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

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:** Lançado quando o endereço da empresa já está cadastrado.

- **NomeProdutoExisteException:** Lançada quando um produto com o mesmo nome já existe.

- **ProdutoNaoEncontradoException:** Lançado 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.

```java
package br.ufal.ic.p2.myfood;

import br.ufal.ic.p2.myfood.Exceptions.AtributoNaoExisteException;
import br.ufal.ic.p2.myfood.Exceptions.ProdutoNaoEncontradoException;

import java.io.Serializable;

public class Produto implements Serializable {
    private static final long serialVersionUID = 1L;


    private static int idCounter = 0; // Gerador de ID automático
    private int id;
    private String nome;
    private float valor;
    private String categoria;

    public Produto(String nome, float valor, String categoria) {
        this.id = ++idCounter;
        this.nome = nome;
        this.valor = valor;
        this.categoria = categoria;
    }

    public int getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public float getValor() {
        return valor;
    }

    public String getCategoria() {
        return categoria;
    }

    public String getAtributo(String atributo) throws AtributoNaoExisteException, ProdutoNaoEncontradoException {
        switch (atributo) {
            case "nome":
                return getNome();
            case "categoria":
                return getCategoria();
            default:
                throw new AtributoNaoExisteException();
        }
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
    public void setValor(float valor) {
        this.valor = valor;
    }
    public void setCategoria(String categoria) {
        this.categoria = categoria;
    }
}
```

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.

```java
package br.ufal.ic.p2.myfood;

import br.ufal.ic.p2.myfood.Exceptions.AtributoInvalidoException;

import java.io.Serializable;

public class Restaurante implements Serializable {
    private static final long serialVersionUID = 1L;

    private static int contadorId = 1;
    private int id;
    private String nome;
    private String endereco;
    private String tipoCozinha;

    public Restaurante(String nome, String endereco, String tipoCozinha) {
        this.id = contadorId++;
        this.nome = nome;
        this.endereco = endereco;
        this.tipoCozinha = tipoCozinha;
    }

    public int getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public String getEndereco() { return endereco; }

    public String getTipoCozinha() { return tipoCozinha; }


    public String getAtributo(String atributo) throws AtributoInvalidoException {
        switch (atributo) {
            case "nome":
                return getNome();
            case "endereco":
                return getEndereco();
            case "tipoCozinha":
                return getTipoCozinha();
            default:
                throw new AtributoInvalidoException();
        }
    }

}
```

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.

```java

import java.io.Serializable;

public abstract class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;

    private static int contadorId = 1;
    private int id;
    private String nome;
    private String email;
    private String senha;
    private String endereco;

    public Usuario(String nome, String email, String senha, String endereco) {
        this.id = contadorId++;
        this.nome = nome;
        this.email = email;
        this.senha = senha;
        this.endereco = endereco;
    }
    public int getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    public String getSenha() {
        return senha;
    }

    public String getEndereco() { return endereco; }

    public String getAtributo(String atributo) throws AtributoInvalidoException {
        switch (atributo) {
            case "nome":
                return getNome();
            case "email":
                return getEmail();
            case "senha":
                return getSenha();
            case "endereco":
                return getEndereco();
            default:
                throw new AtributoInvalidoException();
        }
    }

    public abstract boolean podeCriarEmpresa();
}
```

