

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

Usada pelos funcionários da cantina e deve permitir o cadastro de itens a venda (definindo os respectivas custos, quantidades e preços), além de permitir a atualização dos valores e remoção de itens cadastrados. A aplicação deve controlar cardápio exibindo apenas os itens em estoque, mostrando uma lista de itens pedidos e subtraindo a quantidade dos itens vendidos. Por fim, é esperado que a aplicação possa gerar relatórios de vendas



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. **Exemplo: Item, Venda.**
- **Visão (View):** Interface com o usuário. **Exemplo: ItemView.**
- **Controlador (Controller):** Gerencia a comunicação entre modelo e visão. **Exemplo: ItemController.**

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

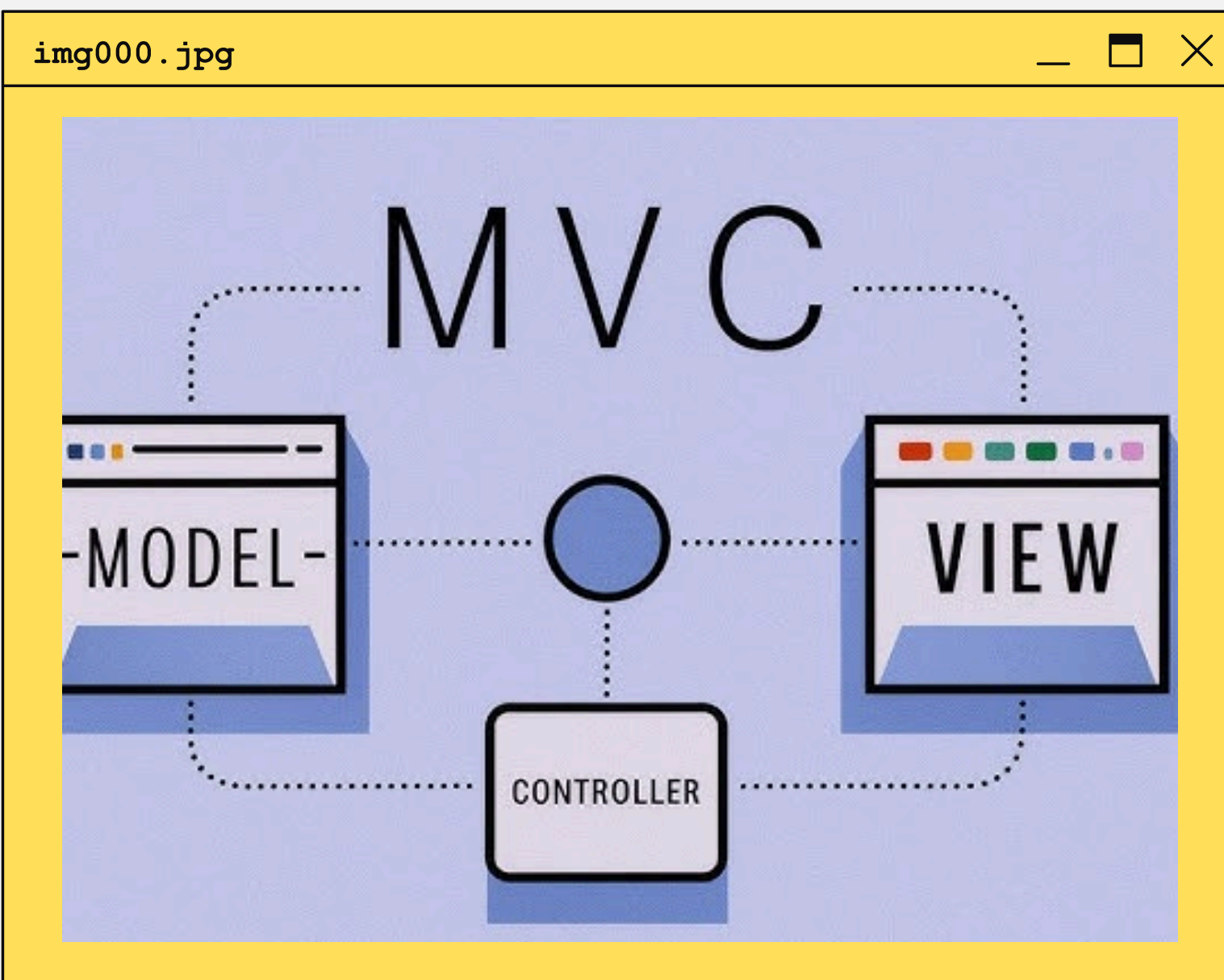
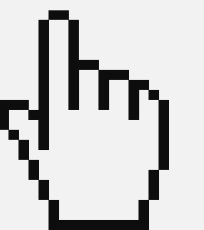
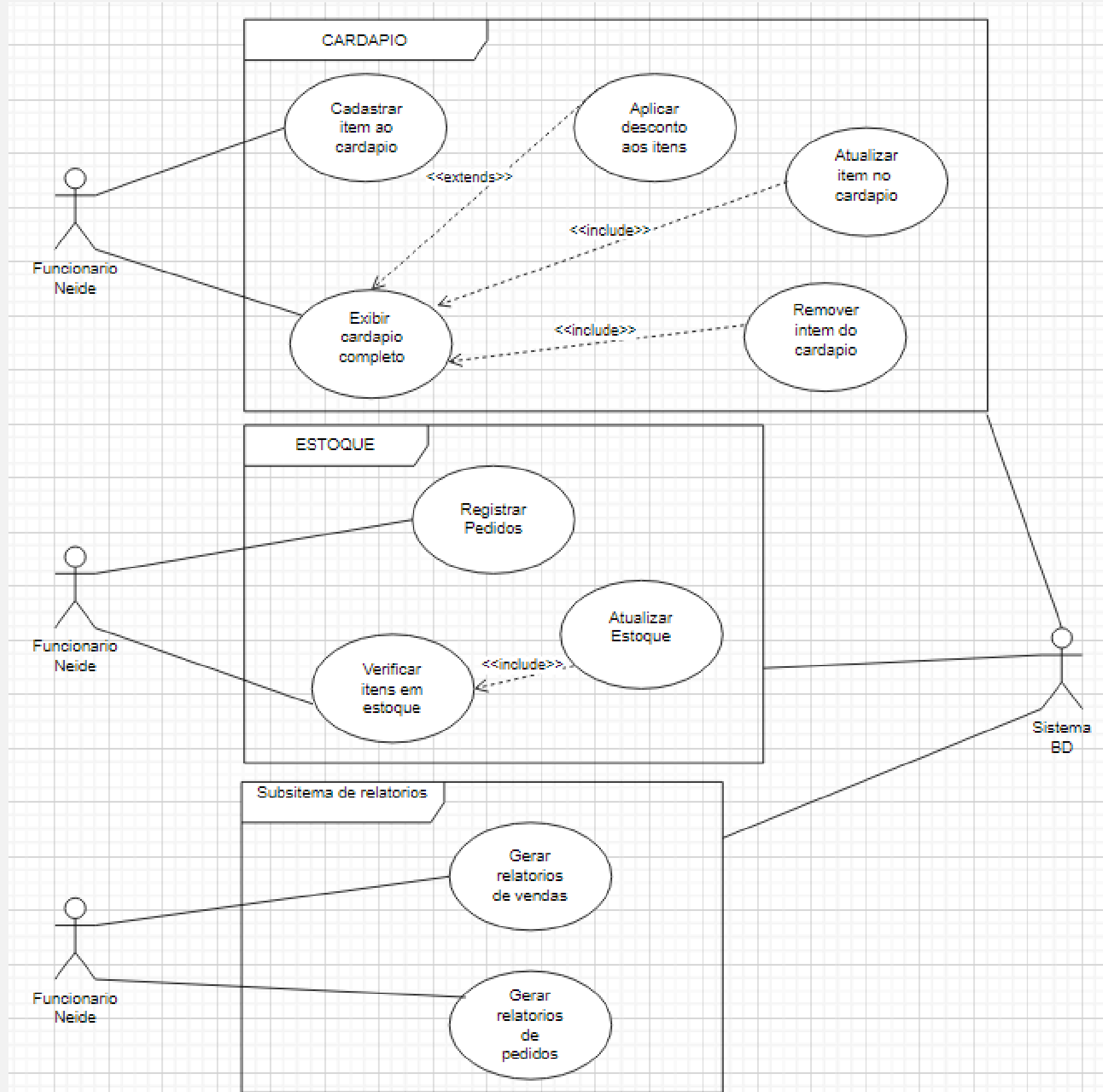
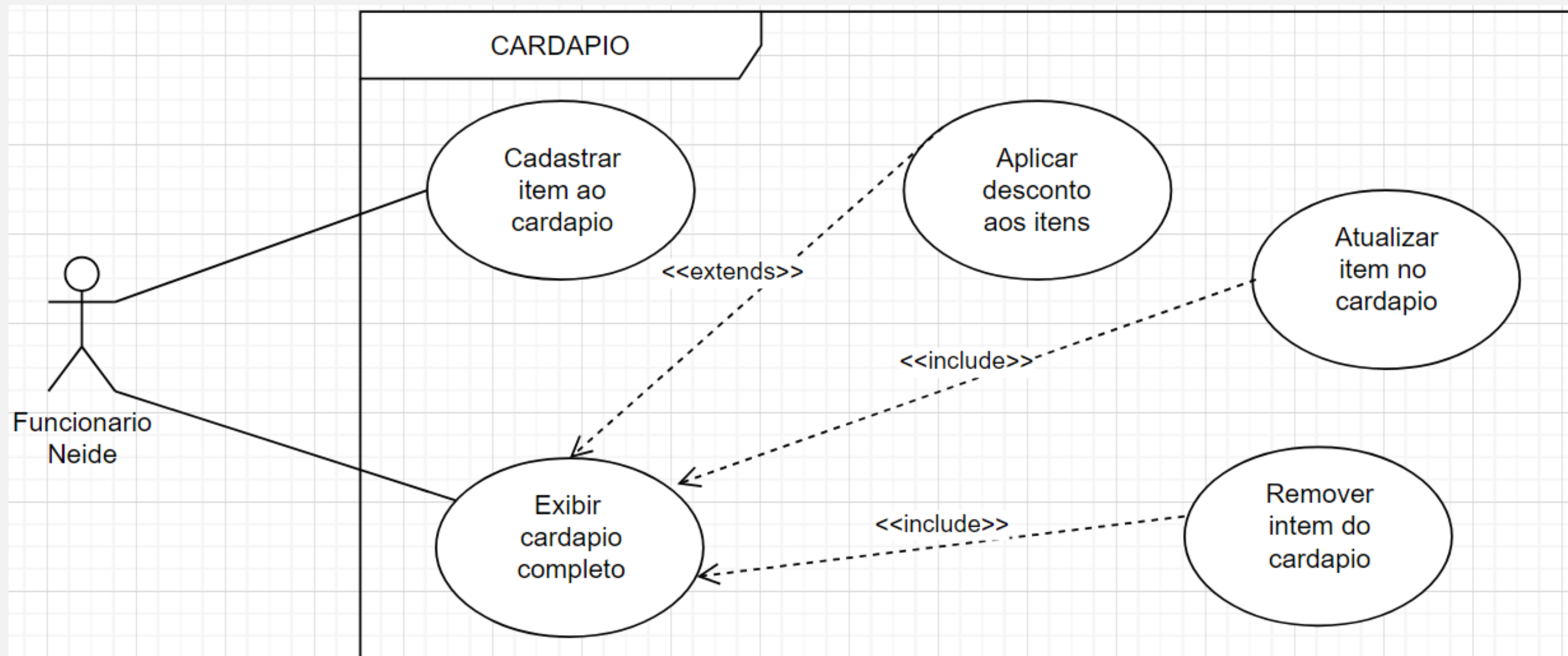


