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

RELATÓRIO - PROGRAMAÇÃO 2 (MyFood)
Ciência da Computação

Matheus Lopes dos Santos

MACEIÓ - AL
2024

Introdução

O projeto desenvolvido é um sistema de gerenciamento para uma plataforma de pedidos e restaurantes, utilizando Programação Orientada a Objetos (POO) com a linguagem Java. O sistema adota o padrão de projeto Facade para simplificar a interface com o usuário e implementa persistência de dados para garantir que as informações sejam salvas e recuperadas de forma eficiente. Além disso, o código está

1. Programação Orientada a Objetos (POO)

A Programação Orientada a Objetos é um paradigma fundamental para o desenvolvimento do sistema, permitindo a modelagem dos dados e comportamentos de forma organizada e modular. O sistema utiliza conceitos de POO como encapsulamento, herança e polimorfismo:

- **Encapsulamento**: Os dados são encapsulados dentro de classes e objetos. Por exemplo, a classe Usuario encapsula os atributos e comportamentos comuns a todos os usuários, como id, nome, email, senha e endereco. Métodos como getAtributo e podeCriarEmpresa garantem o acesso controlado a essas informações.
- **Herança**: O sistema faz uso de herança para especializar tipos de usuários e produtos. As classes Cliente e DonoRestaurante estendem a classe Usuario, herdando seus atributos e métodos, mas adicionando comportamentos específicos. Da mesma forma, o padrão de projeto Facade é implementado por meio da classe Facade, que fornece uma interface simplificada para interagir com o sistema.
- **Polimorfismo**: O método getAtributo na classe Usuario é um exemplo de polimorfismo, pois pode retornar diferentes tipos de atributos com base na string fornecida. As classes que estendem Usuario implementam o método podeCriarEmpresa de maneira específica, demonstrando o polimorfismo em ação.

2. Uso da Linguagem Java

Java foi escolhido como a linguagem principal para o desenvolvimento do sistema devido a suas características robustas e amplamente aceitas no desenvolvimento de software:

- **Classes e Objetos**: O código foi estruturado em várias classes que representam diferentes entidades do sistema, como Usuario, Restaurante, Produto, e Pedido. Cada classe é responsável por encapsular dados e comportamentos específicos.
- **Exceções**: O sistema usa classes de exceções personalizadas, como AtributoInvalidoException, para lidar com condições de erro e garantir a

robustez do código. Isso permite que o sistema informe ao usuário sobre erros específicos de maneira controlada.
- **Serialização**: A implementação da interface Serializable em classes como Usuario permite que objetos sejam convertidos em um formato que pode ser salvo em disco e recuperado mais tarde, facilitando a persistência de dados.

3. Comentários e Documentação

O código do projeto é enriquecido com comentários detalhados que desempenham um papel crucial na sua manutenção e compreensão. Estes comentários estão presentes em diversas partes do código, incluindo:

- **Descrição de Métodos e Funções**: Cada método e função inclui comentários que explicam seu propósito, parâmetros e o valor retornado. Isso facilita a compreensão do que cada parte do código faz e como ela deve ser utilizada.
- **Exceções e Tratamento de Erros**: Comentários adicionais explicam as exceções lançadas e as condições que podem levar a erros, o que ajuda a identificar e resolver problemas de forma mais eficiente.
- **Explicações de Lógica e Algoritmos**: A lógica por trás de decisões importantes e algoritmos complexos é explicada com comentários que clarificam o raciocínio e a abordagem utilizada. Isso é especialmente útil para novos desenvolvedores que possam trabalhar no projeto no futuro.
- **Seções e Blocos de Código**: Comentários também são usados para dividir o código em seções distintas e descrever o que cada bloco faz. Isso melhora a legibilidade e a organização do código.

Esses comentários não apenas tornam o código mais acessível e compreensível, mas também garantem que a equipe de desenvolvimento possa realizar modificações e melhorias com maior eficiência e menor risco de introduzir erros.

4. Design

A arquitetura de software refere-se à estrutura e organização de um sistema de software, que inclui como diferentes componentes e camadas interagem entre si. Dentro desse projeto, utilizamos: Models, Services e Exceptions.

