UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO/IC

PROGRAMAÇÃO 2

RELATÓRIO - MYFOOD

DISCENTE:

João Vitor Alves da Silva

NOVEMBRO DE 2025

MACEIÓ - AL

1. Descrição geral do design arquitetural do sistema

O design que eu escolhi para o MyFood é baseado em uma arquitetura de camadas com clara separação de responsabilidades. A ideia principal foi separar totalmente a lógica de negócio (as regras) dos dados (os modelos).

A arquitetura tem três camadas principais:

1. **Camada de Fachada (Facade):** É o ponto de entrada único do sistema. O EasyAccept (nosso "cliente") só se comunica com essa camada.
2. **Camada de Controladores (Lógica):** É o "cérebro" do sistema. Toda a lógica de negócio, validações e regras estão aqui. Ela é dividida por "contexto" (Usuários, Empresas, Produtos, Pedidos).
3. **Camada de Modelos (Dados):** São as classes que apenas guardam dados, como Usuario, Restaurante, etc. Elas não têm lógica de negócio.

Para a persistência, como a descrição do projeto sugeria, o sistema salva os dados em arquivos. Eu optei por usar a interface Serializable do Java. Cada controlador é responsável por salvar e carregar seus próprios dados em um arquivo .dat separado (ex: myfood_data.dat, empresas_data.dat), usando ObjectOutputStream e ObjectInputStream. Isso acontece quando o sistema inicia (no construtor do controlador) e quando ele encerra (no método encerrarSistema da Facade).

2. Principais componentes e suas interações

O sistema é dividido nestes componentes principais:

1. Facade (Facade.java)

É a "porta da frente" do projeto, como exigido na descrição. É a única classe que o EasyAccept acessa. Ela não contém nenhuma lógica de negócio; sua única função é receber as chamadas e repassá-las para o controlador correto.

2. Controladores (Os "Cérebros")

É onde toda a lógica do sistema acontece. Cada controlador gerencia um domínio específico e funciona como um "Singleton" dentro da Facade (explico melhor na seção de padrões).

- ControladorUsuario.java: Gerencia tudo sobre usuários (criação, login, busca, persistência).
- ControladorEmpresa.java: Gerencia a criação e busca de restaurantes.
- ControladorProduto.java: Gerencia os produtos de cada empresa.

- ControladorPedido.java: Gerencia a criação de pedidos, adição de produtos ao carrinho e fechamento.

3. Modelos (Os Dados)

São as classes "burras" que só armazenam os dados. Todas implementam Serializable para poderem ser salvas em arquivo.

- Usuario.java (classe abstrata), Cliente.java, DonoDeEmpresa.java
- Restaurante.java
- Produto.java
- Pedido.java

4. Interações

O fluxo de uma operação é quase sempre o mesmo:

1. O EasyAccept (cliente) chama um método na Facade (ex: criarPedido).
2. A Facade.java imediatamente chama o método correspondente no controlador (controladorPedido.criarPedido(...)).
3. O ControladorPedido executa a lógica. Para isso, ele pode precisar falar com outros controladores:
   - Ele pergunta ao ControladorUsuario se o ID do cliente é de um DonoDeEmpresa.
   - Ele checa seus próprios mapas para ver se já existe um pedido aberto.
4. Se tudo estiver OK, o ControladorPedido cria um novo objeto new Pedido(...) e o armazena em seus mapas.
5. O controlador retorna a resposta (o ID do pedido) para a Facade, que a retorna para o EasyAccept.

3. Padrões de Projeto Adotados

Durante o desenvolvimento, eu identifiquei a necessidade e apliquei dois padrões de projeto principais.

1. Padrão de Projeto: Facade (Fachada)

- **Descrição Geral:** O padrão Facade fornece uma interface unificada e simplificada para um conjunto de interfaces de um subsistema (no caso, todos os nossos controladores). Ele esconde a complexidade do sistema e facilita o uso.
- **Problema Resolvido:** Ele resolve o problema de acoplamento entre o cliente e o sistema. Sem ele, o EasyAccept precisaria saber da existência do ControladorUsuario, ControladorPedido, etc., e como inicializar e interagir com cada um deles.
- **Identificação da Oportunidade:** A oportunidade foi óbvia, pois a própria descrição do

projeto nos obrigava a criar uma classe Facade para ser o ponto de entrada dos testes de aceitação. Isso é exatamente o que o padrão Facade propõe.
- **Aplicação no Projeto:** A classe myfood.Facade é a aplicação direta desse padrão. Ela é a única classe public no pacote myfood (além dos modelos) que o "mundo exterior" (EasyAccept) vê. Quando o teste chama facade.criarUsuario(...), ele não sabe que, por baixo dos panos, a Facade está simplesmente repassando a chamada para this.controladorUsuario.criarCliente(...).

2. Padrão de Projeto: Singleton (Instância Única)

- **Descrição Geral:** O padrão Singleton garante que uma classe tenha apenas *uma* instância e fornece um ponto de acesso global a essa instância.
- **Problema Resolvido:** Ele resolve o problema de ter um estado global compartilhado. Você usa ele quando precisa que uma única instância de um objeto gerencie um recurso (como um banco de dados, ou, no meu caso, os mapas de dados).
- **Identificação da Oportunidade:** Eu percebi que precisava de um lugar central para armazenar *todos* os usuários, *todas* as empresas, etc. Por exemplo, os mapas usuariosPorId e usuariosPorEmail no ControladorUsuario. Se a cada chamada do EasyAccept um *novo* ControladorUsuario fosse criado, os dados se perderiam, pois os mapas estariam sempre vazios.
- **Aplicação no Projeto:** Os meus Controladores (ControladorUsuario, ControladorEmpresa, etc.) são implementados como Singletons. No construtor da Facade, eu crio as instâncias *únicas* de cada controlador: this.controladorUsuario = new ControladorUsuario();. A partir daí, todas as chamadas do EasyAccept (como criarUsuario, login, criarEmpresa) usam *essas mesmas instâncias* dos controladores, garantindo que todos os dados sejam mantidos nos mesmos mapas e o estado do sistema seja consistente.