**Trabalho Prático DCC025 - Orientação a Objetos**

Professor: Gleiph Ghiotto Lima de Menezes

Autores:

- Giovanni Almeida Dutra - 202065032AB

- Quezia Emanuelly da Silva Oliveira - 202265561C

- Augusto Benedito Carvalho da Silva - 202265552C

Link do repositório: https://github.com/queziayxz/Sistema\_Gerenciamento\_PizzaPlanet

**1 - Objetivo**

O presente trabalho tem como objetivo a implementação de um sistema de gerenciamento que aborda os conteúdos da disciplina orientação a objetos, utilizando a linguagem de programação java e de recursos como classes, pacotes, herança, polimorfismo, interfaces, coleções, tratamento de exceções, interface gráfica e leitura/escrita em arquivos.

**2 -  Domínio da aplicação**

Consiste no desenvolvimento de um sistema de gerenciamento de uma pizzaria, contando com funcionalidades como: Cadastro de Clientes, Login de Clientes e Administradores, Gerenciamento de Produtos e Pizzas, Visualização de Pedidos e outras diversas, tanto para o Cliente, quanto para os Administradores. Os principais requisitos desse sistema estarão listados abaixo:

**Para o Cliente:**

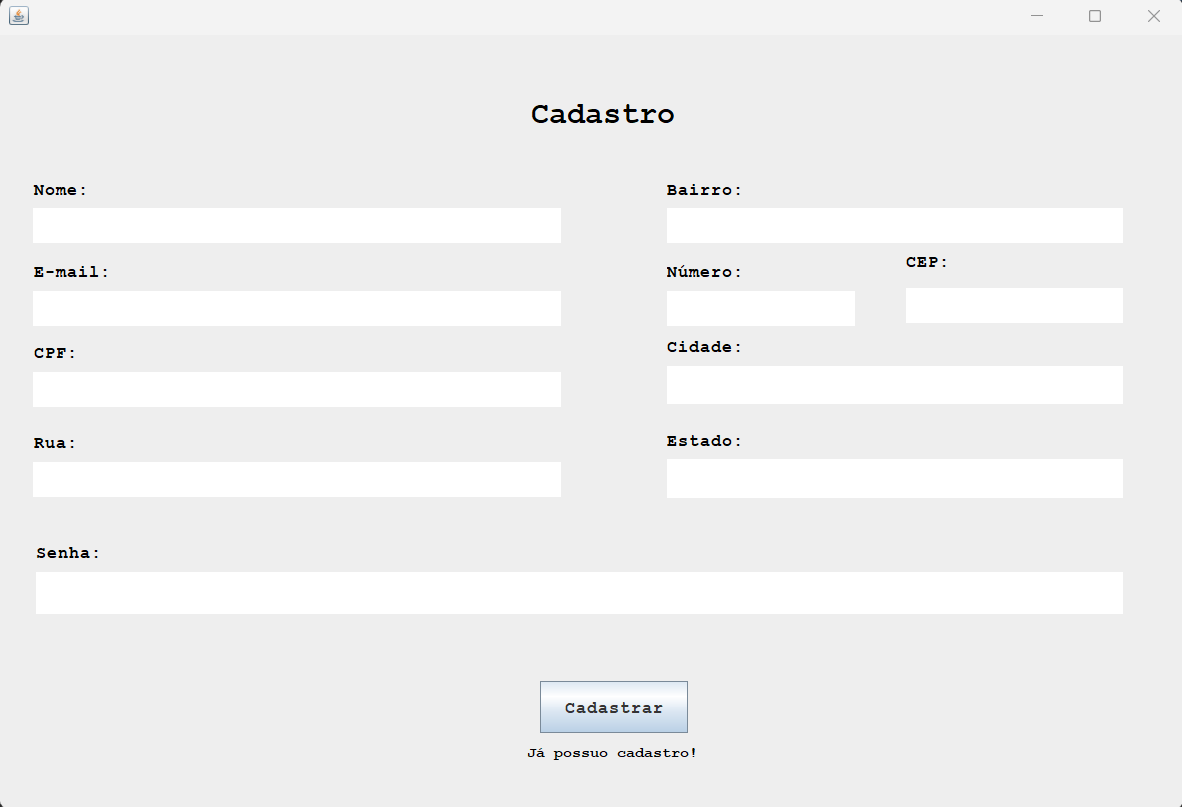
* Cadastro único por usuário, utilizando o CPF como “ID pessoal”.
* Carrinho de compras, possibilitando o usuário comprar além de pizzas(produto principal), acompanhamentos.
* Possibilidade de pagamento de diferentes formas, alterando o valor final para o Cliente, com base na escolha do mesmo.

**Para o Administrador:**

* Janelas separadas para controle de Produtos, Ingredientes, Pizzas e Formas de pagamento.
* Operações de Criar, Atualizar, Visualizar e Deletar para Produtos, Ingredientes, Pizzas e Formas de pagamento.
* Registro em arquivos, que atuarão como banco de dados, que confere maior segurança e praticidade.

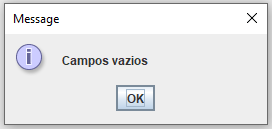
**3 – Interface**

A seguir, imagens de todas as seções do programa com detalhamento:



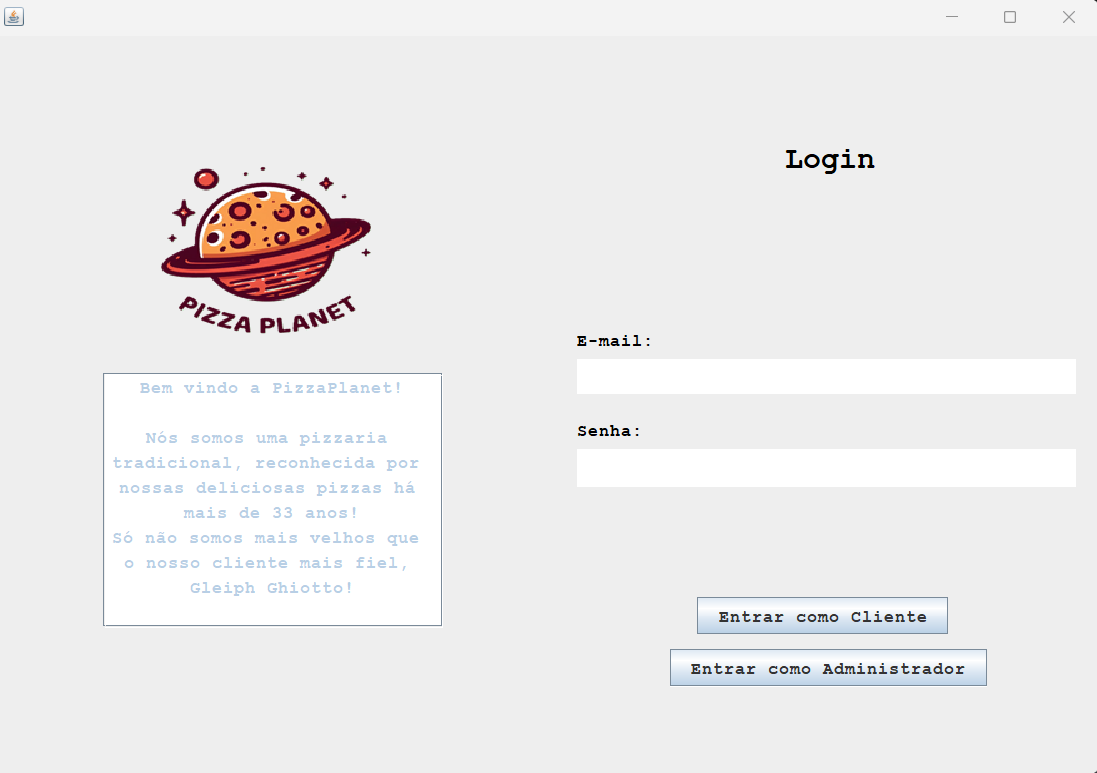
Tela Inicial (Cadastro de Usuários)