4.1 Exceptions

As Exceptions (ou exceções) são classes que representam erros ou situações anormais que podem ocorrer durante a execução de um programa. No nosso código, existem várias classes de exceção personalizadas que estendem a classe Exception para tratar casos específicos. Isso permite gerenciar e tratar erros de forma controlada, facilitando a identificação e solução de problemas. A seguir, a descrição das exceptions usadas no projeto:

AtributoInvalidoException: lançada quando um atributo fornecido é considerado inválido.
AtributoNaoExisteException: lançada quando um atributo solicitado não existe.
CategoriaInvalidaException: lançada quando a categoria fornecida é inválida.
CpfInvalidoException: lançada quando o CPF fornecido é inválido.
DonoNaoPodePedidoException: lançada quando um dono de empresa tenta fazer um pedido, o que não é permitido.
EmailExistenteException: lançada quando uma tentativa de criar uma conta com um email já existente é feita.
EmailInvalidoException: lançada quando um email fornecido não é considerado válido.
EmpresaNaoCadastradaException: lançada quando uma operação é realizada em uma empresa que não está cadastrada.
EmpresaNaoEncontradaException: lançada quando uma empresa não pode ser encontrada.
EnderecoDuplicadoException: lançada quando há uma tentativa de cadastrar duas empresas com o mesmo nome e local.
EnderecoEmpresaInvalidoException: lançada quando o endereço da empresa é considerado inválido.
EnderecoInvalidoException: lançada quando o endereço fornecido é considerado inválido.
EntregadorEmEntregaException: lançada quando um entregador tenta realizar uma ação enquanto ainda está em uma entrega.
EntregadorNaoValidoException: lançada quando uma ação é solicitada para um entregador que não é considerado válido.
EntregadorSemEmpresaException: lançada quando um entregador não está associado a nenhuma empresa.
FormatoHoraInvalidoException: lançada quando o formato da hora fornecido é inválido.
HorarioInvalidoException: lançada quando o horário fornecido é inválido.
IndiceInvalidoException: lançada quando o índice fornecido é inválido.
IndiceMaiorException: lançada quando o índice é maior do que o esperado.
LoginSenhaInvalidosException: lançada quando o login ou a senha são inválidos.
MercadoInvalidoException: lançada quando não é um mercado válido.
NaoEhPossivelLiberarException: lançada quando não é possível liberar um produto que não está sendo preparado.
NaoExistePedidoAbertoException: lançada quando não existe um pedido em aberto.
NaoExistePedidoEntregaException: lançada quando não existe um pedido para entrega.

E alguns outros que seguem a mesma nomenclatura e estão descritos no código.

4.2 Services

Os Services (ou serviços) são classes que encapsulam a lógica de negócios do aplicativo. Elas costumam fornecer métodos que realizam operações relacionadas a funcionalidades específicas do sistema, como criação, leitura, atualização e exclusão (CRUD) de dados. Os serviços ajudam a manter o código organizado e a separar a lógica de negócios da interface do usuário e do acesso a dados.

Os serviços são responsáveis pela persistência de dados no formato de arquivos, permitindo salvar e carregar diversas entidades do sistema, como empresas, entregas, pedidos, produtos e usuários. Cada classe é projetada para gerenciar um tipo específico de entidade, utilizando um Map para armazenar dados e um fluxo de entrada/saída (ObjectOutputStream e ObjectInputStream) para a serialização e deserialização dos objetos. Isso garante que as informações possam ser armazenadas de maneira durável e recuperadas quando necessário, facilitando o gerenciamento e a integridade dos dados ao longo do ciclo de vida do aplicativo.

4.3 Models

Os **Models** (ou modelos) representam as entidades do domínio da aplicação. Eles são usados para armazenar dados e comportamentos relacionados a essas entidades. No projeto, classes como Restaurante, Mercado, Farmacia, Entregador e Entrega são exemplos de modelos. Cada uma dessas classes contém atributos que representam propriedades da entidade e métodos que definem seu comportamento. Os modelos ajudam a organizar e estruturar os dados da aplicação, facilitando o gerenciamento e a manipulação desses dados.

A classe **Cliente** representa um usuário do tipo cliente no sistema MyFood. Ela herda as propriedades e comportamentos da classe Usuario, mas é restrita em suas funcionalidades, não podendo criar empresas. O construtor inicializa o cliente com nome, email, senha e endereço, enquanto o método podeCriarEmpresa retorna falso, indicando que um cliente não tem permissão para essa ação. A classe também implementa o método ehEntregador, que retorna falso, indicando que um cliente não é um entregador.