Diagrama de Casos de Uso

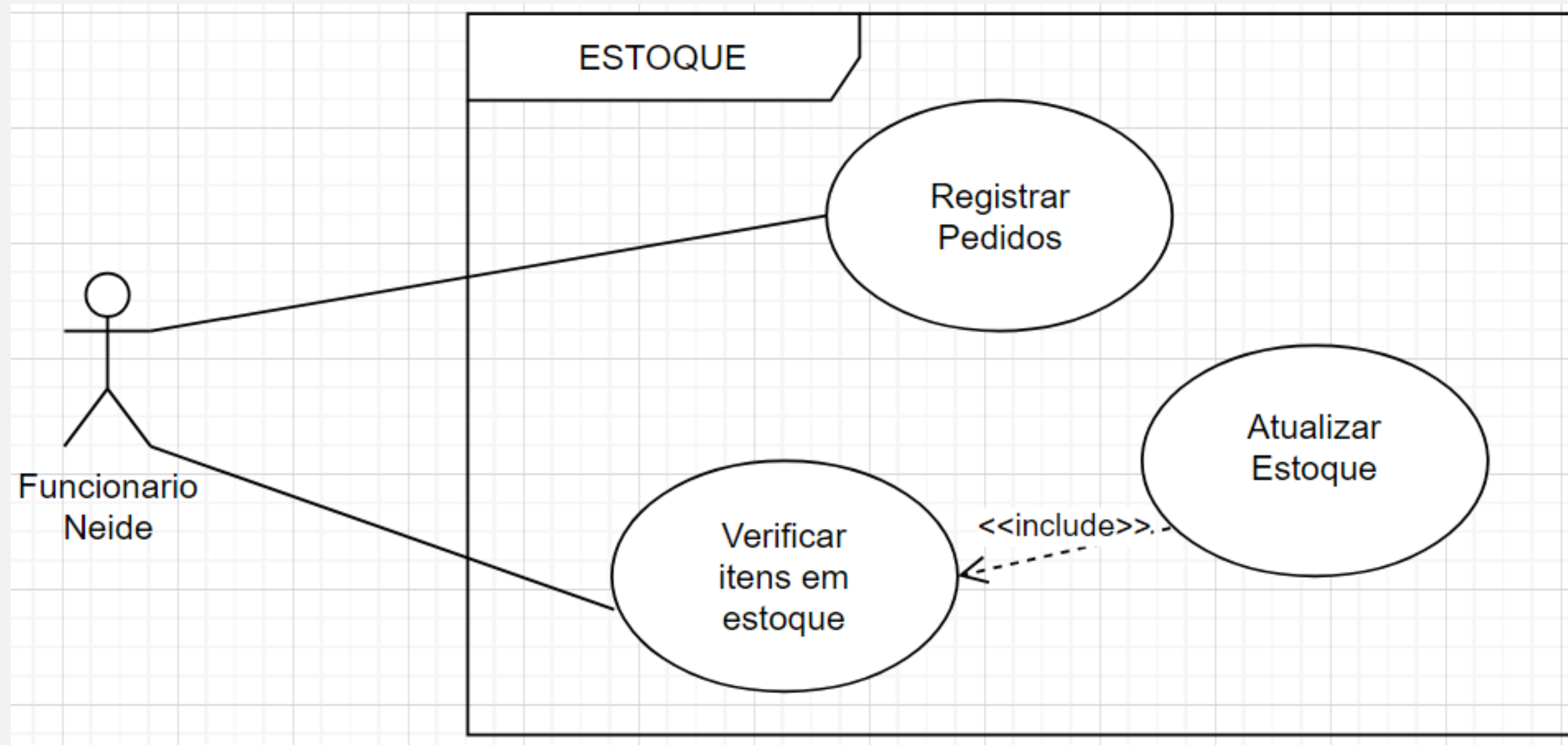
Representação visual das interações entre os atores e o sistema. Mostra as funcionalidades principais do sistema e como os usuários interagem com elas.