Caso o usuário que esteja se cadastrando, tenha esquecido de preencher algum campo, um pop-up irá alerta-lo:



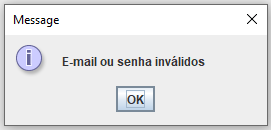
Pop-up de campo vazio

Caso o usuário já tenha cadastro, basta clicar em “Já possuo cadastro!” para ser redirecionado para a tela de login:



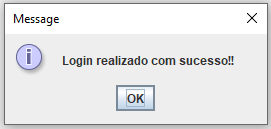
Tela de Login

Caso o usuário não preencha algum dos campos e tente prosseguir com o login, o Pop-up de campo vazio irá alerta-lo. Caso o usuário preencha as credenciais de forma incorreta, o seguinte Pop-up:

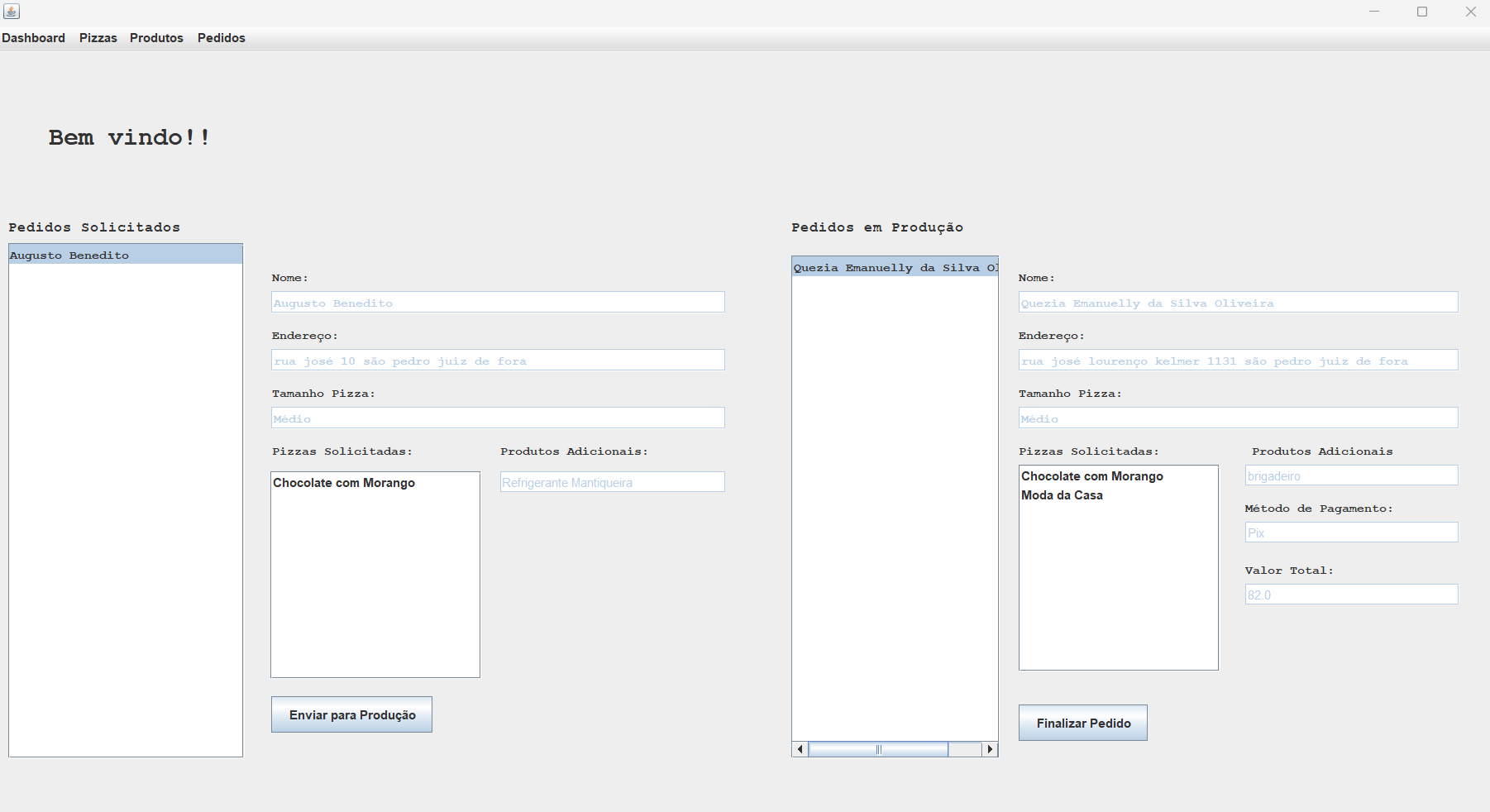


Pop-up de login inválido

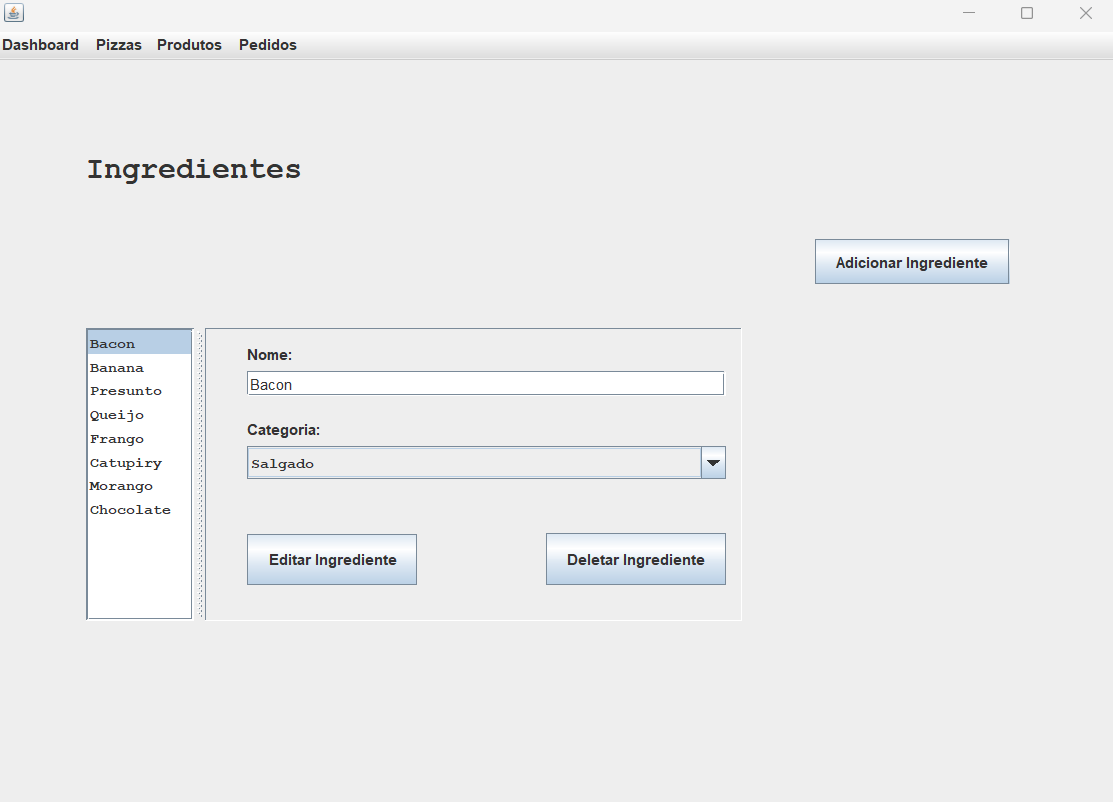
Caso o login seja bem sucedido, um Pop-up irá alertar e o usuário será redirecionado para a próxima página:



