Sistema de controle de cardápio - Neide’s

Aplicação dos Princípios SOLID e Arquitetura MVC

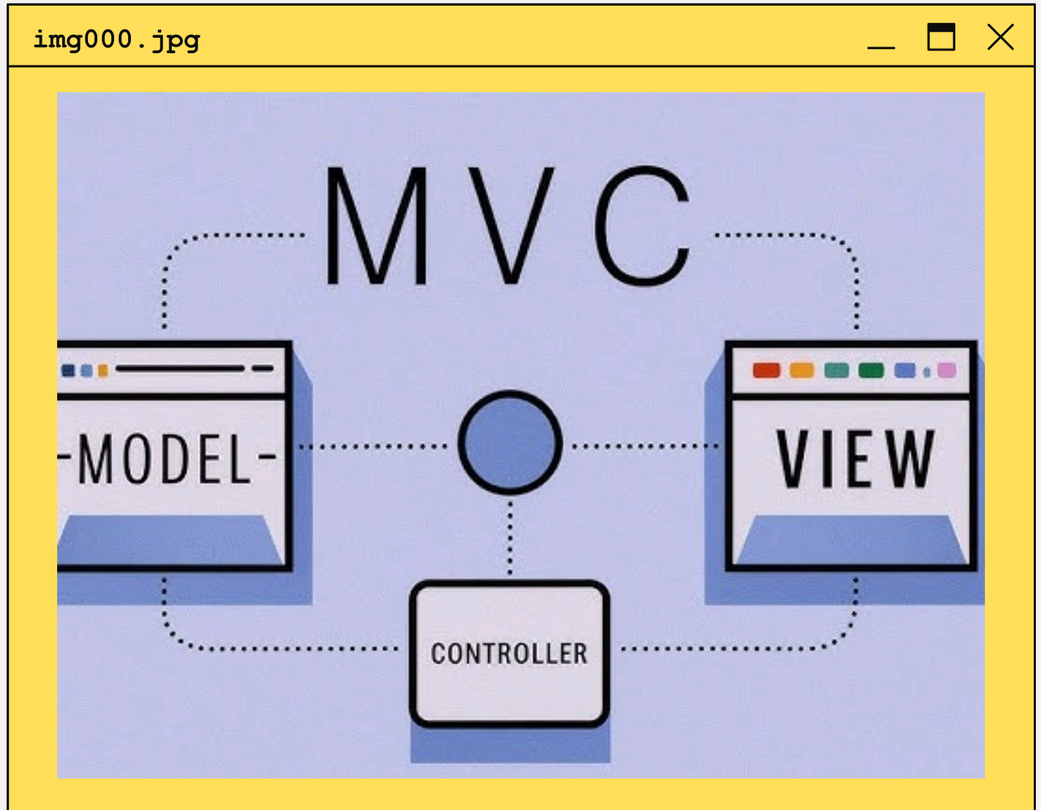
Vitória de Moraes Dutra - GES - 414
Lucca Marcondes - GES - 420

S203 - Arquitetura e Desenho de Software

Controle de Cardápio

O Neides Project foi desenvolvido para facilitar a gestão da cantinas, oferecendo recursos como cadastro de itens, controle de estoque, aplicação de descontos e geração de relatórios de vendas. Ele promove organização e flexibilidade com base nos princípios SOLID, arquitetura MVC e padrões de design modernos

Arquitetura MVC



MVC é um padrão de arquitetura de software que separa uma aplicação em três camadas: Modelo, Visão e Controlador

- **Modelo (Model):** Responsável pelos dados e lógica de negócios.
- **Visão (View):** Interface com o usuário.
- **Controlador (Controller):** Gerencia a comunicação entre modelo e visão.

Motivo da escolha: MVC separa responsabilidades, facilitando manutenção e escalabilidade.

