Instituto Nacional de Telecomunicações - INATEL

Sistema de Controle de Cardápio

Aplicação 2

Vitória De Moraes Dutra - GES - 414

Lucca Marcondes Madeira - GES - 420

Arquitetura e Desenho de Software - S203

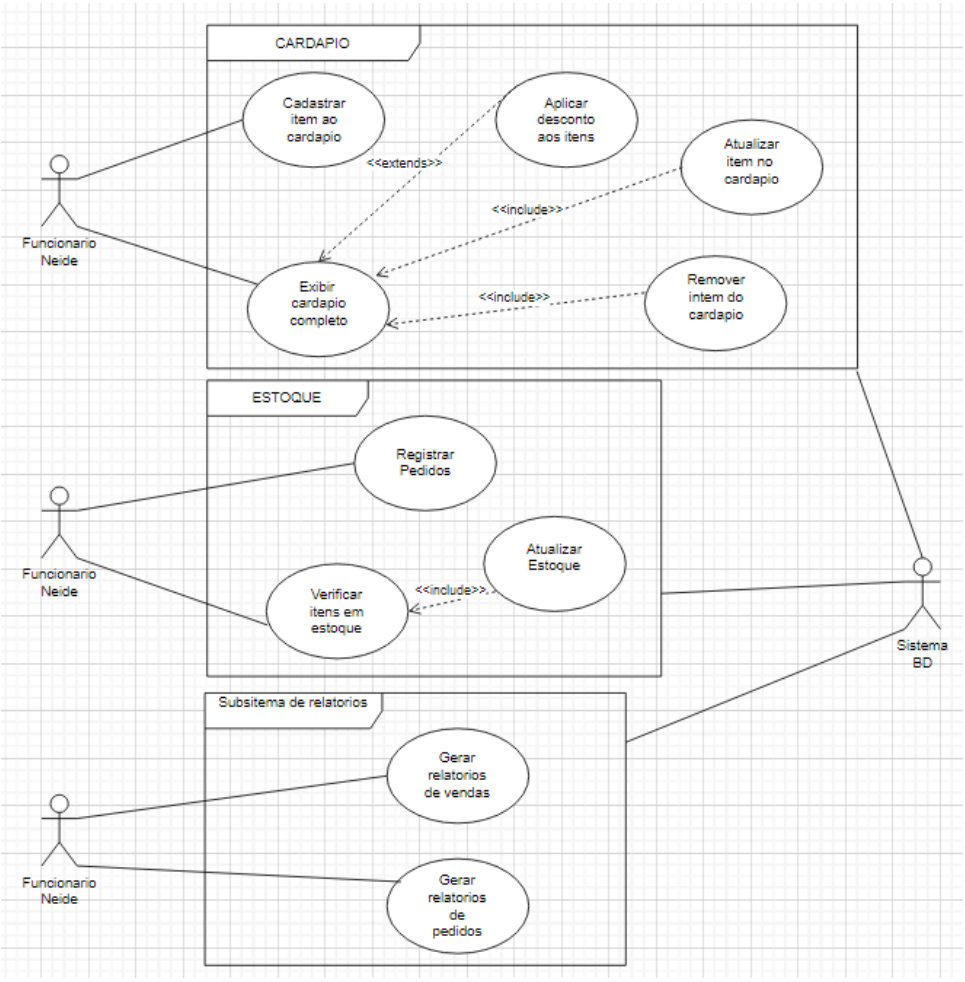
Documentação do Projeto Neides Project

Introdução ao Projeto

Este documento apresenta a documentação do projeto Neides Project, um sistema para gerenciamento de cantina utilizando Django, seguindo o padrão arquitetural MVC (Model-View-Controller). O sistema visa permitir o cadastro, atualização, remoção e exibição de itens à venda, além do controle de estoque e geração de relatórios de vendas.

