza `XMLEncoder` para salvar um mapa de dados em um arquivo e `XMLDecoder` para carregar esses dados quando necessário. O método `save` grava os dados no arquivo especificado, enquanto o método `load` recupera os dados do arquivo, retornando um novo `HashMap` se o arquivo não existir. Esta classe é essencial para a persistência de usuários, empresas, produtos e pedidos no sistema, garantindo que essas informações sejam armazenadas e recuperadas corretamente entre sessões.

```
9 public class DataPersistenceManager {  ± MasioCesar
10
11     private final String fileName;
12
13     > public DataPersistenceManager(String fileName) { this.fileName = fileName; }
14
15
16     public void save(Map<?, ?> data) {  ± MasioCesar
17         try (XMLEncoder encoder = new XMLEncoder(new FileOutputStream(fileName))) {
18             encoder.writeObject(data);
19         } catch (Exception ignored) {
20         }
21     }
22
23
24     public Map<?, ?> load() {  ± MasioCesar
25         File file = new File(fileName);
26         if (!file.exists()) {
27             return new HashMap<>();
28         }
29
30         try (XMLDecoder decoder = new XMLDecoder(new FileInputStream(fileName))) {
31             return (Map<?, ?>) decoder.readObject();
32         } catch (Exception ignored) {
33         }
34
35         return new HashMap<>();
36     }
37 }
```

4. PADRÕES DE PROJETO

4.1. FACADE

Descrição Geral

O padrão Facade é um padrão de design estrutural que fornece uma interface unificada e simplificada para um conjunto de interfaces em um subsistema. O objetivo principal é abstrair e esconder a complexidade do sistema subjacente, oferecendo uma interface mais acessível e compreensível para os clientes. A ideia é

desacoplar o cliente da complexidade do sistema, facilitando o uso e a manutenção do sistema.

Problema Resolvido

O padrão Facade é projetado para resolver o problema de sistemas complexos com múltiplas interfaces e subsistemas. Em um cenário onde o sistema possui diversas operações e interações complexas, o padrão Facade ajuda a reduzir o acoplamento e a dependência direta entre o cliente e o subsistema. Ele simplifica a comunicação com o sistema, tornando a interface de uso mais intuitiva e menos propensa a erros.

Identificação da Oportunidade

No desenvolvimento do sistema, a oportunidade de aplicar o padrão Facade foi identificada devido à complexidade crescente das operações e interações entre diferentes componentes, como gerenciamento de usuários, empresas, produtos e pedidos. Com diversos serviços e modelos envolvidos, tornou-se necessário um mecanismo para simplificar e centralizar a interação com o sistema, garantindo uma interface mais clara e fácil de usar.

Aplicação no Projeto

O escopo do projeto é extenso e envolve a administração de diversos modelos, como usuários, empresas, produtos e pedidos, além de múltiplos processos de validação e manipulação de dados. Esse ambiente complexo é ideal para a aplicação do padrão Facade, que visa simplificar a interação com o sistema e tornar o acesso às funcionalidades mais direto e intuitivo.

A Facade foi implementada para fornecer uma interface centralizada e simplificada, facilitando a comunicação com o sistema e ocultando a complexidade dos subsistemas subjacentes. Ela oferece métodos que atendem aos principais requisitos funcionais do sistema, como criação, alteração, leitura e remoção de entidades, além de operações relacionadas a pedidos e produtos.

Alguns métodos da Facade incluem:

1. `public void criarUsuario(String nome, String email, String senha, String endereco)` - Cria um novo usuário no sistema.
2. `public String getAtributoUsuario(int id, String atributo)` - Recupera um atributo específico de um usuário.

3. `public int criarEmpresa(String tipoEmpresa, int donoId, String nome, String endereco, String tipoCozinha)` - Cria uma nova empresa associada a um dono.
4. `public void editarProduto(int produtoId, String nome, float valor, String categoria)` - Editar os detalhes de um produto existente.
5. `public int criarPedido(int cliente, int empresa)` - Cria um novo pedido associando um cliente a uma empresa.

Esses métodos facilitam o gerenciamento das entidades e processos do sistema, simplificando a interface com o usuário e proporcionando uma experiência mais coesa e organizada. A Facade atua como um ponto de acesso único para essas operações, reduzindo o acoplamento e a complexidade na comunicação com os subsistemas.

4.2. MEDIATOR

Descrição Geral

O padrão Mediator é um padrão de design comportamental que tem como objetivo facilitar a comunicação entre diferentes objetos em um sistema, evitando que eles se comuniquem diretamente. O Mediator atua como um intermediário central que gerencia as interações entre os objetos, permitindo que eles colaborem de forma indireta. Isso reduz o acoplamento entre os objetos e facilita a manutenção do sistema, já que mudanças em um objeto não afetam diretamente os outros.

A estrutura básica do Mediator envolve:

- **Mediator:** A interface ou classe abstrata que define as operações usadas para coordenar a comunicação entre os objetos.
- **Colleague:** Os objetos que interagem entre si, mas que não se comunicam diretamente. Em vez disso, eles enviam mensagens ao Mediator, que coordena as interações.

Problema Resolvido

O Mediator é projetado para resolver o problema de comunicação direta entre múltiplos objetos, o que pode levar a um sistema com dependências complexas e difíceis de gerenciar. Em sistemas grandes, se os objetos tiverem que se referenciar e comunicar diretamente uns com os outros, o código pode ficar excessivamente acoplado e propenso a erros, tornando a manutenção e a extensão do sistema mais difícil.