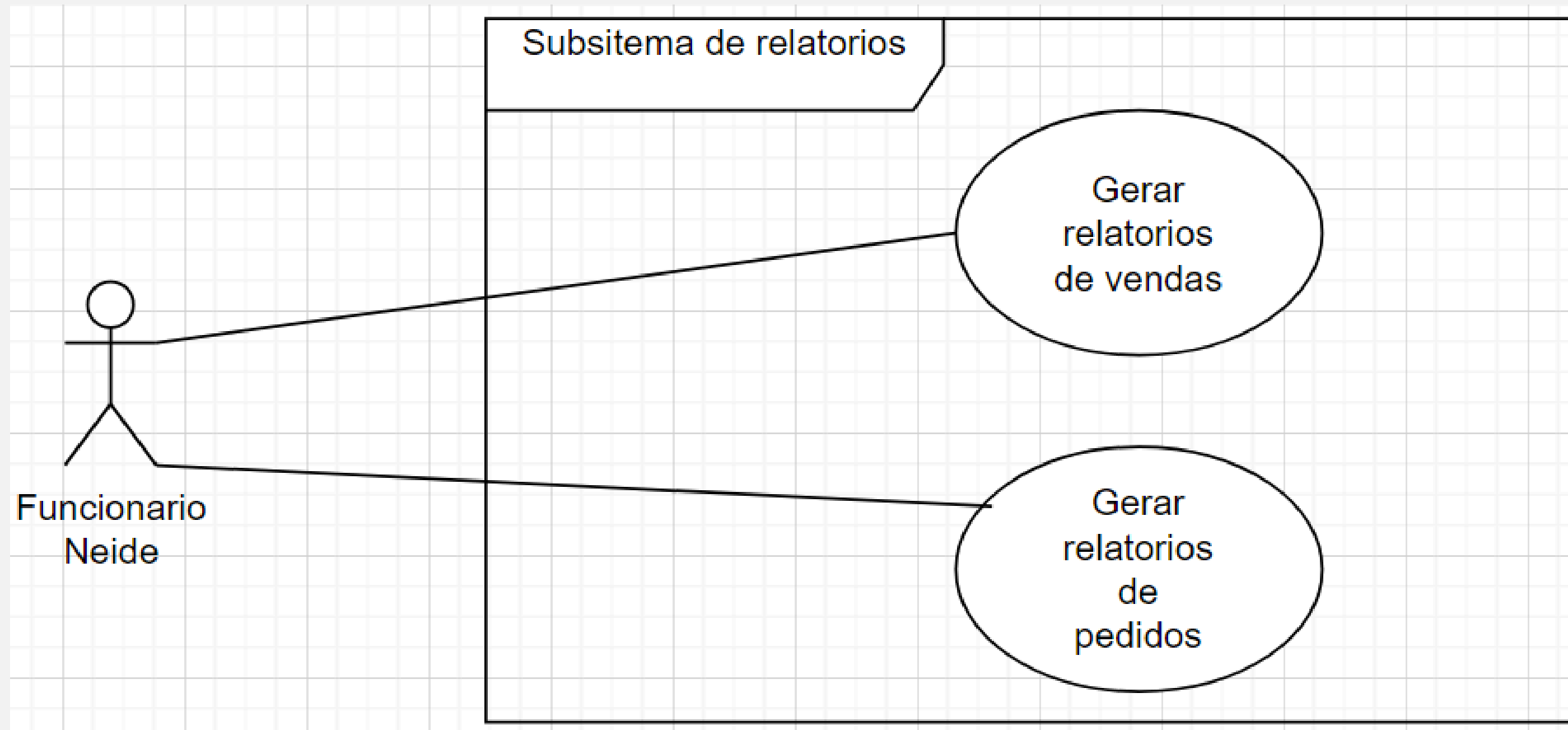
Fronteira Cardápio

A primeira fronteira envolve todas as operações relacionadas ao gerenciamento do cardápio. Ela lida com o cadastro, atualização, exibição e remoção de itens, além da possibilidade de aplicar descontos.



Fronteira Estoque

A segunda fronteira se concentra no controle de estoque. Esta fronteira lida com a verificação e atualização do estoque, além do registro de pedidos.