Diagrama de Casos de Uso

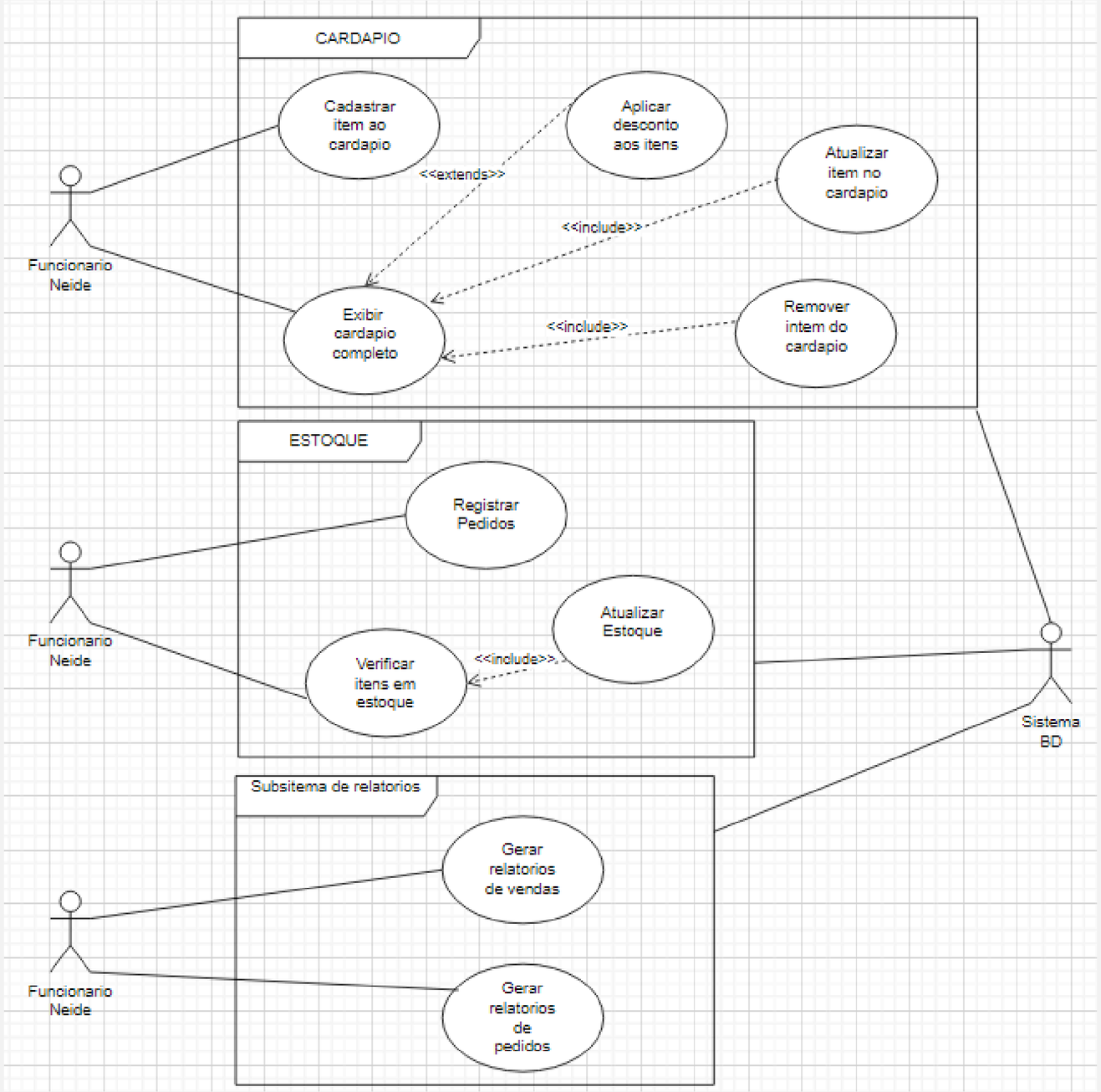
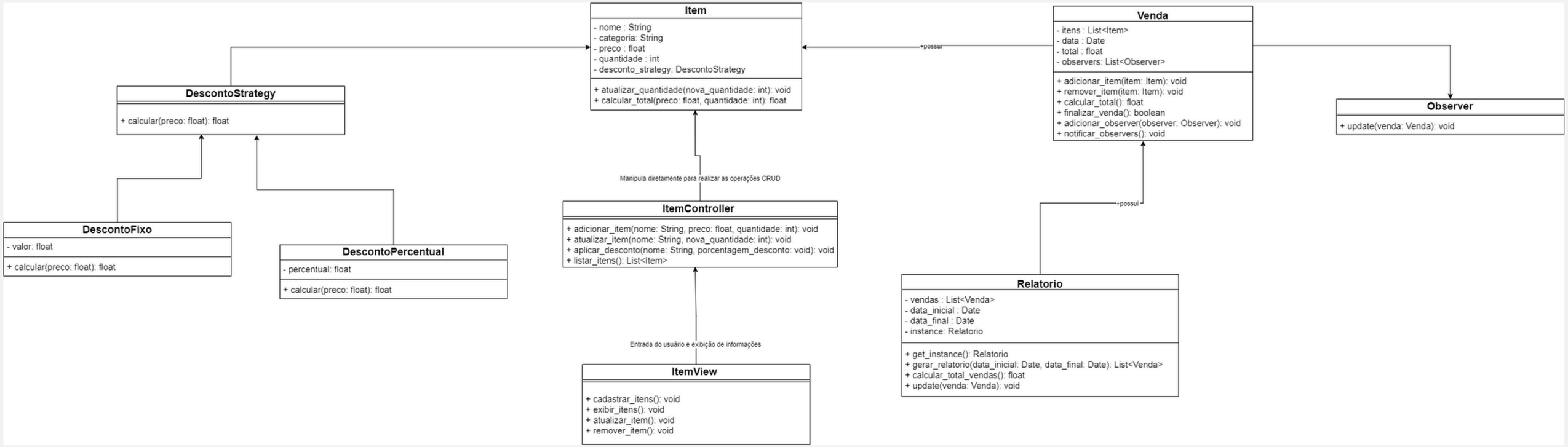