O padrão Mediator reduz esse acoplamento, centralizando a lógica de comunicação em um único ponto, facilitando a coordenação e simplificando a lógica do sistema.

Identificação da Oportunidade

Durante o desenvolvimento do sistema, que envolvia diversas entidades como Usuario, Empresa, Produto, Pedido, e Entrega, foi identificada a oportunidade de aplicar o padrão Mediator para evitar o acoplamento entre essas entidades. A necessidade de coordenar as interações entre usuários que criam empresas, empresas que gerenciam produtos, pedidos feitos por clientes e entregas realizadas tornou evidente que permitir que essas entidades se referenciem diretamente resultaria em um código mais difícil de manter.

A aplicação do Mediator permitiu centralizar todas essas interações, possibilitando que novos requisitos fossem integrados ao sistema sem impactar as entidades diretamente envolvidas nas interações.

Aplicação no Projeto

No projeto, o Mediator foi implementado para gerenciar as interações entre diversas entidades do sistema. Ele coordena a comunicação entre Usuario, Empresa, Produto e Pedido. Cada entidade se comunica com o sistema por meio do Mediator, que organiza e simplifica as interações, permitindo que cada componente seja desacoplado das demais entidades.

```
1 package br.ufal.ic.p2.myfood.services.mediator;
2
3 > import ...
4
5
6
7
8
9 public interface Mediator {
10     Usuario getUsuarioById(int id) throws Exception;
11     Empresa getEmpresaById(int id) throws Exception;
12     boolean isDonoEmpresa(int clienteId, int empresaId) throws Exception;
13     Produto getProdutoById(int id) throws Exception;
14     Map<Integer, Empresa> getAllEmpresas() throws Exception;
15 }
```

Utilização do Mediator em no método adicionarProduto da class PedidoManager para obter o ID da empresa e o ID do Produto:

```
public void adicionarProduto(int numeroPedido, int produtoId) throws Exception {  MasioCesar
    Pedido pedido = pedidosPorCliente.get(numeroPedido);

    if (pedido == null) {
        throw new NaoExistePedidoEmAberto();
    }

    if (pedido.getEstado().equals("preparando")) {
        throw new NaoEPossivelAdicionarProdutosAUMPedidoFechado();
    }

    Empresa empresaDoPedido = mediator.getEmpresaById(pedido.getEmpresa());

    // Obtém o produto usando o Mediator
    Produto produto = mediator.getProdutoById(produtoId);

    if (produto == null) {
        throw new ProdutoNaoEncontradoException();
    }

    if (!produto.getEmpresa().equals(empresaDoPedido.getNome())) {
        throw new ProdutoNaoPertenceEmpresaException();
    }

    pedido.adicionarProduto(produto);
}
```

4.3. SINGLETON

Descrição Geral

O padrão de design Singleton é um padrão de criação que garante que uma classe tenha apenas uma instância, fornecendo um ponto de acesso global a essa instância. O objetivo é controlar o número de instâncias de uma classe, permitindo que ela seja instanciada uma única vez e que essa instância seja reutilizada em todo o sistema.

Problema Resolvido

O Singleton resolve o problema de garantir que determinadas classes tenham apenas uma única instância, particularmente em situações onde o acesso global ou compartilhado a um recurso ou gerenciador é necessário. Ele também evita a criação desnecessária de múltiplas instâncias que poderiam causar inconsistências e desperdício de recursos.

Identificação da Oportunidade

No projeto, foi identificada a necessidade de gerenciar componentes importantes como `UsuarioManager`, `EmpresaManager`, `ProdutoManager`, e `PedidoManager` de forma centralizada e controlada. Essas classes desempenham papéis críticos no gerenciamento de entidades, e garantir que haja apenas uma instância em execução evita problemas de sincronização, inconsistências nos dados, ou comportamento não esperado em operações complexas.

Aplicação no Projeto

No sistema, o Singleton foi implementado para garantir que gerenciadores centrais como `UsuarioManager`, `EmpresaManager`, `ProdutoManager`, e `PedidoManager` existam como instâncias únicas. No construtor da classe `Sistema`, essas instâncias são inicializadas chamando o método `getInstance` de cada gerenciador, assegurando que o sistema use sempre a mesma instância em todas as operações:

```
public Sistema() {  
    // Inicializar a única instância  
    this.usuarioManager = UsuarioManager.getInstance();  
    this.empresaManager = EmpresaManager.getInstance( mediator: this);  
    this.produtoManager = ProdutoManager.getInstance( mediator: this);  
    this.pedidosManager = PedidoManager.getInstance( mediator: this);  
}
```

No exemplo do `PedidoManager`, o método `getInstance` verifica se a instância já foi criada. Caso contrário, ele cria uma nova instância e a retorna, garantindo que a mesma instância seja reutilizada:

```
public class PedidoManager {  
    // Instância única da classe PedidoManager  
    private static PedidoManager instance;  
  
    public static PedidoManager getInstance(Mediator mediator) {  
        if (instance == null) {  
            instance = new PedidoManager(mediator);  
        }  
        return instance;  
    }  
}
```

Isso simplifica a arquitetura e o gerenciamento de recursos do sistema, garantindo que cada gerenciador tenha uma instância única, reduzindo o risco de bugs relacionados a múltiplas instâncias não sincronizadas.