Pop-up de login realizado



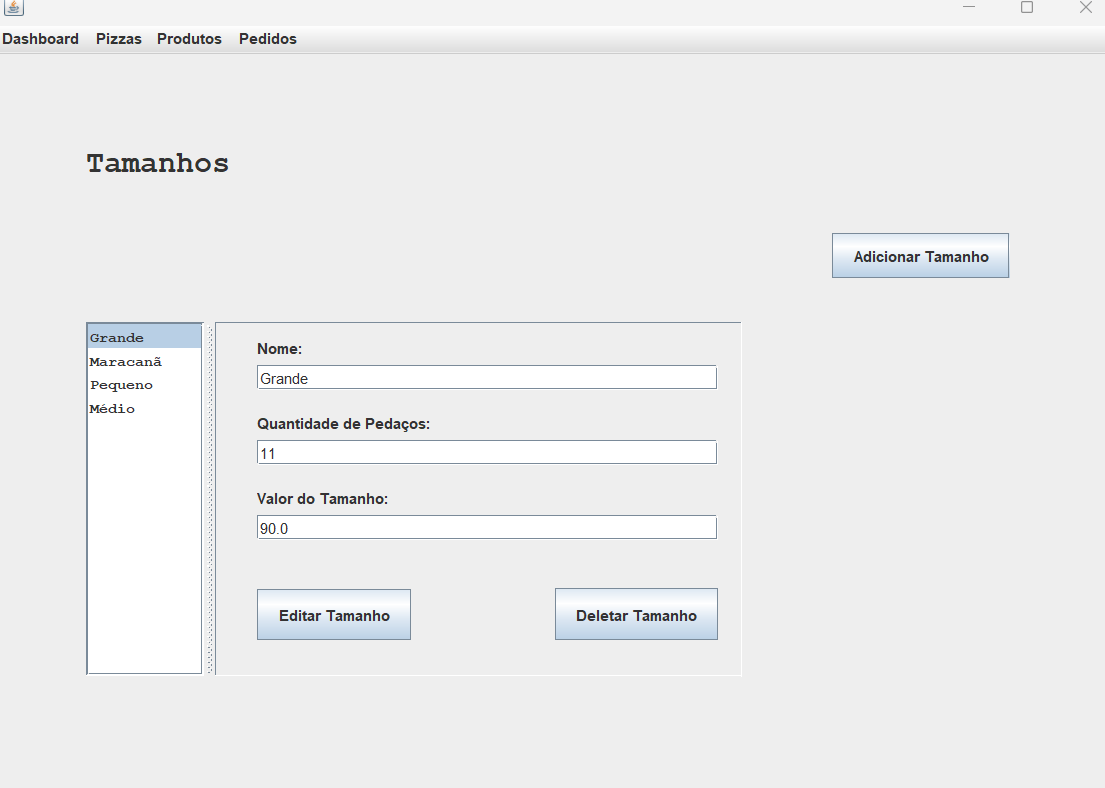
Tela Dashboard - Administrativo



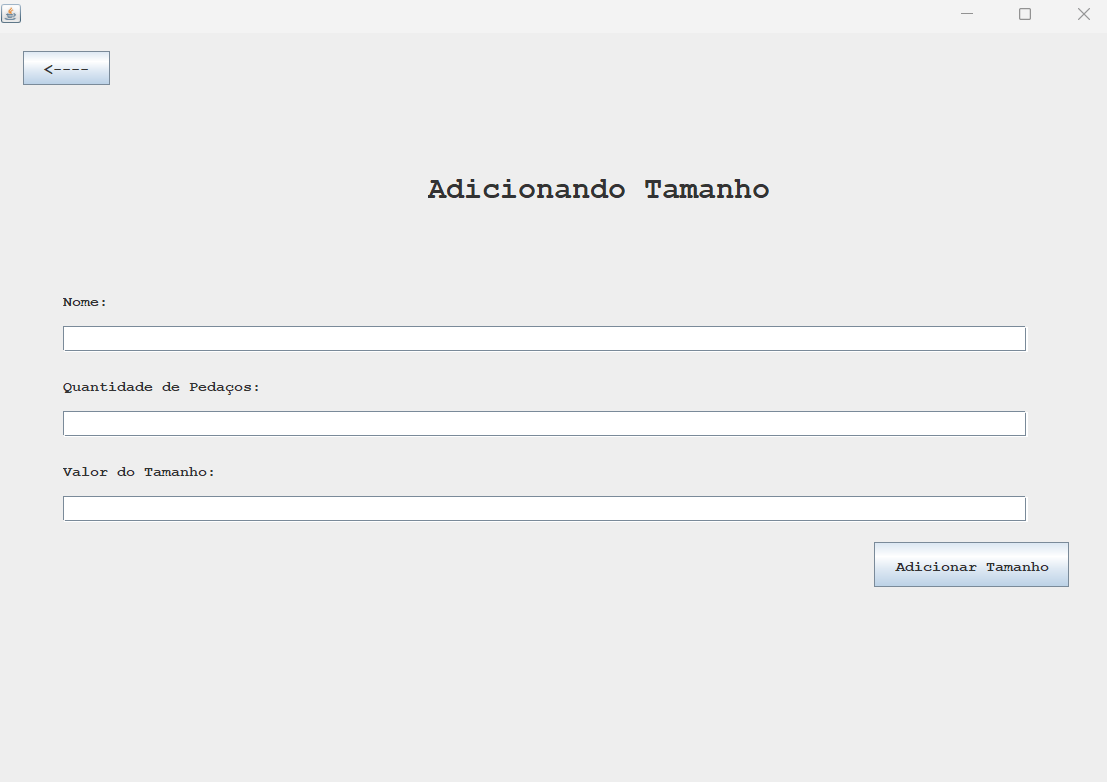
Tela de Gerenciamento de Ingredientes – Administrativo



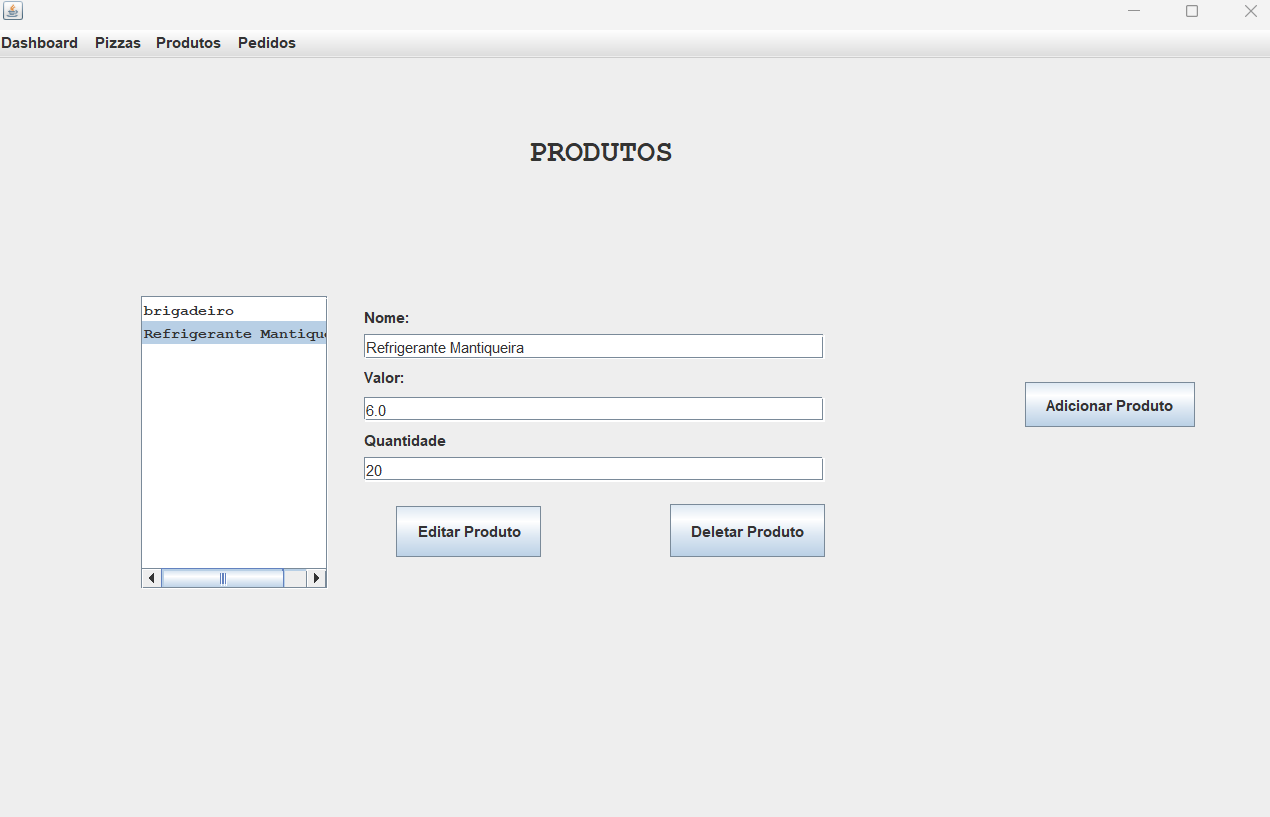
Tela de Cadastro de Ingrediente – Administrativo