Diagrama de Classes



Design Patterns no Projeto

Observer

O padrão 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.

O padrão **Observer** 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 Singleton 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.

O padrão **Singleton** será aplicado à classe Relatorio, garantindo que apenas uma instância dela exista durante a execução da aplicação.

No projeto:

- <u>Modificações na classe:</u>
  - 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

O padrão Strategy 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.

Princípios SOLID no Projeto

Single Responsibility Principle (SRP)

Definição:

Uma classe deve ter apenas uma razão para mudar, ou seja, deve ser responsável por apenas uma tarefa. Isso torna o código mais organizado, fácil de manter e menos propenso a erros.

No projeto:

A classe ***ItemController*** é responsável apenas por gerenciar a lógica de controle

```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 (Open/Closed Principle)

Definição:

As entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação. Podem ser adicionadas novas funcionalidades a uma classe sem alterar o seu código original.

No projeto:

A classe ***Venda*** pode ser estendida para gerar novos tipos de relatórios sem alterar o código existente

```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 (Liskov Substitution Principle)

Definição:

Objetos de uma superclasse devem ser substituíveis por objetos de uma subclasse sem afetar a correção do programa. Em outras palavras, uma subclasse deve ser substituível por sua superclasse sem que o comportamento do programa seja alterado.

No projeto:

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 (Interface Segregation Principle)

Definição:

É um princípio de design de software que afirma que os clientes não devem ser forçados a depender de interfaces que não utilizam. Em vez de uma única interface grande, é melhor ter várias interfaces menores e mais específicas.

No projeto:

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 (Dependency Inversion Principle)

Definição:

É um princípio de design de software que afirma que os módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.

