



UNIVERSIDADE FEDERAL DE ALAGOAS - CAMPUS A.C. SIMÕES  
INSTITUTO DE COMPUTAÇÃO

## **Relatório - Programação 2 ( MyFood ) Ciência da computação**

JOSÉ HERBERTY DE OLIVEIRA FREIRE (CC)

LEONARDO BARBOSA BARROS (CC)

MACEIÓ - AL  
2024

# Sumário

<b>1.INTRODUÇÃO</b>	<b>3</b>
<b>2.DESIGN DO SISTEMA</b>	<b>4</b>
2.1.Exceptions	4
2.2.Models	6
2.3.Services	11
<b>3.PADRÃO DE PROJETO FACADE</b>	<b>13</b>
3.1.Importância do Padrão Facade	13

## **1.INTRODUÇÃO**

O projeto MyFood é um sistema de gerenciamento de usuários e restaurantes desenvolvido em Java, utilizando o paradigma de orientação a objetos e a técnica de serialização para persistência de dados. O sistema visa facilitar a criação e o gerenciamento de pedidos em restaurantes, fornecendo funcionalidades para a criação de usuários, restaurantes, produtos e pedidos. A serialização é empregada para armazenar e recuperar o estado dos objetos, permitindo que eles sejam salvos em arquivos e recuperados conforme necessário.

## 2.DESIGN DO SISTEMA

O design do sistema é estruturado em diversas camadas e componentes que interagem para fornecer uma solução robusta e escalável. A arquitetura é dividida em três principais áreas: Exceptions, Models e Services.

### 2.1.Exceptions

As exceções são usadas para tratar erros e condições especiais que podem ocorrer durante a execução do sistema. As seguintes exceções são implementadas:

- **NomeInvalidoException:** Lançada quando um nome inválido é fornecido.
- **EmailInvalidoException:** Lançada quando um email inválido é fornecido.
- **SenhaInvalidaException:** Lançada quando uma senha inválida é fornecida.
- **EnderecoInvalidoException:** Lançada quando um endereço inválido é fornecido.
- **EmailExistenteException:** Lançado quando um email já está em uso.
- **CpfInvalidoException:** Lançada quando um CPF inválido é fornecido.

- **NomeEmpresaExistenteException:** Lançada quando o nome da empresa já existe.
- **EnderecoDuplicadoException:** Lançada quando o endereço da empresa já está cadastrado.
- **NomeProdutoExisteException:** Lançada quando um produto com o mesmo nome já existe.
- **ProdutoNaoEncontradoException:** Lançada quando um produto não é encontrado.
- **AtributoInvalidoException:** Lançada quando um atributo inválido é solicitado.
- **PedidoNaoEncontradoException:** Lançada quando um pedido não é encontrado.
- **ProdutoNaoPertenceEmpresaException:** Lançada quando um produto não pertence à empresa especificada.
- **PedidoEmAbertoException:** Lançada quando já existe um pedido em aberto para o mesmo cliente e restaurante.
- **DonoNaoPodePedidoException:** Lançada quando um dono tenta fazer um pedido para seu próprio restaurante.

- **RemoverProdutoPedidoFechadoException:** Lançada quando um produto é removido de um pedido fechado.

## 2.2.Models

Os modelos representam as entidades principais do sistema:

1. **Produto:** Representa um produto disponível em um restaurante. Possui atributos como id, nome, valor e categoria. A classe também inclui métodos para acessar e modificar esses atributos.

```
1 package br.ufal.ic.p2.myfood;
2
3 import br.ufal.ic.p2.myfood.Exceptions.AtributoNaoExisteException;
4 import br.ufal.ic.p2.myfood.Exceptions.ProdutoNaoEncontradoException;
5
6 import java.io.Serializable;
7
8 public class Produto implements Serializable {
9     private static final long serialVersionUID = 1L;
10
11
12     private static int idCounter = 0; // Gerador de ID automático
13     private int id;
14     private String nome;
15     private float valor;
16     private String categoria;
17
18     public Produto(String nome, float valor, String categoria) {
19         this.id = ++idCounter;
20         this.nome = nome;
21         this.valor = valor;
22         this.categoria = categoria;
23     }
24
25     public int getId() {
26         return id;
27     }
28
29     public String getNome() {
30         return nome;
31     }
32
33     public float getValor() {
34         return valor;
35     }
36
37     public String getCategoria() {
38         return categoria;
39     }
40
41     public String getAtributo(String atributo) throws AtributoNaoExisteException, ProdutoNaoEncontradoException {
42         switch (atributo) {
43             case "nome":
44                 return getNome();
45             case "categoria":
46                 return getCategoria();
47             default:
48                 throw new AtributoNaoExisteException();
49         }
50     }
51
52     public void setNome(String nome) {
53         this.nome = nome;
54     }
55     public void setValor(float valor) {
56         this.valor = valor;
57     }
58     public void setCategoria(String categoria) {
59         this.categoria = categoria;
60     }
61 }
```