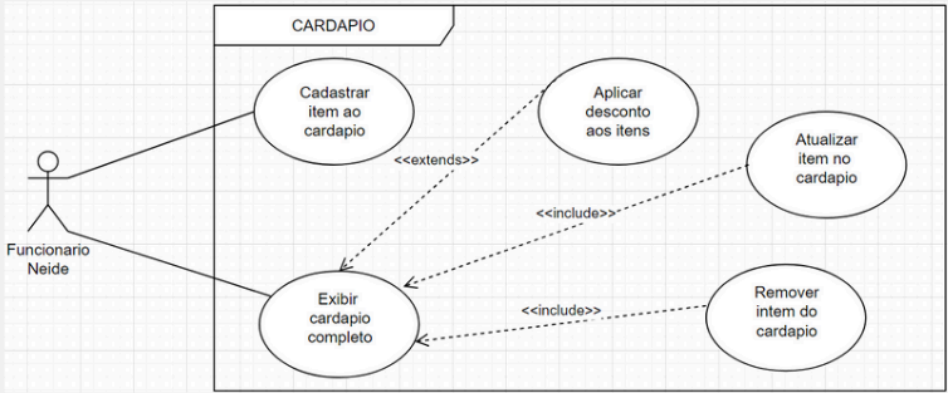
Diagrama de Casos de Uso

O diagrama de casos de uso apresenta as funcionalidades principais do sistema, e as interações entre os atores e o sistema, seguindo o padrão MVC. Neste diagrama, é possível visualizar os casos de uso, em que descreve as principais atividades realizadas pelo ator principal, o funcionário da Neide, e o sistema de banco de dados.



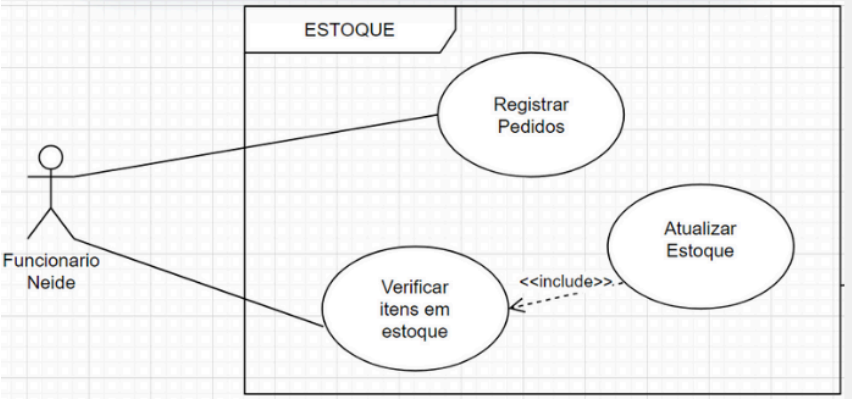
Primeira Fronteira

Esta fronteira engloba todas as operações relacionadas ao gerenciamento do cardápio, que é um dos principais objetivos do sistema. O funcionário da cantina pode realizar o cadastro de novos itens, fornecendo informações como nome, categoria, preço e quantidade disponível. Além disso, há a possibilidade de atualizar os dados dos itens já existentes, como modificar preços em função de mudanças no mercado ou até mesmo visando um preço justo. A exibição do cardápio completo, que considera apenas os itens que estão disponíveis em estoque, garantindo que o cliente visualize apenas opções que podem ser efetivamente vendidas. Todas essas operações são integradas ao banco de dados, que armazena as informações de forma centralizada e consistente.



Segunda Fronteira

Concentra-se no controle de estoque. Essa fronteira lida com a verificação dos itens disponíveis em estoque, exibindo uma lista atualizada de produtos e suas respectivas quantidades. Além disso, a funcionalidade de atualização do estoque permite que o funcionário faça ajustes manuais sempre que necessário, como em casos de reposição de mercadorias ou correção de inconsistências identificadas durante a verificação de itens em estoque. Assim, garantindo que a quantidade de itens seja monitorada e ajustada de maneira eficiente.



Terceira Fronteira

Esta fronteira está focada na geração de relatórios que auxiliam no controle financeiro e de pedidos feitos na cantina. Os relatórios de vendas oferecem uma visão geral dos itens vendidos em um determinado período, incluindo informações como o total arrecadado e a quantidade de produtos comercializados. Já os relatórios de pedidos, por sua vez, detalham as transações realizadas, permitindo que o funcionário acompanhe os registros de forma organizada e identifique possíveis inconsistências.