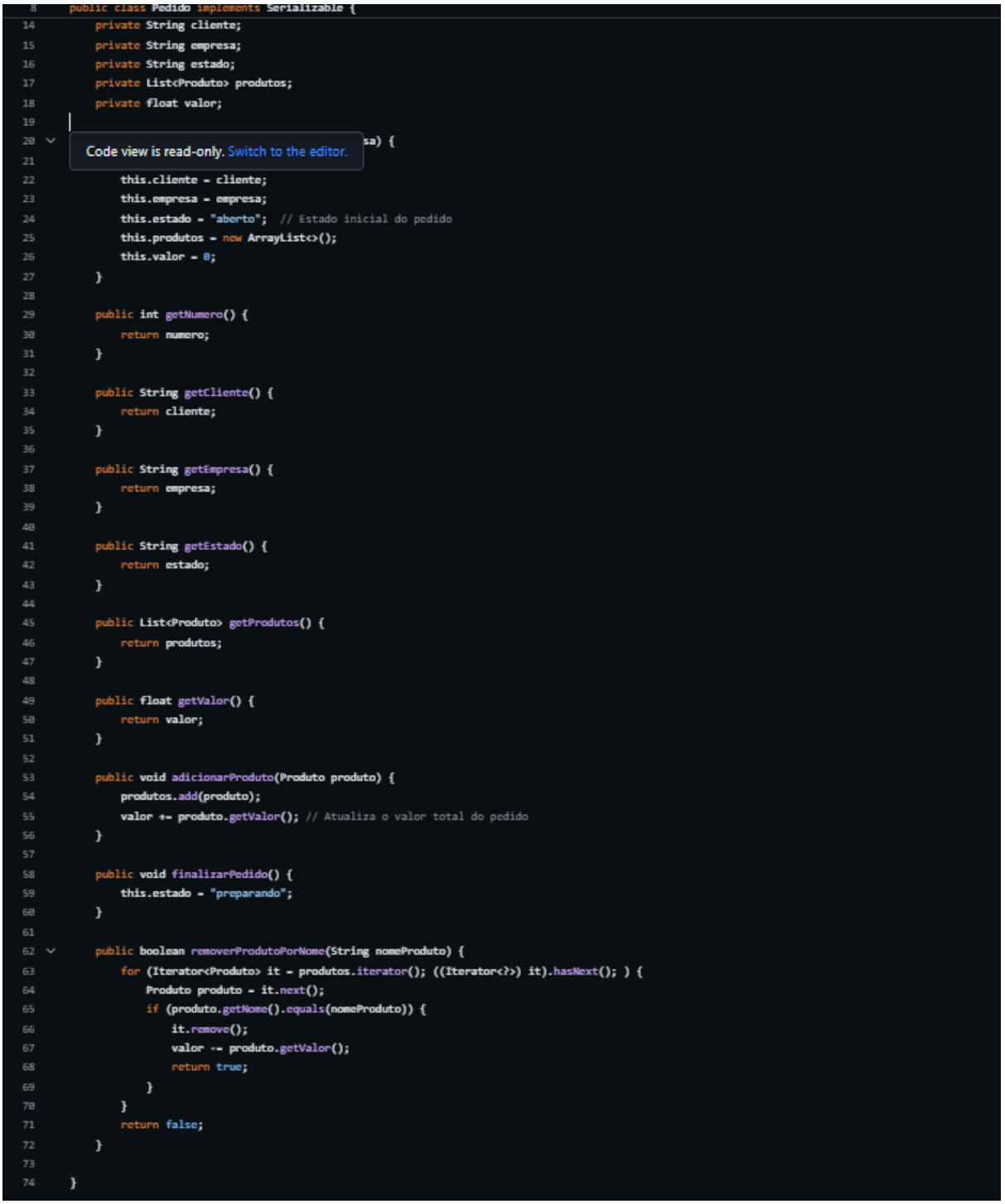
4. **Cliente:** Extende a classe Usuario e representa um usuário que faz pedidos em restaurantes. Não possui atributos adicionais além dos herdados da classe Usuario.

```java
package br.ufal.ic.p2.myfood;

public class Cliente extends Usuario{

    public Cliente(String nome, String email, String senha, String endereco) {
        super(nome, email, senha, endereco);
    }

    public boolean podeCriarEmpresa() {
        return false; // Cliente não pode criar empresas
    }

}
```

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.

```java
package br.ufal.ic.p2.myfood;

import br.ufal.ic.p2.myfood.Exceptions.AtributoInvalidoException;

public class DonoRestaurante extends Usuario{
    private String cpf;

    public DonoRestaurante(String nome, String email, String senha, String endereco, String cp
        super(nome, email, senha, endereco);
        this.cpf = cpf;
    }

    public String getCpf() { return cpf; }

    @Override
    public String getAtributo(String atributo) throws AtributoInvalidoException {
        if ("cpf".equalsIgnoreCase(atributo)) {
            return getCpf();
        }
        return super.getAtributo(atributo);  // Chamando o método da classe pai
    }

    @Override
    public boolean podeCriarEmpresa() {
        return true; // DonoRestaurante pode criar empresas
    }
}
```

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.



```
public class Pedido implements Serializable {
    private String cliente;
    private String empresa;
    private String estado;
    private List<Produto> produtos;
    private float valor;

    Code view is read-only. Switch to the editor.

        this.cliente = cliente;
        this.empresa = empresa;
        this.estado = "aberto";  // Estado inicial do pedido
        this.produtos = new ArrayList<>();
        this.valor = 0;
    }

    public int getNumero() {
        return numero;
    }

    public String getCliente() {
        return cliente;
    }

    public String getEmpresa() {
        return empresa;
    }

    public String getEstado() {
        return estado;
    }

    public List<Produto> getProdutos() {
        return produtos;
    }

    public float getValor() {
        return valor;
    }

    public void adicionarProduto(Produto produto) {
        produtos.add(produto);
        valor += produto.getValor(); // Atualiza o valor total do pedido
    }

    public void finalizarPedido() {
        this.estado = "preparando";
    }

    public boolean removerProdutoPorNome(String nomeProduto) {
        for (Iterator<Produto> it = produtos.iterator(); ((Iterator<?>) it).hasNext(); ) {
            Produto produto = it.next();
            if (produto.getNome().equals(nomeProduto)) {
                it.remove();
                valor -= produto.getValor();
                return true;
            }
        }
        return false;
    }

}
```

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:

```java
package br.ufal.ic.p2.myfood.services;

import br.ufal.ic.p2.myfood.Pedido;

import java.io.*;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class PedidoPorRestauranteSave {

    private static final String FILE_PATH = "pedidorPorRestaurante.dat";

    public static void salvarPedidosPorRestaurante(Map<Integer, List<Pedido>> pedidosPorRestaurante) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(FILE_PATH))) {
            oos.writeObject(pedidosPorRestaurante);
        }
    }

    @SuppressWarnings("unchecked")
    public static Map<Integer, List<Pedido>> carregarPedidosPorRestaurante() throws IOException,
            ClassNotFoundException {
        File file = new File(FILE_PATH);
        if (!file.exists()) {
            return new HashMap<>();
        }
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(FILE_PATH))) {
            return (Map<Integer, List<Pedido>>) ois.readObject();
        }
    }
}
```

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.