2. **Restaurante:** Representa um restaurante que pode ser gerenciado pelos donos. Inclui atributos como id, nome, endereço e tipo de cozinha. Fornece métodos para acessar esses atributos e também para validar se o restaurante pode ser criado ou editado.

```
1 package br.ufal.ic.p2.myfood;
2
3 import br.ufal.ic.p2.myfood.Exceptions.AtributoInvalidoException;
4
5 import java.io.Serializable;
6
7 public class Restaurante implements Serializable {
8     private static final long serialVersionUID = 1L;
9
10    private static int contadorId = 1;
11    private int id;
12    private String nome;
13    private String endereco;
14    private String tipoCozinha;
15
16    public Restaurante(String nome, String endereco, String tipoCozinha) {
17        this.id = contadorId++;
18        this.nome = nome;
19        this.endereco = endereco;
20        this.tipoCozinha = tipoCozinha;
21    }
22
23    public int getId() {
24        return id;
25    }
26
27    public String getNome() {
28        return nome;
29    }
30
31    public String getEndereco() { return endereco; }
32
33    public String getTipoCozinha() { return tipoCozinha; }
34
35
36    public String getAtributo(String atributo) throws AtributoInvalidoException {
37        switch (atributo) {
38            case "nome":
39                return getNome();
40            case "endereco":
41                return getEndereco();
42            case "tipoCozinha":
43                return getTipoCozinha();
44            default:
45                throw new AtributoInvalidoException();
46        }
47    }
48
49 }
```

3. **Usuario:** Classe abstrata que serve como base para os tipos de usuário do sistema, como clientes e donos de restaurantes. Contém atributos como id, nome, email, senha e endereço, além de métodos para acessar esses atributos e verificar permissões.

```
4
5 import java.io.Serializable;
6
7 public abstract class Usuario implements Serializable {
8     private static final long serialVersionUID = 1L;
9
10    private static int contadorId = 1;
11    private int id;
12    private String nome;
13    private String email;
14    private String senha;
15    private String endereco;
16
17    public Usuario(String nome, String email, String senha, String endereco) {
18        this.id = contadorId++;
19        this.nome = nome;
20        this.email = email;
21        this.senha = senha;
22        this.endereco = endereco;
23    }
24    public int getId() {
25        return id;
26    }
27
28    public String getNome() {
29        return nome;
30    }
31
32    public String getEmail() {
33        return email;
34    }
35
36    public String getSenha() {
37        return senha;
38    }
39
40    public String getEndereco() { return endereco; }
41
42    public String getAtributo(String atributo) throws AtributoInvalidoException {
43        switch (atributo) {
44            case "nome":
45                return getNome();
46            case "email":
47                return getEmail();
48            case "senha":
49                return getSenha();
50            case "endereco":
51                return getEndereco();
52            default:
53                throw new AtributoInvalidoException();
54        }
55    }
56
57    public abstract boolean podeCriarEmpresa();
58 }
```



4. **Cliente:** Estende a classe `Usuario` e representa um usuário que faz pedidos em restaurantes. Não possui atributos adicionais além dos herdados da classe `Usuario`.

```
1 package br.ufal.ic.p2.myfood;
2
3 public class Cliente extends Usuario{
4
5     public Cliente(String nome, String email, String senha, String endereco) {
6         super(nome, email, senha, endereco);
7     }
8
9     public boolean podeCriarEmpresa() {
10         return false; // Cliente não pode criar empresas
11     }
12
13 }
```

5. **DonoRestaurante:** Também estende a classe `Usuario` e representa um usuário que é dono de um restaurante. Adiciona o atributo `cpf`, que é obrigatório para donos de restaurantes.

```
1 package br.ufal.ic.p2.myfood;
2
3 import br.ufal.ic.p2.myfood.Exceptions.AtributoInvalidoException;
4
5 public class DonoRestaurante extends Usuario{
6     private String cpf;
7
8     public DonoRestaurante(String nome, String email, String senha, String endereco, String cpf) {
9         super(nome, email, senha, endereco);
10        this.cpf = cpf;
11    }
12
13    public String getCpf() { return cpf; }
14
15    @Override
16    public String getAtributo(String atributo) throws AtributoInvalidoException {
17        if ("cpf".equalsIgnoreCase(atributo)) {
18            return getCpf();
19        }
20        return super.getAtributo(atributo); // Chamando o método da classe pai
21    }
22
23    @Override
24    public boolean podeCriarEmpresa() {
25        return true; // DonoRestaurante pode criar empresas
26    }
27 }
```