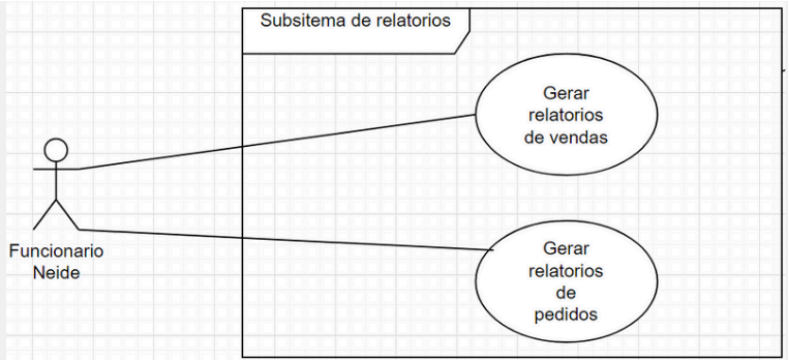
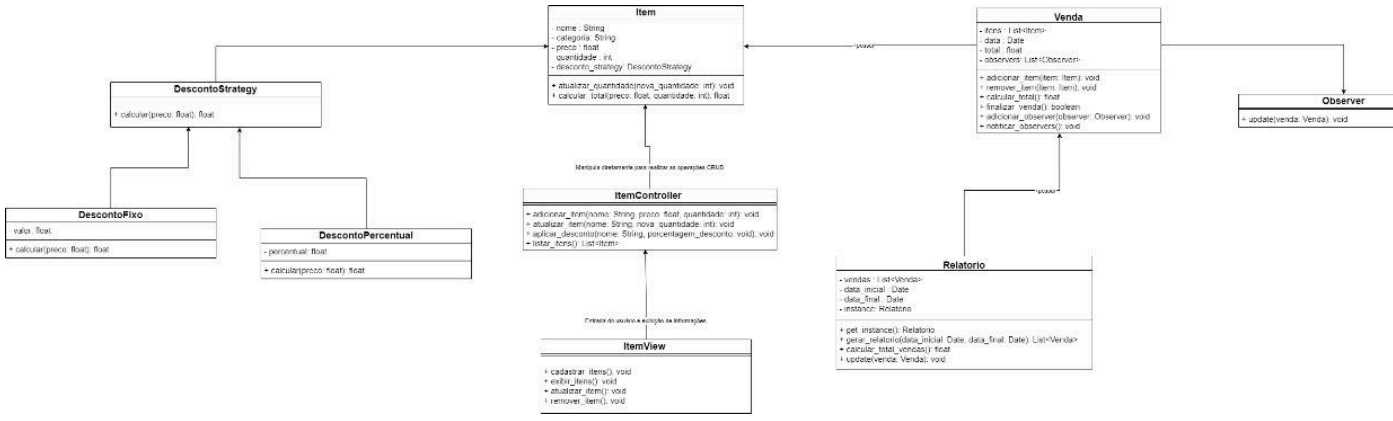


Diagrama de Classes

O diagrama de classes apresenta as classes utilizadas no sistema, suas propriedades, métodos e relacionamentos. Este diagrama segue os princípios SOLID e a estrutura MVC. Foram aplicados os design patterns **Strategy**, para gerenciar diferentes tipos de cálculo de desconto, e **Observer**, para notificar alterações nas vendas em tempo real. Dessa forma, proporcionam flexibilidade, escalabilidade e facilidade de manutenção, permitindo a expansão do sistema sem comprometer sua estrutura.



Design Patterns:

- **Observer:** Define uma relação de dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

- Este padrão será aplicado ao acompanhar mudanças em Venda. Quando uma venda é finalizada, os relatórios precisam ser atualizados automaticamente.

No projeto:

- Novas classes/interfaces:
  - Interface Observer
    - Métodos:
      - update(venda: Venda): void
  - Classe Venda
    - Adicionado um atributo privado: observers: List<Observer>.
    - Métodos:
      - adicionar_observer(observer: Observer): void
      - notificar_observers(): void
      - Modificação em finalizar_venda(): Chamar notificar_observers() para informar os observadores sobre a nova venda.
  - Classe Relatorio
    - Implementação da interface Observer.
      - Métodos:
        - update(venda: Venda): void será usado para registrar as vendas no relatório.

- **Singleton:** O padrão garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a ela. É útil para representar objetos únicos, como um gerenciador de banco de dados ou um objeto de configuração.
  - Este será aplicado à classe Relatorio, garantindo que apenas uma instância dela exista durante a execução da aplicação.

No projeto:

- Modificações na classe:
  - Relatorio
    - Adicionado um atributo privado estático: instance: Relatorio.étodos:
    - Adicionado um método público estático: get_instance(): Relatorio, que retorna a única instância da classe.

- **Strategy:** Este padrão define uma família de algoritmos, encapsula cada um deles e os torna intercambiáveis. Permite que você selecione um algoritmo em tempo de execução.
    - O padrão **Strategy** será aplicado ao calcular o desconto para os itens, permitindo diferentes estratégias de cálculo, como descontos fixos ou percentuais.

No projeto:

- Novas classes/interfaces:
    - Interface DescontoStrategy
        - Métodos:
            - calcular(preco: float): float
    - Classe DescontoFixo
        - Atributo: valor: float:
        - Métodos:
            - calcular(preco: float): float — subtrai um valor fixo do preço.
    - Classe DescontoPercentual
        - Atributo: percentual: float
        - Métodos:
            - calcular(preco: float): float — aplica um percentual de desconto ao preço.
    - Classe Item
        - Implementação do atributo: desconto_strategy: DescontoStrategy.
            - Métodos:
                - calcular_total(): float — utilize o método calcular da estratégia de desconto associada.