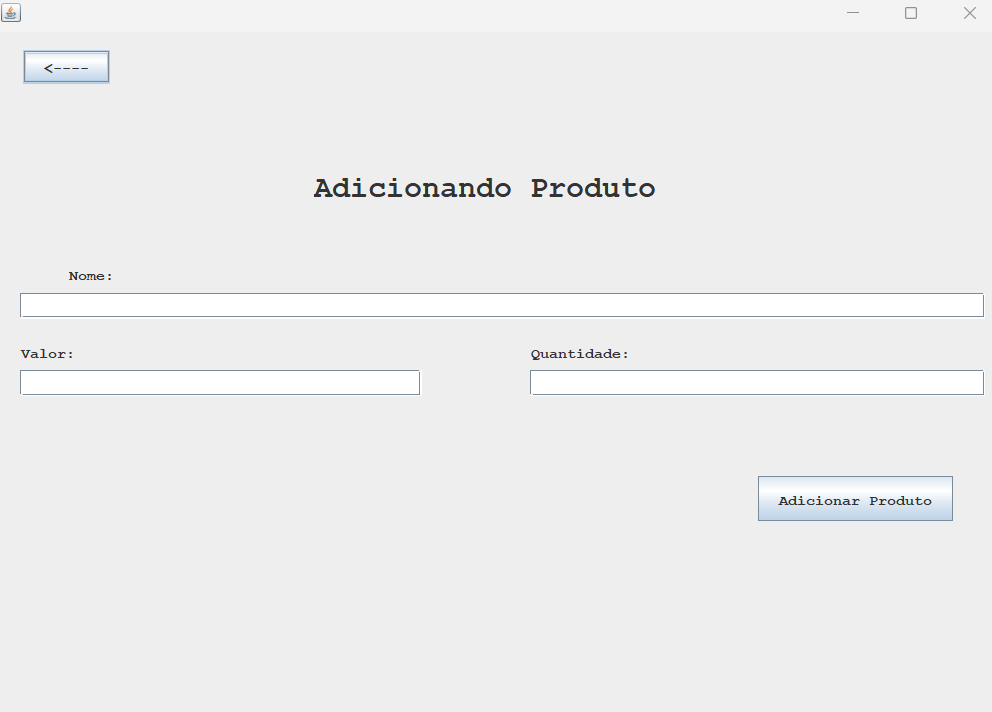
Tela de Gerenciamento de Tamanhos – Administrativo



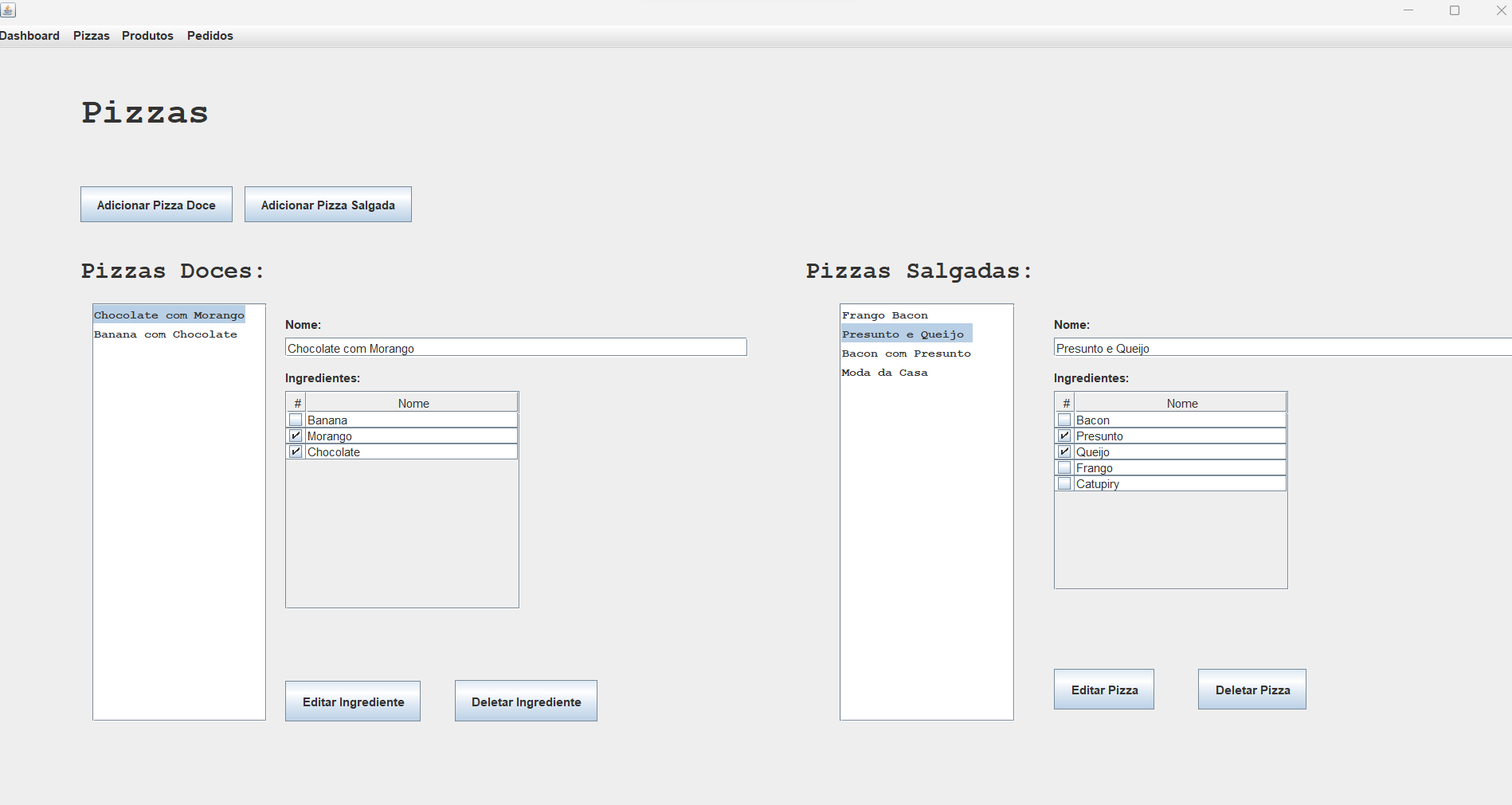
Tela de Cadastro de Tamanho – Administrativo