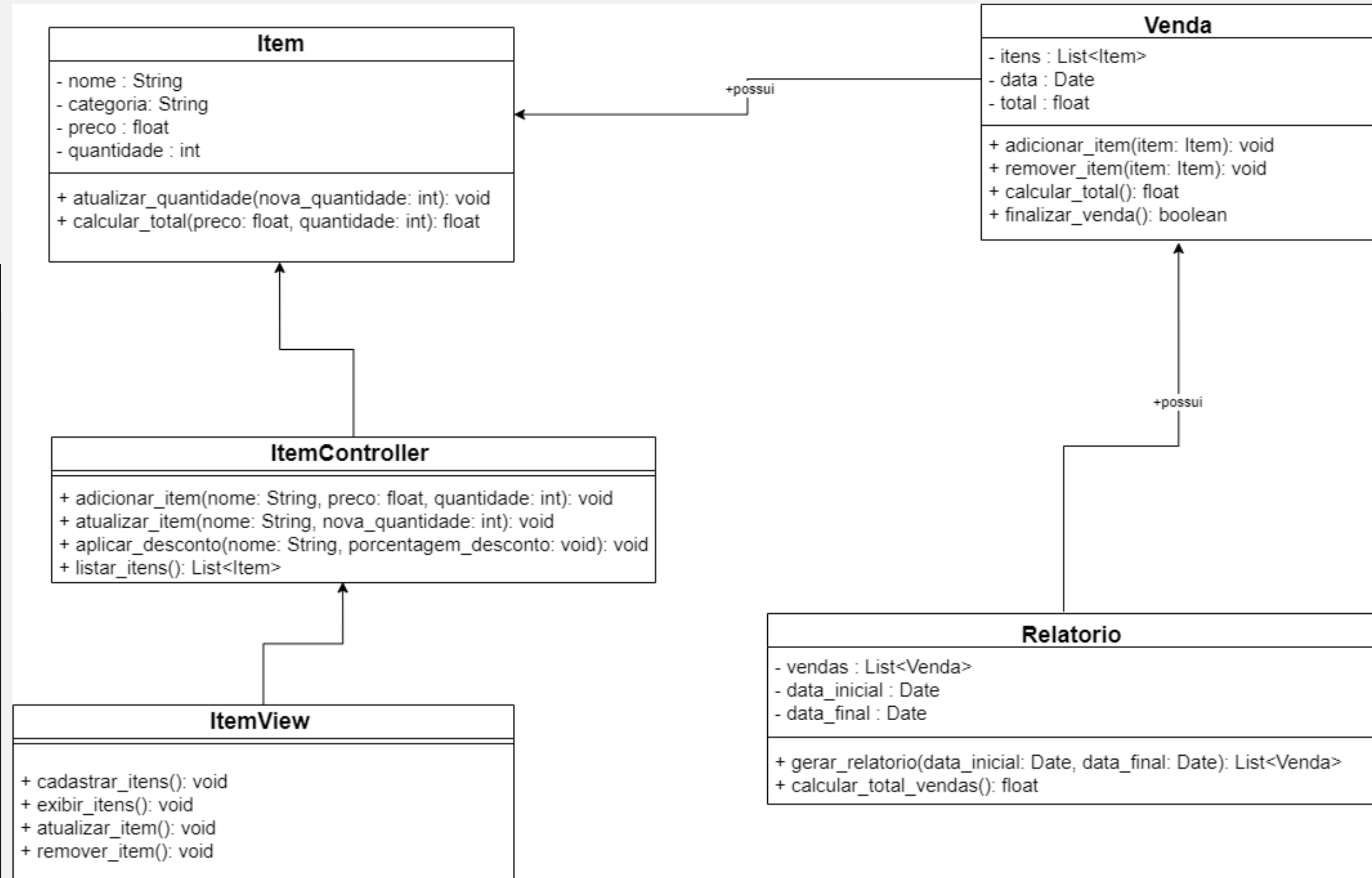
Fronteira Subsistema de Relatórios

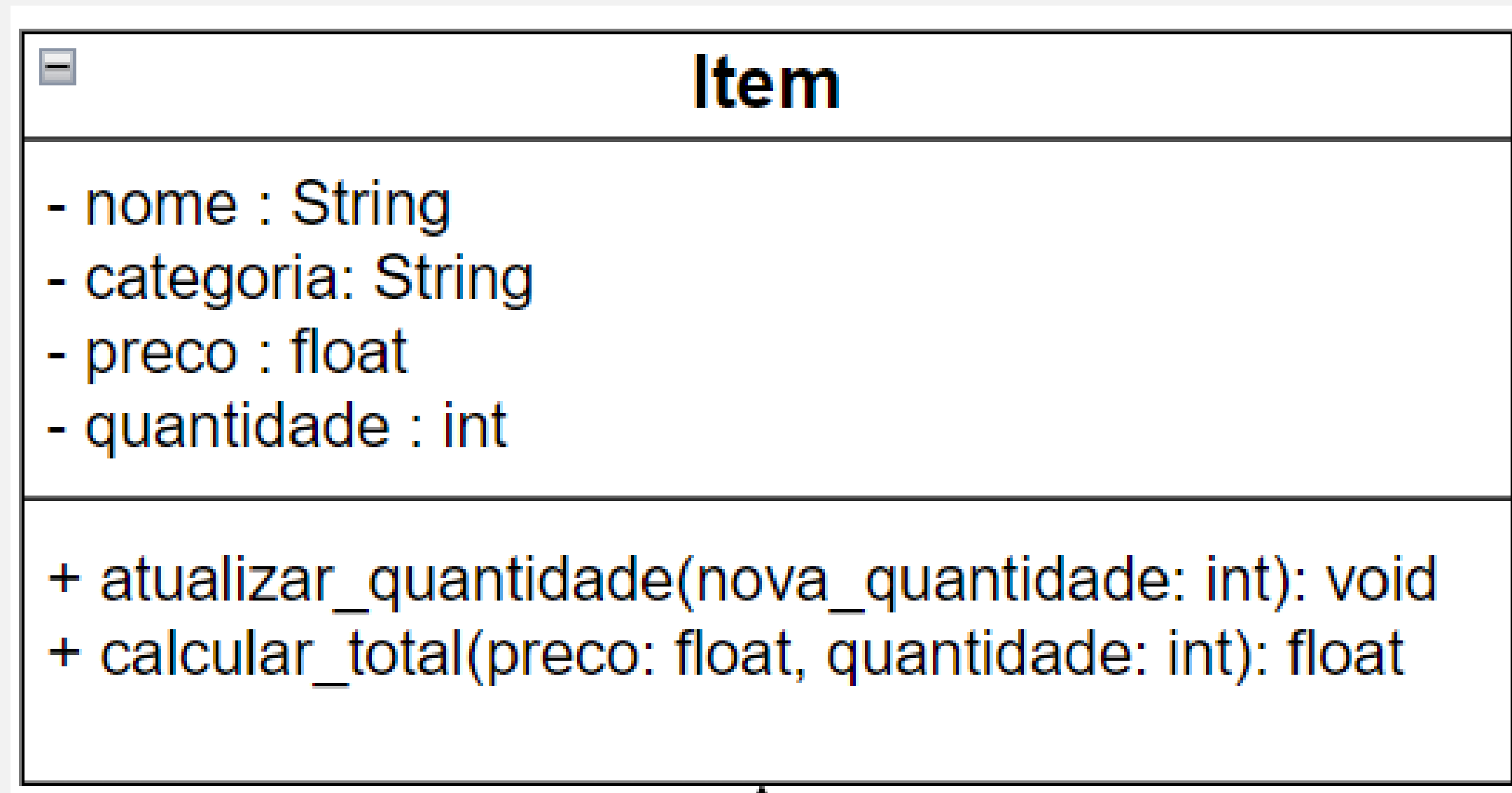
Esta fronteira está focada na geração de relatórios que auxiliam no controle financeiro e de pedidos feitos na cantina.

Diagrama de Classes

Representação visual
da estrutura do
sistema.

Mostra as classes, seus
atributos, métodos e
como elas interagem.