A classe **DonoRestaurante** representa um usuário do tipo dono de restaurante no sistema MyFood. Ela herda as propriedades e comportamentos da classe Usuario e adiciona um atributo específico: o CPF do dono. O construtor inicializa o dono do restaurante com nome, email, senha, endereço e CPF. A classe inclui um método getter para acessar o CPF e um método getAtributo, que permite retornar o CPF ou outros atributos da superclasse, levantando uma exceção se o atributo solicitado for inválido. O método podeCriarEmpresa retorna verdadeiro, permitindo que donos de restaurantes criem empresas, enquanto o método ehEntregador retorna falso, indicando que um dono de restaurante não é um entregador.

A classe **Empresa** é uma classe abstrata que define a estrutura básica de uma empresa no sistema MyFood e implementa a interface Serializable para permitir a persistência de dados. Ela possui atributos como ID único, tipo de empresa, nome, endereço e uma lista de entregadores associados. O construtor inicializa esses atributos e atribui um ID único a cada instância, utilizando um contador estático. Métodos getters e setters são fornecidos para acessar e modificar os atributos, incluindo uma inicialização tardia da lista de entregadores. A classe também contém métodos abstratos que devem ser implementados pelas subclasses, permitindo que definam se a empresa é um mercado ou farmácia, além de métodos para manipular atributos de forma flexível. O método getAtributo permite a recuperação de valores de atributos baseados em strings, levantando uma exceção personalizada se o atributo solicitado for inválido.

A classe **Entrega** representa uma entrega no sistema MyFood e implementa a interface Serializable para permitir a persistência de dados. Seus atributos incluem um ID único para a entrega, o ID do pedido associado, o ID do entregador responsável e o endereço de destino da entrega. O construtor inicializa esses atributos com os valores fornecidos. A classe possui métodos getters para encapsular os atributos, permitindo que sejam acessados de forma controlada. O método getAtributo permite obter o valor de um atributo específico, utilizando informações persistidas de pedidos e usuários, carregadas através das classes de serviço PedidoSave e UsuarioSave. O método trata vários casos, retornando informações como ID do pedido, nome do entregador, nome do cliente, nome da empresa relacionada ao pedido e o destino da entrega. Caso o atributo solicitado seja inválido ou não exista, exceções personalizadas são lançadas, garantindo um tratamento adequado de erros.

A classe **Entregador** representa um entregador no sistema MyFood e herda da classe Usuario, aproveitando a herança para reutilizar atributos e comportamentos comuns. Seus atributos específicos incluem o tipo de veículo utilizado pelo entregador (como moto, carro ou bicicleta) e a placa desse veículo. O construtor inicializa tanto os atributos herdados da classe Usuario quanto os atributos específicos da classe Entregador. A classe possui métodos getters e setters para o encapsulamento dos atributos, permitindo o acesso e a modificação controlada das informações do entregador. O método getAtributo é sobrescrito para permitir a recuperação de atributos específicos do entregador, como a placa e o tipo de veículo, além de delegar a busca para o método da classe pai quando necessário. A classe também implementa uma lógica de negócio que determina que um entregador não pode criar empresas, retornando false no método podeCriarEmpresa. Por fim, o método ehEntregador sempre retorna true, confirmando que o objeto representa um entregador.

A classe **Farmacia** representa uma farmácia no sistema MyFood, herdando da classe Empresa. Ela possui atributos específicos como se funciona 24 horas e o

número de funcionários. O construtor inicializa esses atributos, enquanto métodos sobrescritos permitem verificar o tipo da empresa (retornando false para mercado e true para farmácia) e acessar informações específicas. O método getAtributo é adaptado para retornar atributos específicos da farmácia, delegando a chamada para a classe pai quando necessário.

A classe **Mercado** representa um mercado no sistema MyFood, herdando da classe Empresa. Ela possui atributos específicos, como horários de abertura e fechamento, e o tipo de mercado (supermercado, minimercado ou atacadista). O construtor inicializa esses atributos, e métodos sobrescritos identificam o tipo da empresa (retornando true para mercado e false para farmácia). Métodos getAtributo e setAtributo permitem acessar e modificar os horários de funcionamento e o tipo de mercado, delegando chamadas à classe pai quando necessário.