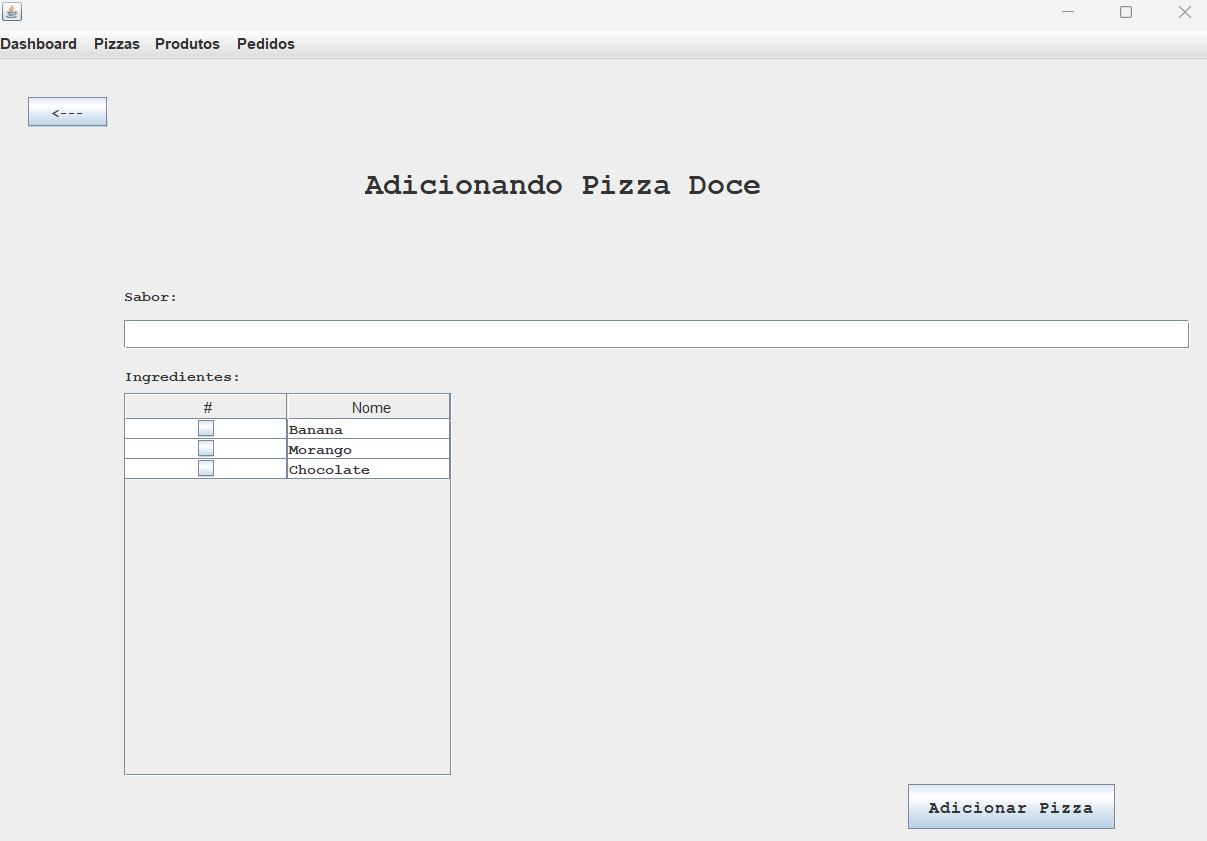
Tela de Gerenciamento de Produtos – Administrativo



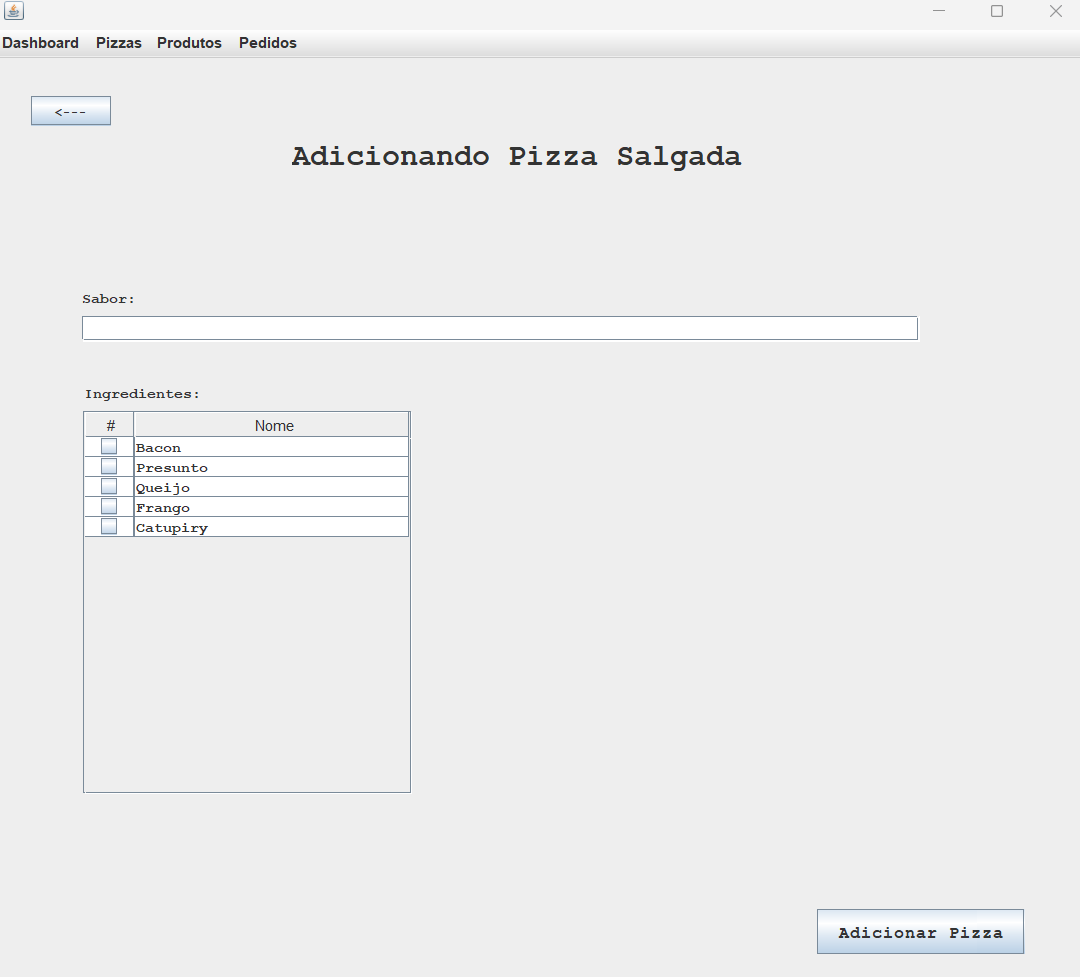
Tela de Cadastro de Produto – Administrativo