Estrutura MVC e Princípios SOLID

O sistema foi desenvolvido seguindo o padrão MVC, que separa as responsabilidades em três camadas:

➔ **Model**: Responsável pela manipulação dos dados (Item, Venda, Relatorio).
➔ **View**: Responsável pela exibição das informações ao usuário (ItemView).
➔ **Controller:** Gerencia as interações entre a View e o Model (ItemController).

Princípios Aplicados no Código

Os princípios SOLID foram aplicados nos modelos e controladores para garantir um código mais robusto e fácil de manter.

- **SRP:** `ItemController` gerencia apenas a lógica de controle, separando-a da manipulação de dados.

```python
class ItemController:
    def __init__(self, item_model):
        self.item_model = item_model

    def add_item(self, nome, preco, quantidade):
        """Adiciona um novo item ao sistema"""
        if preco < 0 or quantidade < 0:
            raise ValueError("Preço e quantidade devem ser positivos")
        self.item_model.create(nome=nome, preco=preco, quantidade=quantidade)

    def update_stock(self, item_id, nova_quantidade):
        """Atualiza a quantidade em estoque"""
        if nova_quantidade < 0:
            raise ValueError("Quantidade deve ser positiva")
        self.item_model.update(item_id, quantidade=nova_quantidade)
```

- **OCP:** `Venda` é extensível para adição de funcionalidades.

```python
class Venda:
    def __init__(self, item, quantidade, valor_total):
        self.item = item
        self.quantidade = quantidade
        self.valor_total = valor_total

class Relatorio:
    def gerar(self):
        raise NotImplementedError("Este método deve ser implementado pelas subclasses")

class RelatorioMensal(Relatorio):
    def gerar(self):
        """Gera um relatório de vendas mensais"""
        # Lógica para relatório mensal
        return "Relatório Mensal Gerado"

class RelatorioDiario(Relatorio):
    def gerar(self):
        """Gera um relatório de vendas diárias"""
        # Lógica para relatório diário
        return "Relatório Diário Gerado"
```

- **LSP:** Estratégias de desconto respeitando a interface ***DiscountStrategy***.

```python
class DiscountStrategy:
    def apply_discount(self, total):
        raise NotImplementedError("Este método deve ser implementado")

class FixedDiscount(DiscountStrategy):
    def __init__(self, desconto_fixo):
        self.desconto_fixo = desconto_fixo

    def apply_discount(self, total):
        return total - self.desconto_fixo

class PercentageDiscount(DiscountStrategy):
    def __init__(self, percentual):
        self.percentual = percentual

    def apply_discount(self, total):
        return total * (1 - self.percentual / 100)

# Exemplo de uso
def calcular_total_com_desconto(total, strategy: DiscountStrategy):
    return strategy.apply_discount(total)
```

- **ISP:** Separação de interfaces para validação e geração de relatórios.

```python
# utils/Validator.py
class Validator:
    def validate(self, data):
        raise NotImplementedError("Este método deve ser implementado")

class ItemValidator(Validator):
    def validate(self, data):
        """Valida dados de itens"""
        if "nome" not in data or "preco" not in data:
            raise ValueError("Dados de item incompletos")

# utils/ReportGenerator.py
class ReportGenerator:
    def generate(self):
        raise NotImplementedError("Este método deve ser implementado")

class PDFReportGenerator(ReportGenerator):
    def generate(self):
        """Gera relatório em formato PDF"""
        return "Relatório PDF Gerado"
```

- **DIP:** `ItemController` depende de abstrações como `Item`.

```python
# controllers/ItemController.py
class ItemController:
    def __init__(self, item_model):
        self.item_model = item_model  # Abstração (interface)

    def add_item(self, nome, preco, quantidade):
        """Adiciona um novo item ao sistema"""
        self.item_model.create(nome=nome, preco=preco, quantidade=quantidade)

# models/ItemModel.py
class ItemModel:
    def create(self, nome, preco, quantidade):
        """Interface para criação de itens"""
        raise NotImplementedError("Este método deve ser implementado")

# Implementação concreta
class MySQLItemModel(ItemModel):
    def create(self, nome, preco, quantidade):
        """Cria um item no banco de dados MySQL"""
        # Lógica para salvar no banco
        print(f"Item {nome} adicionado ao banco MySQL")
```

Princípios da Arquitetura de Software no Projeto

Os princípios da arquitetura de software foram aplicados para garantir um sistema organizado, fácil de manter e escalável. A separação de responsabilidades através da arquitetura MVC facilita a organização do código e a evolução do sistema, enquanto os princípios SOLID foram aplicados nos modelos e controladores para garantir um código mais robusto e fácil de manter.