No projeto:

ItemController depende da interface do modelo, em vez de uma implementação específica.

```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

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.

Facilita 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
```

Encapsulamento

Aplicação no projeto:

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

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**.

Baixo Acoplamento

Aplicação no projeto:

- **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
```

Permite mudanças em uma parte do sistema sem quebrar as outras.

Utilização do Banco de Dados - MySQL

Estrutura das tabelas

Untitled - TextEdit

File Edit View Help

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
);
```

Estrutura das tabelas

Untitled - TextEdit

File Edit View Help

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)
);
```

Estrutura das tabelas

Untitled - TextEdit

File Edit View Help

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
);
```

Inserção de itens

```python
import mysql.connector

def inserir_item(nome, preco, quantidade):
    # Query SQL para inserir o item na tabela Itens
    query = "INSERT INTO Itens (nome, preco, quantidade_estoque) VALUES (%s, %s, %s)"
    valores = (nome, preco, quantidade)

    # Executa a query e confirma a transação
    cursor.execute(query, valores)
    conexao.commit()

    print(f"Item '{nome}' inserido com sucesso!")


    cursor.close()
    conexao.close()


inserir_item("Coxinha", 5.50, 100)
```

Registro de Vendas

```python
def registrar_venda(item_id, quantidade, valor_total):
    # Verifica se há estoque suficiente para realizar a venda
    cursor.execute("SELECT quantidade_estoque FROM Itens WHERE id = %s", (item_id,))
    estoque = cursor.fetchone()[0]  # Recupera a quantidade atual do estoque
    if estoque < quantidade:
        print("Estoque insuficiente!")
        return  # Interrompe a execução caso não haja estoque suficiente

    # Query para registrar a venda na tabela Vendas
    query_venda = "INSERT INTO Vendas (item_id, quantidade, valor_total) VALUES (%s, %s, %s)"
    valores_venda = (item_id, quantidade, valor_total)
    cursor.execute(query_venda, valores_venda)

    # Query para atualizar o estoque na tabela Itens
    query_estoque = "UPDATE Itens SET quantidade_estoque = quantidade_estoque - %s WHERE id = %s"
    valores_estoque = (quantidade, item_id)
    cursor.execute(query_estoque, valores_estoque)

    # Confirma as alterações no banco de dados
    conexao.commit()

    print("Venda registrada com sucesso!")

    # Fecha o cursor e a conexão
    cursor.close()
    conexao.close()

# Registra a venda de 2 unidades do item com ID 1, totalizando 11.00
registrar_venda(1, 2, 11.00)
```

Geração de Relatório

```python
def gerar_relatorio(tipo):
    # Define a query SQL com base no tipo de relatório solicitado
    if tipo == "diario":
        # Seleciona todas as vendas realizadas na data atual
        query = "SELECT * FROM Vendas WHERE DATE(data_venda) = CURDATE()"
    elif tipo == "mensal":
        # Seleciona todas as vendas realizadas no mês atual
        query = "SELECT * FROM Vendas WHERE MONTH(data_venda) = MONTH(CURDATE())"
    else:
        print("Tipo de relatório inválido!")
        return  # Interrompe a execução caso o tipo seja inválido

    # Executa a query e obtém os resultados das vendas
    cursor.execute(query)
    vendas = cursor.fetchall()

    # Monta o conteúdo do relatório a partir dos resultados obtidos
    conteudo = f"Relatório {tipo.capitalize()}:\n" + "\n".join([str(venda) for venda in vendas])

    # Query SQL para salvar o relatório na tabela Relatorios
    query_relatorio = "INSERT INTO Relatorios (tipo, conteudo) VALUES (%s, %s)"
    valores_relatorio = (tipo, conteudo)
    cursor.execute(query_relatorio, valores_relatorio)

    # Confirma as alterações no banco de dados
    conexao.commit()

    print(f"Relatório {tipo} gerado com sucesso!")


# Gera um relatório do tipo diário
gerar_relatorio("diario")
```