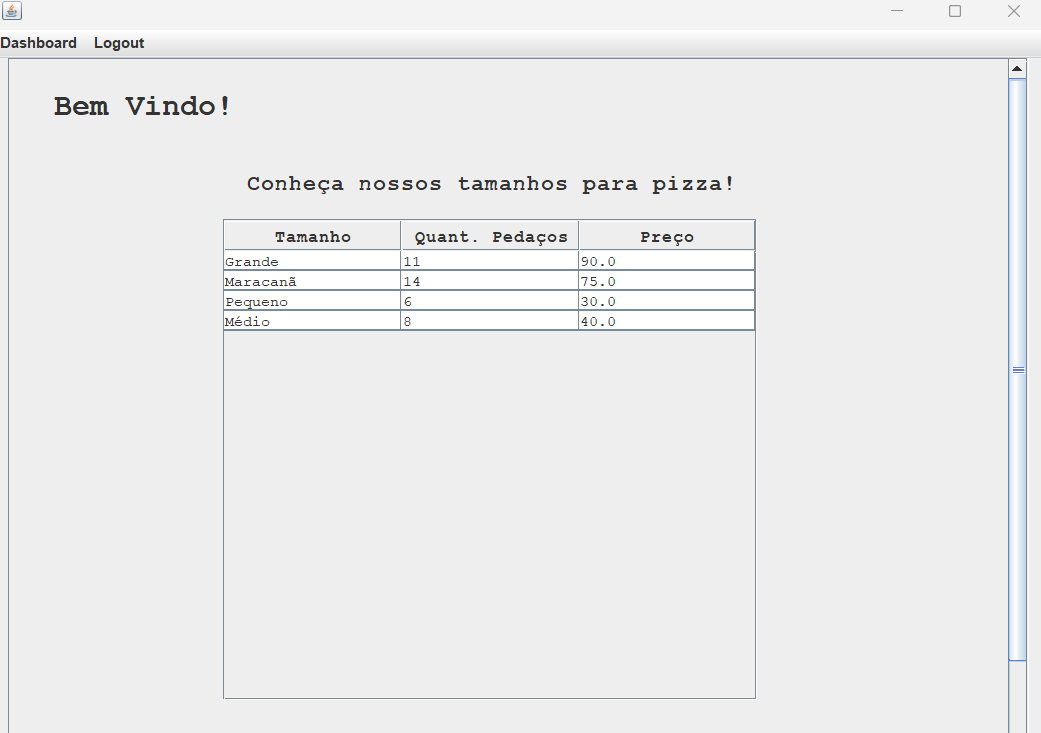
Tela de Gerenciamento de Pizzas – Administrativo



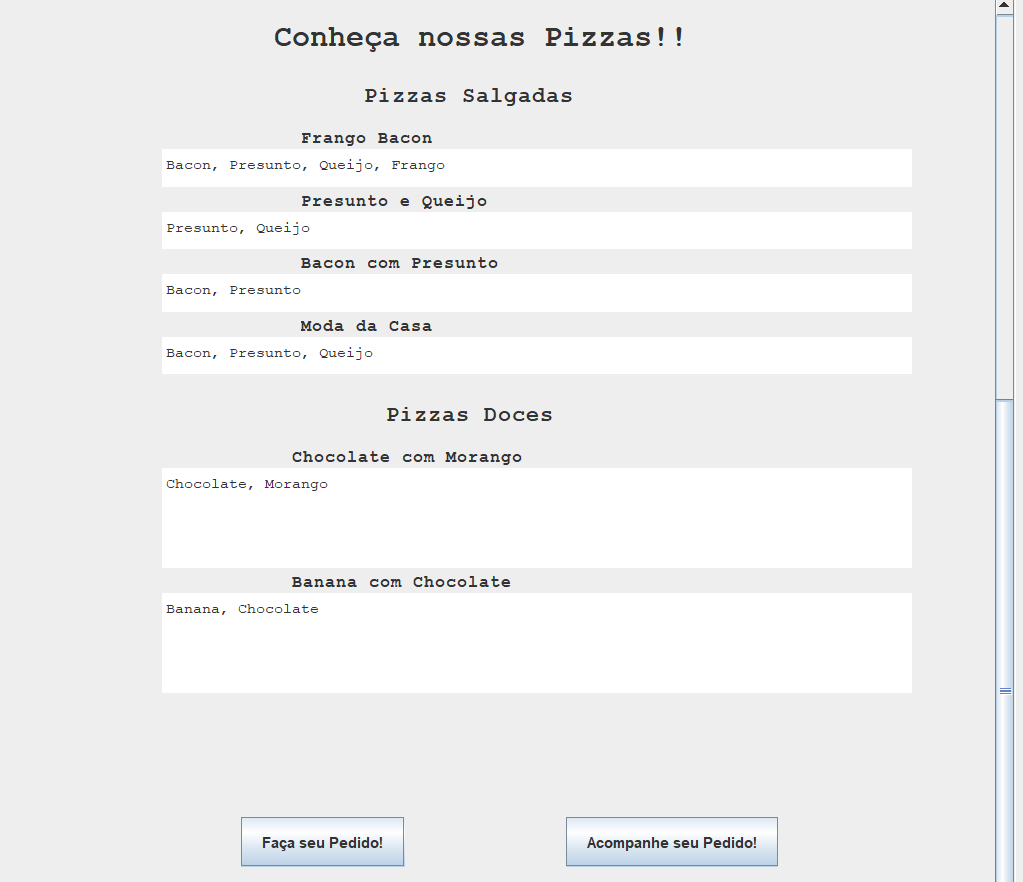
Tela de Cadastro de Pizza Doce – Administrativo



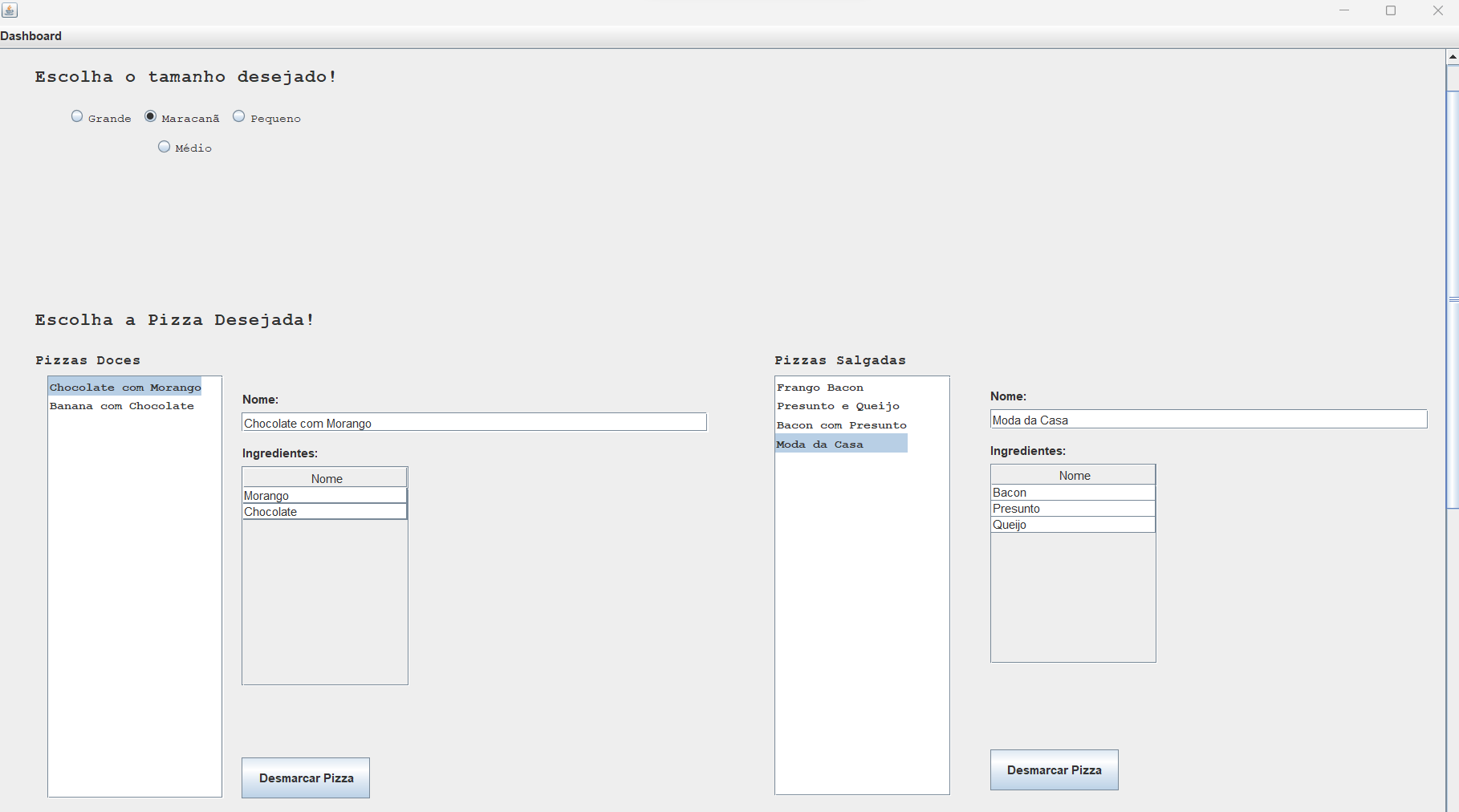
Tela de Cadastro de Pizza Salgada – Administrativo