Coesão: Cada componente deve ter uma única responsabilidade e focar em uma tarefa bem definida. Um módulo coeso realiza apenas uma coisa, mas a faz bem.

Aplicação no projeto:

- "*ItemController*" se preocupa apenas com a lógica de controle de itens, enquanto "*ItemModel*" lida com a persistência no banco de dados.

Dessa maneira, facilitando a leitura e manutenção do código, evitando que alterações em uma funcionalidade afetem outras partes do sistema.

Encapsulamento: Os detalhes internos de um componente devem ser ocultados, expondo apenas as funcionalidades necessárias por meio de interfaces públicas.

Aplicação no projeto:

- ***Models:*** O acesso direto aos dados no banco é feito apenas pelos métodos dos modelos.

```python
class ItemModel:
    def __init__(self):
        self.__estoque = {}

    def adicionar_item(self, nome, preco, quantidade):
        self.__estoque[nome] = {"preco": preco, "quantidade": quantidade}

    def consultar_estoque(self):
        return self.__estoque
```

- ***Controllers:*** Usa métodos para interagir com os modelos, escondendo os detalhes de implementação.
    - ***ItemController*** interage com ***ItemModel*** sem expor diretamente os dados do banco.

Dessa forma, reduz o impacto de mudanças no código interno, pois os detalhes são abstraídos.

Baixo acoplamento: Componentes tem o menor número possível de dependências entre si, tornando o sistema mais flexível.

Aplicação no projeto:

- ***Camadas do MVC:***
    - Model não conhece a View diretamente. As alterações no Model são notificadas pelo Controller, que atualiza a View.
    - Controller atua como intermediário, garantindo baixo acoplamento entre Model e View.
- **Dependência de Abstrações:** Usa interfaces em vez de dependências concretas.

```python
class ItemController:
    def __init__(self, model):
        self.model = model  # Depende de uma abstração, não de uma classe específica
```

Assim, permitindo mudanças em uma parte específica do sistema sem quebrar outras partes, Isso ocorre ao depender de abstrações em vez de implementações concretas, garantindo flexibilidade e fácil manutenção.

Banco de Dados - MySQL

O **MySQL** é um sistema de gerenciamento de banco de dados relacional de código aberto, que utiliza SQL para gerenciar e manipular dados. Ele é amplamente utilizado por sua eficiência, escalabilidade e suporte a diversas aplicações, sendo ideal para armazenar e organizar informações em sistemas web e corporativos.

No projeto: O MySQL oferece vantagens significativas para o projeto, como a organização e o gerenciamento eficiente das informações. Ele permite o armazenamento estruturado de dados, garantindo integridade e consistência por meio de chaves primárias e relacionamentos. Além disso, possibilita consultas rápidas e flexíveis, essenciais para funcionalidades como gestão de estoque, vendas e relatórios, sendo uma solução confiável e escalável.

Estrutura das tabelas (MySQL):

- **Itens:**

```sql
CREATE TABLE Itens (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nome VARCHAR(255) NOT NULL,
    preco DECIMAL(10, 2) NOT NULL,
    quantidade_estoque INT NOT NULL,
    data_criacao DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

- **Vendas:**

```sql
CREATE TABLE Vendas (
    id INT AUTO_INCREMENT PRIMARY KEY,
    item_id INT NOT NULL,
    quantidade INT NOT NULL,
    valor_total DECIMAL(10, 2) NOT NULL,
    data_venda DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (item_id) REFERENCES Itens(id)
);
```

- **Relatórios:**

```sql
CREATE TABLE Relatorios (
    id INT AUTO_INCREMENT PRIMARY KEY,
    tipo ENUM('diario', 'mensal') NOT NULL,
    conteudo TEXT NOT NULL,
    data_criacao DATETIME DEFAULT CURRENT_TIMESTAMP
);
```

Conclusão: O projeto Controle de Cardápio da cantina Neide foi desenvolvido utilizando o padrão arquitetural MVC, e seguindo os princípios SOLID e design patterns como Strategy, Observer e Singleton, garantindo organização, flexibilidade e fácil manutenção do sistema. A integração com o banco de dados MySQL permite um gerenciamento eficiente das informações, essencial para funcionalidades como cadastro de itens, controle de itens em estoque e geração de relatórios. Essa abordagem assegura que o sistema atenda às necessidades da cantina de forma confiável, escalável e com potencial para futuras expansões.