6. **Pedido:** Representa um pedido feito por um cliente em um restaurante. Inclui informações como o cliente, a empresa (restaurante), estado do pedido e uma lista de produtos. Possui métodos para adicionar e remover produtos, além de gerenciar o estado do pedido.

```
8 public class Pedido implements Serializable {
14     private String cliente;
15     private String empresa;
16     private String estado;
17     private List<Produto> produtos;
18     private float valor;
19
20     public Pedido(String cliente, String empresa) {
21         this.cliente = cliente;
22         this.empresa = empresa;
23         this.estado = "aberto"; // Estado inicial do pedido
24         this.produtos = new ArrayList<>();
25         this.valor = 0;
26     }
27
28     public int getNumero() {
29         return numero;
30     }
31
32     public String getCliente() {
33         return cliente;
34     }
35
36     public String getEmpresa() {
37         return empresa;
38     }
39
40     public String getEstado() {
41         return estado;
42     }
43
44     public List<Produto> getProdutos() {
45         return produtos;
46     }
47
48     public float getValor() {
49         return valor;
50     }
51
52     public void adicionarProduto(Produto produto) {
53         produtos.add(produto);
54         valor += produto.getValor(); // Atualiza o valor total do pedido
55     }
56
57     public void finalizarPedido() {
58         this.estado = "preparando";
59     }
60
61     public boolean removerProdutoPorNome(String nomeProduto) {
62         for (Iterator<Produto> it = produtos.iterator(); ((Iterator<?>) it).hasNext(); ) {
63             Produto produto = it.next();
64             if (produto.getNome().equals(nomeProduto)) {
65                 it.remove();
66                 valor -= produto.getValor();
67                 return true;
68             }
69         }
70         return false;
71     }
72 }
73
74 }
```

## 2.3.Services

A camada de serviços é responsável pela lógica de negócios e gerenciamento das entidades do sistema. As principais funcionalidades implementadas incluem:

- **Criação e gerenciamento de usuários:** Métodos para criar clientes e donos de restaurantes, realizar login e gerenciar os dados dos usuários.
- **Criação e gerenciamento de restaurantes e produtos:** Métodos para criar novos restaurantes e produtos, editar dados existentes e listar informações.
- **Gestão de pedidos:** Métodos para criar, adicionar produtos a pedidos, fechar pedidos e listar informações sobre pedidos.
- **Persistência de dados:** Utiliza a técnica de serialização para armazenar e recuperar o estado dos objetos em arquivos. Serialização é um processo que transforma um objeto em uma sequência de bytes, permitindo que ele seja salvo em um arquivo ou transmitido pela rede e depois restaurado. Esse método facilita a persistência dos dados, permitindo que o estado dos objetos seja preservado entre diferentes sessões de uso do sistema.

## 2.4.exemplo da utilização das persistências para salvar e carregar dados:

```
1  package br.ufal.ic.p2.myfood.services;
2
3  import br.ufal.ic.p2.myfood.Pedido;
4
5  import java.io.*;
6  import java.util.HashMap;
7  import java.util.List;
8  import java.util.Map;
9
10 public class PedidoPorRestauranteSave {
11
12     private static final String FILE_PATH = "pedidorPorRestaurante.dat";
13
14     public static void salvarPedidosPorRestaurante(Map<Integer, List<Pedido>> pedidosPorRestaurante) throws IOException {
15         try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(FILE_PATH))) {
16             oos.writeObject(pedidosPorRestaurante);
17         }
18     }
19
20     @SuppressWarnings("unchecked")
21     public static Map<Integer, List<Pedido>> carregarPedidosPorRestaurante() throws IOException,
22         ClassNotFoundException {
23         File file = new File(FILE_PATH);
24         if (!file.exists()) {
25             return new HashMap<>();
26         }
27         try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(FILE_PATH))) {
28             return (Map<Integer, List<Pedido>>) ois.readObject();
29         }
30     }
31 }
```

### 3.PADRÃO DE PROJETO FACADE

O padrão de projeto Facade é utilizado para fornecer uma interface simplificada e unificada para um conjunto de interfaces em um subsistema. Ele define uma interface de alto nível que torna o subsistema mais fácil de usar.

No contexto do projeto MyFood, o padrão Facade é implementado pela classe Sistema. Esta classe atua como uma fachada que encapsula a complexidade dos serviços e operações relacionadas a usuários, restaurantes, produtos e pedidos. Através da classe Sistema, os usuários podem interagir com o sistema de forma mais direta e simplificada, sem precisar se preocupar com a complexidade interna das operações.

#### 3.1.Importância do Padrão Facade

- **Simplificação:** A classe Sistema oferece uma interface simplificada para realizar operações complexas, como criar usuários, gerenciar restaurantes e processar pedidos. Isso facilita a interação com o sistema e reduz a necessidade de entender a complexidade dos detalhes internos.
- **Encapsulamento:** O padrão Facade ajuda a encapsular as operações complexas e a lógica de negócios em uma única classe. Isso promove uma separação clara entre a lógica de aplicação e a interface do usuário.
- **Facilidade de Manutenção:** Com uma fachada centralizada, qualquer alteração na lógica de negócios ou na estrutura do subsistema pode ser feita de forma mais controlada, minimizando o impacto nas interações do usuário com o sistema.

Em resumo, o padrão Facade foi crucial para tornar o sistema MyFood mais acessível e fácil de utilizar, fornecendo uma interface unificada e simplificada para interações com as funcionalidades complexas do sistema.