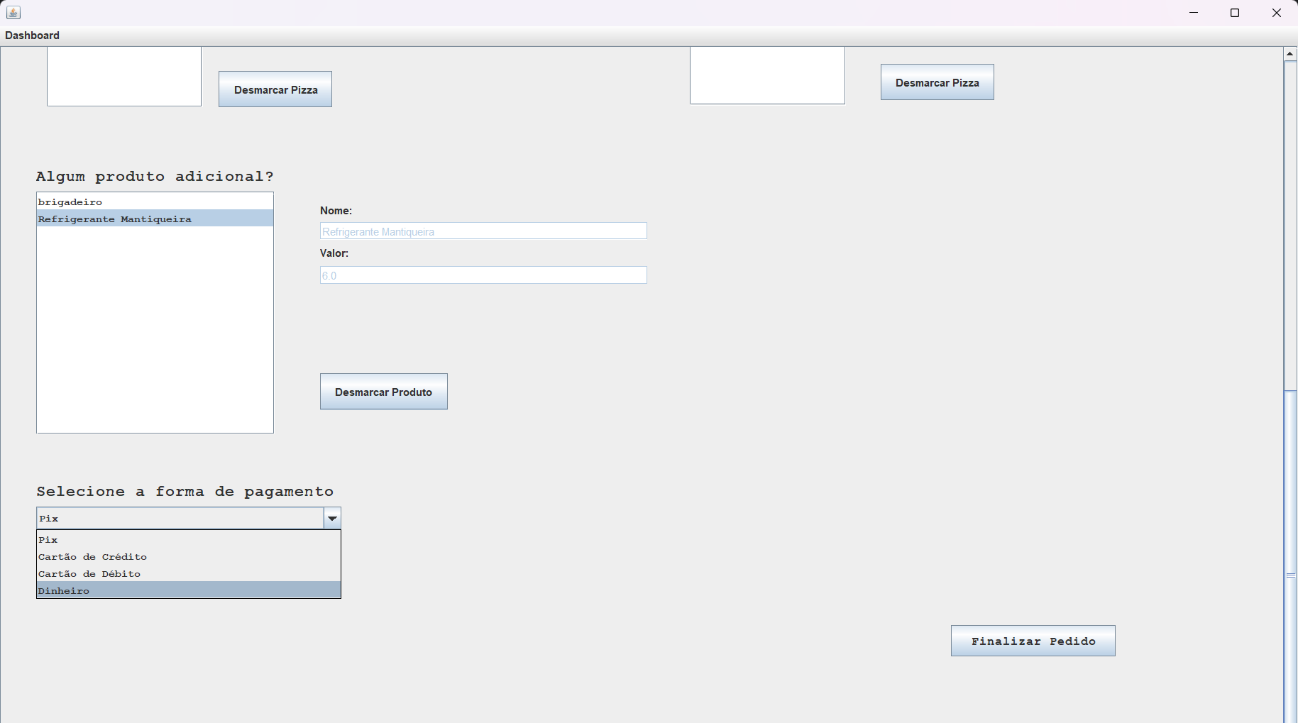
Tela Dashboard – Cliente



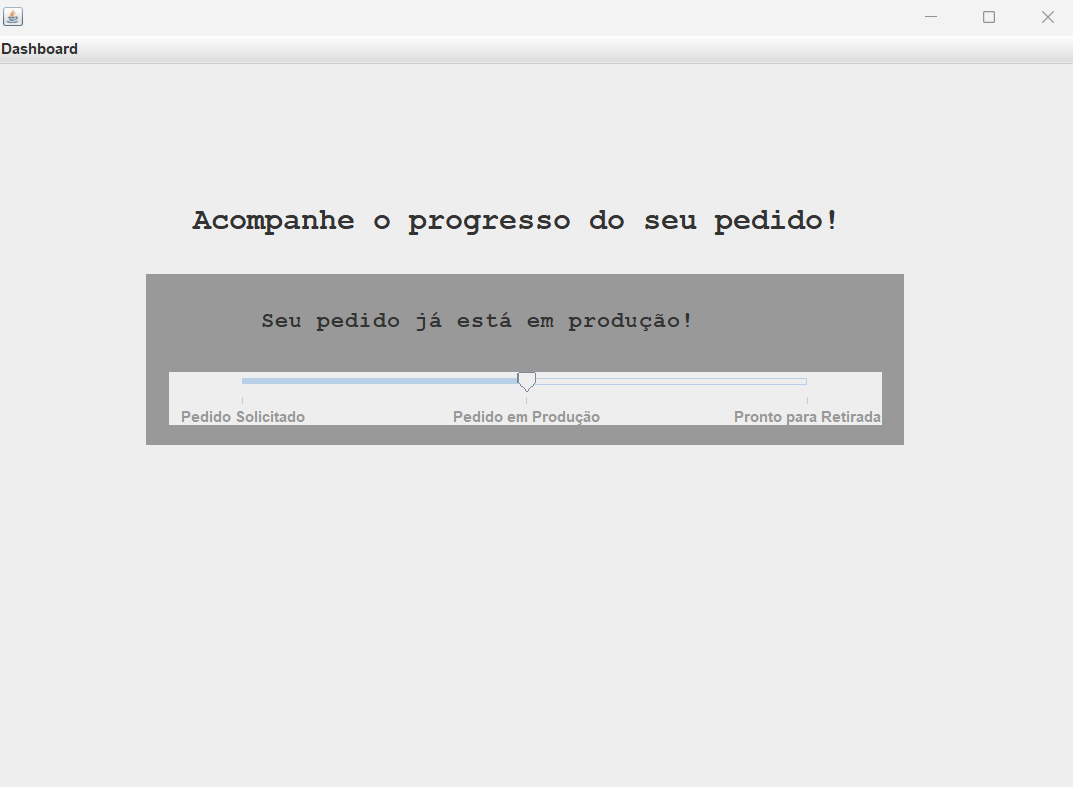
Tela Dashboard – Cliente



Tela Solicitar Pedido – Cliente



Tela Solicitar Pedido – Cliente



Tela Acompanhar Pedido - Cliente

**4 - Classes e Pacotes**

O conceito de **Herança** foi utilizado para evitar atribuições desnecessárias, como no caso do Usuário, que possui atributos necessários tanto para o **Cliente**, quanto para o **Administrador**. Com a utilização desse conceito, foi possível evitar código desnecessário.

O conceito de **Polimorfismo** foi utilizado para evitar declarações de métodos desnecessárias. Foi utilizada juntamente com o conceito de **Classe Abstrata** e **Herança** em **Pagamento** (sendo a classe abstrata, com funções) e em **Pix**, **Dinheiro**, **CartaoDebito** e **CartaoCredito** (sendo as classes concretas, com sobrescrição das funções declaradas em Pagamento).

O conceito de **Coleções** foi utilizado em conjunto com **Tratamento de Exceções**, **Leitura** e **Escrita** de arquivos para armazenar os dados do banco de dados, em tempo de execução, manipula-los, e atualizar os bancos. Caso houvesse algum equívoco ao passar os dados para o banco, o mesmo irá lançar a exceção correspondente.

O conceito de **Interface Gráfica** foi utilizado para facilitar a navegação tanto para os **Clientes** quanto para o **Administrador**.