A classe **Pedido** representa um pedido feito por um cliente a uma empresa e implementa a interface **Serializable** para permitir a persistência dos dados. Os atributos incluem um número único do pedido, o nome do cliente, o nome da empresa, o estado atual do pedido (como "aberto" ou "preparando"), uma lista de produtos e o valor total do pedido. O construtor inicializa o pedido com o cliente e a empresa, atribuindo um número único e configurando o estado como "aberto". Métodos como **adicionarProduto(Produto produto)** permitem adicionar produtos ao pedido e atualizar o valor total. O método **finalizarPedido()** muda o estado para "preparando", enquanto **removerProdutoPorNome(String nomeProduto)** remove produtos com base no nome, ajustando o valor total conforme necessário. Os getters fornecem acesso aos atributos do pedido, permitindo obter informações sobre o número, cliente, empresa, estado, produtos e valor total.

A classe **Produto** representa um item oferecido por uma empresa, implementando a interface Serializable para permitir a persistência de dados. Possui atributos como ID único, nome, valor e categoria, com um construtor que inicializa esses parâmetros e gera automaticamente um ID. A classe oferece métodos getters para acessar os atributos e um método getAtributo(String atributo) que retorna valores específicos com tratamento de exceções para atributos inexistentes, além de métodos setters para modificar o nome, valor e categoria do produto.

A classe **Restaurante** representa um tipo específico de empresa, herdando da classe Empresa e incorporando o conceito de herança da Programação Orientada a Objetos (POO). Ela possui um atributo específico, tipoCozinha, que define o estilo culinário do restaurante. O construtor inicializa os atributos herdados, além do tipo de cozinha. A classe oferece um método getter para acessar o tipo de cozinha e métodos sobrescritos que indicam que um restaurante não é um mercado nem uma farmácia. Além disso, possui um método getAtributo(String atributo) que permite

acessar atributos específicos do restaurante, delegando a responsabilidade para a classe pai caso o atributo solicitado não seja reconhecido.

A classe **Usuario** é uma classe abstrata que representa um usuário genérico do sistema, servindo como base para outras classes. Ela implementa a interface Serializable para permitir a serialização. A classe possui atributos como id, nome, email, senha e endereco, inicializados por meio de um construtor. Métodos getters são fornecidos para acessar esses atributos, e um método getAtributo(String atributo) permite obter o valor de um atributo específico, lançando uma exceção caso o atributo não exista. A classe também define dois métodos abstratos: podeCriarEmpresa(), que deve ser implementado para indicar se o usuário pode criar uma empresa, e ehEntregador(), que deve ser implementado para verificar se o usuário é um entregador.

5. Padrão de Projeto Facade

O padrão de projeto Facade é utilizado para fornecer uma interface simplificada para o sistema complexo subjacente:

- **Classe Facade**: A classe Facade atua como uma fachada que simplifica a interação com o sistema, oferecendo métodos para criar usuários, gerenciar produtos e pedidos, e consultar informações. Isso oculta a complexidade do sistema subjacente e fornece uma interface clara e intuitiva para os usuários e desenvolvedores.
- **Integração com o Sistema**: A Facade delega as solicitações para a classe Sistema, que implementa a lógica de negócio detalhada. Essa separação entre a fachada e o sistema subjacente facilita a manutenção e a escalabilidade do sistema.

6. Persistência de Dados

O sistema utiliza a persistência de dados para garantir que as informações sejam armazenadas e recuperadas de forma eficaz:

- **Serialização de Objetos**: As classes Sistema e UsuarioSave, entre outras, implementam métodos para salvar e carregar dados de objetos em arquivos. Isso permite que o estado do sistema seja preservado entre execuções.
- **Métodos de Salvamento e Carregamento**: Métodos como UsuarioSave.carregarUsuarios() e ProdutoSave.salvarProdutos() são responsáveis por carregar e salvar dados em arquivos, garantindo que as informações persistam e possam ser recuperadas quando o sistema for reiniciado.

Conclusão

O projeto demonstra a aplicação eficaz de Programação Orientada a Objetos, a utilização da linguagem Java, o padrão de projeto Facade e técnicas de persistência de dados. A estrutura modular e a organização do código facilitam a manutenção e a escalabilidade, enquanto a persistência de dados assegura a continuidade do estado do sistema.

Este projeto não só exemplifica boas práticas de design de software, mas também fornece uma base sólida para futuras expansões e melhorias.