O sistema foi estruturado a partir da arquitetura **MVC** (Model – View – Controller). Baseado nisso, dentro de **Model**, temos:

**Class Usuario:**

* + Classe abstrata que define alguns atributos comuns à Administrador e Cliente. Seguem os tipos e atributos: String Nome, String Email, String Senha e CPF Cpf. Essa classe também é responsável pela validação de “inputs” vazios, tanto no login como no cadastro.

**Class Administrador:**

* + Classe concreta que herda os campos da Classe Usuario e define o atributo List<Administrador> listaAdm, que irá ser usado para armazenar os administradores presentes no banco de dados, em tempo de execução.

**Class Cliente:**

* + Classe concreta que herda os campos da Classe Usuario e define os atributos List<Cliente> listaCliente que irá ser usado para armazenar os administradores presentes no banco de dados, em tempo de execução e Endereco endereco, que será utilizado para determinar o valor da entrega do pedido.

**Class CPF:**

* + Classe concreta que é responsável por validar e armazenar o CPF de um Cliente, quando o mesmo estiver em processo de criação de usuário do sistema. Tal validação consiste em verificar se o CPF realmente é válido ou se é inválido. Caso inválido, é lançada uma exceção “CPFInvalido”.

**Class Endereco:**

* + Classe concreta que é responsável por validar e armazenar o endereço que o usuário cadastre. Os atributos dessa classe são: Rua, Número, Bairro, CEP, Cidade e Estado, todos do tipo String. Tal validação consiste em verificar os campos, verificando se eles estão vazios. Caso positivo, é lançada uma exceção “CampoVazio”. Além disso, é verificado o campo “Número” e caso seja inválido, também é lançada uma exceção “EnderecoInvalido”.

**Class Pagamento:**

* + Classe abstrata que define alguns atributos e métodos comuns às formas de pagamento possíveis no sistema. Os tipos, seguidos dos nomes dos atributos são: double Valor, List<Pagamento> listaPix e List<Pagamento> listaCartao. Vale ressaltar que tanto listaPix quanto listaCartao são estáticos e serão responsáveis por armazenar os dados presentes no banco de dados, em tempo de execução. As classes que herdarão são: Pix, Dinheiro, CartaoCredito e CartaoDebito. Todas as classes que herdão Pagamento, possuem o “Override” de um método chamado “retornaTotal”. Esse método influencia no valor final que o Cliente/Usuário deverá pagar pelo pedido.

**Class Pix:**

* + Classe concreta que herda a classe Pagamento e possuí um atributo “chave” do tipo String.

**Class Dinheiro:**

* + Classe concreta que herda a classe Pagamento.

**Class CartaoCredito/CartaoDebito:**

* + Classes concretas que herdam a classe Pagamento e possuem um campo “numConta” do tipo String. O que se difere entre elas é o valor calculado pelo método “retornaTotal”.

**Class Pizza:**

* + Classe abstrata que possui um atributo “nome” do tipo String e outro “listaIngredientes” do tipo List<Ingrediente>.

**Class PizzaSalgada:**

* + Classe concreta que herda a classe Pizza e define o atributo estático listaPizzasSalgadas do tipo List<PizzaSalgada>. Esse atributo será responsável por armazenar as pizzas salgadas cadastradas no banco de dados, em tempo de execução.

**Class PizzaDoce:**

* + Classe concreta que herda a classe Pizza e define o atributo estático listaPizzasDoces do tipo List<PizzaDoce>. Esse atributo será responsável por armazenar as pizzas doces cadastradas no banco de dados, em tempo de execução.

**Class Ingrediente:**

* + Classe concreta que possui os seguintes atributos: String nome, String categoria e List<Ingrediente> listaIngrediente. O atributo listaIngrediente será responsável por armazenar os ingredientes cadastrados no banco de dados, em tempo de execução.

**Class Tamanho:**

* + Classe concreta que possui os seguintes atributos: String nome, int quantidadePecas, double valor e List<Tamanho> listaTamanhos. O atributo listaTamanhos será responsável por armazenar os tamanhos cadastrados no banco de dados, em tempo de execução.

**Class Produto:**

* + Classe concreta que possui os seguintes atributos: String nome, float valor, int quantidade e List<Produto> ListaProduto. Este último será responsável por armazenar os produtos cadastrados no banco de dados, em tempo de execução.

**Class BancoDeDados:**

* + Classe concreta responsável por criar e ler todos os arquivos que serão utilizados como banco de dados pelo sistema. Foi utilizada a biblioteca Gson para auxilio na leitura/escrita de arquivos “.json”.

Referente à **View**, teremos as seguintes interfaces gráficas:

**Ingredientes:**

* **Class AdicionaIngrediente:**
  + Responsável pela view “Cadastro de Ingrediente”.
* **Class GerenciamentoIngrediente:**
  + Responsável pela view “Gerenciamento de Ingredientes”.

**Pedidos:**

* **Class FazerPedido:**
  + Responsável pela view “Solicitar Pedido”.

**Pizzas:**

* **Class AdicionaPizzaDoce:**
  + Responsável pela view “Cadastro de Pizza Doce”.
* **Class AdicionaPizzaSalgada:**
  + Responsável pela view “Cadastro de Pizza Salgada”.
* **Class GerenciamentoPizza:**
  + Responsável pela view “Gerenciamento de Pizzas”.

**Produtos:**

* **Class AdicionaProduto:**
  + Responsável pela view “Cadastro de Produto”.
* **Class GerenciamentoProduto:**
  + Responsável pela view “Gerenciamento de Produtos”.

**Tamanhos:**

* **Class AdicionaTamanho:**
  + Responsável pela view “Cadastro de Tamanho”.
* **Class GerenciamentoTamanho:**
  + Responsável pela view “Gerenciamento de Tamanhos”.

**Class Cadastro:**

* + Responsável pela view “Tela Inicial”.

**Class Login:**

* + Responsável pela view “Tela de Login”.

**Class DashboardCliente:**

* + Responsável pela view “Dashboard” do Cliente.

**Class DashboardAdm:**

* + Responsável pela view “Dashboard” do Administrador.

Referente à **Controller**, teremos os seguintes:

* **AdministradorController:**
  + Responsável por validar o login como Administrador.
* **ClienteController:**
  + Responsável por validar o login como Cliente e o cadastro do novo usuário.
* **IngredienteController:**
  + Responsável por adicionar, editar e deletar um ingrediente do banco de dados.
* **PagamentoController:**
  + Responsável por adicionar, editar e deletar uma forma de pagamento do banco de dados.
* **PedidoController:**
  + Responsável por calcular o valor do pedido e alterar os status do pedido.
* **PizzaController:**
  + Responsável por adicionar, editar e deletar uma pizza do banco de dados.
* **ProdutoController:**
  + Responsável por adicionar, editar e deletar um produto do banco de dados.
* **TamanhoController:**
  + Responsável por adicionar, editar e deletar um tamanho do banco de dados.

**5 - Instrução Para Compilação e Execução do Código**

São pré-requisitos ter o Maven e o JDK instalado e configurado, após isso,

descompacte o arquivo.

Após descompactar, chegue até a pasta na qual descompactou pelo CMD e digite o comando "mvn clean", dê enter, e depois dê o comando "mvn install".

Ao fazer isso, esse comando criará uma pasta de nome "target" no arquivo, da qual permitirá a execução do programa.

Dentro da pasta target, haverá um arquivo executável, geralmente com o nome do projeto e de final SNAPSHOT-jar-with-dependencies.jar.

Sendo assim, para executar o programa é simples, volte ao CMD e escreva o

comando "java -jar target/NomeDoExecutavel.jar”.

Por exemplo: java -jar target/Nome-1.0-SNAPSHOT-jar-with-dependencies.jar

**6 - Usuários de Acesso**

* **Administrador:**
  + **Email:** [administrador@adm.com.br](mailto:administrador@adm.com.br)
  + **Senha:** 123456789
* **Usuário:**
  + **Email:** usuario@user.com
  + **Senha:** 123456