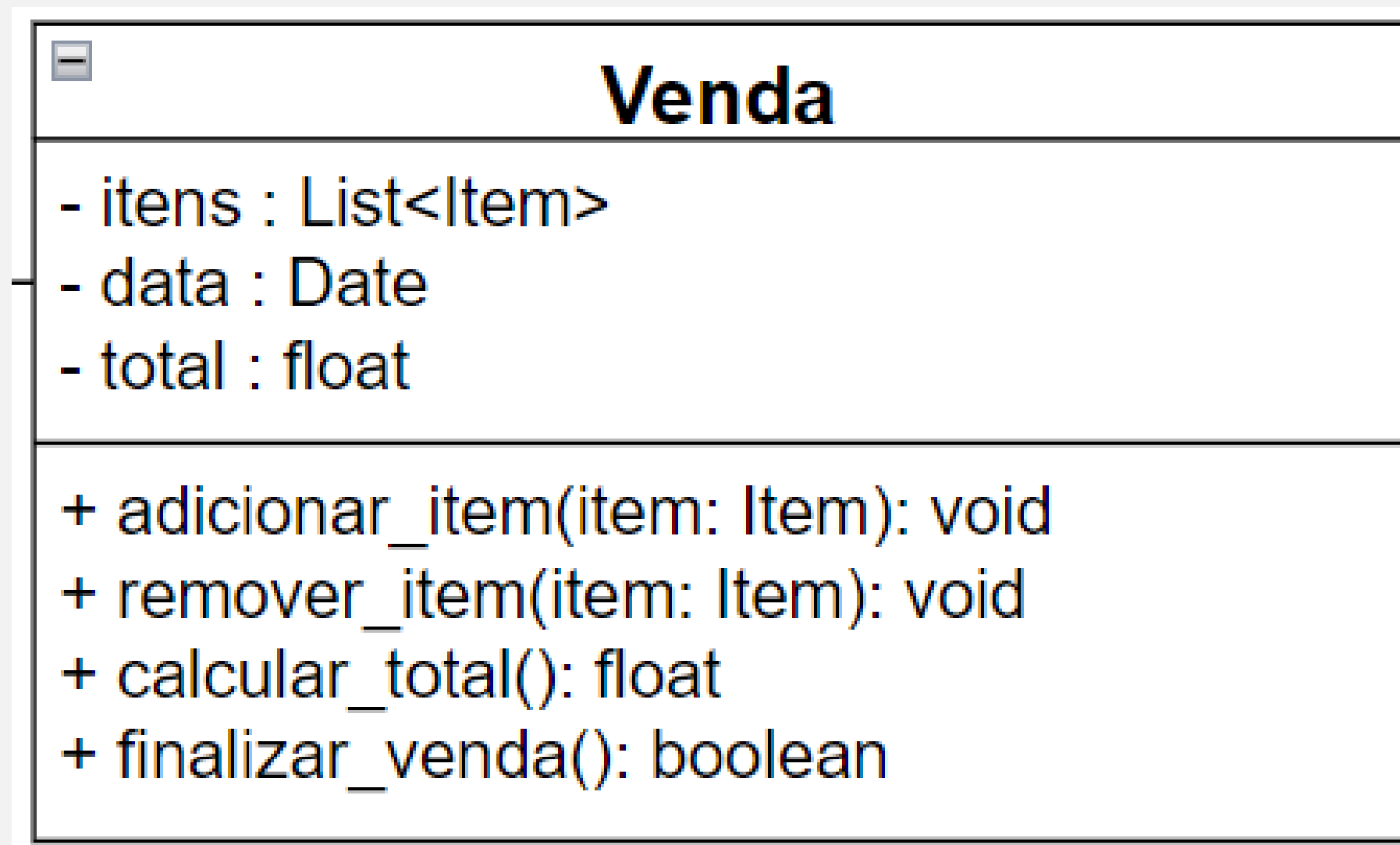




Representa um item à venda

Atributos: nome, categoria, preco, quantidade.


Métodos: atualizar_quantidade(), calcular_total().



Gerencia a venda de itens e calcula o total da venda

Atributos: itens, data, total.

Métodos: adicionar_item(), remover_item(), calcular_total().

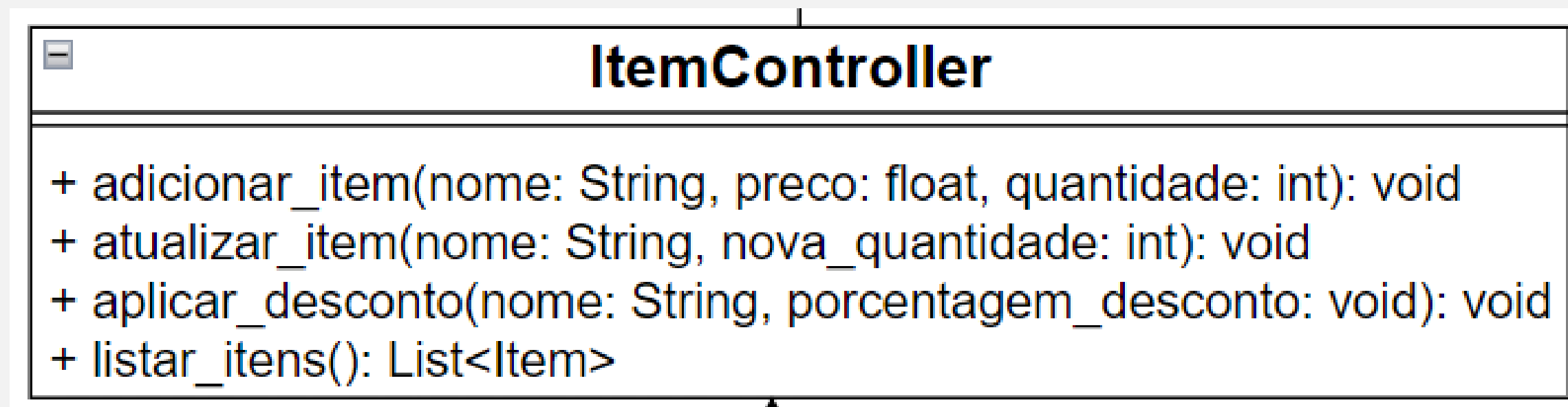


Relatorio
- vendas : List<Venda> - data_inicial : Date - data_final : Date
+ gerar_relatorio(data_inicial: Date, data_final: Date): List<Venda> + calcular_total_vendas(): float

Responsável por gerar relatórios de vendas em um período

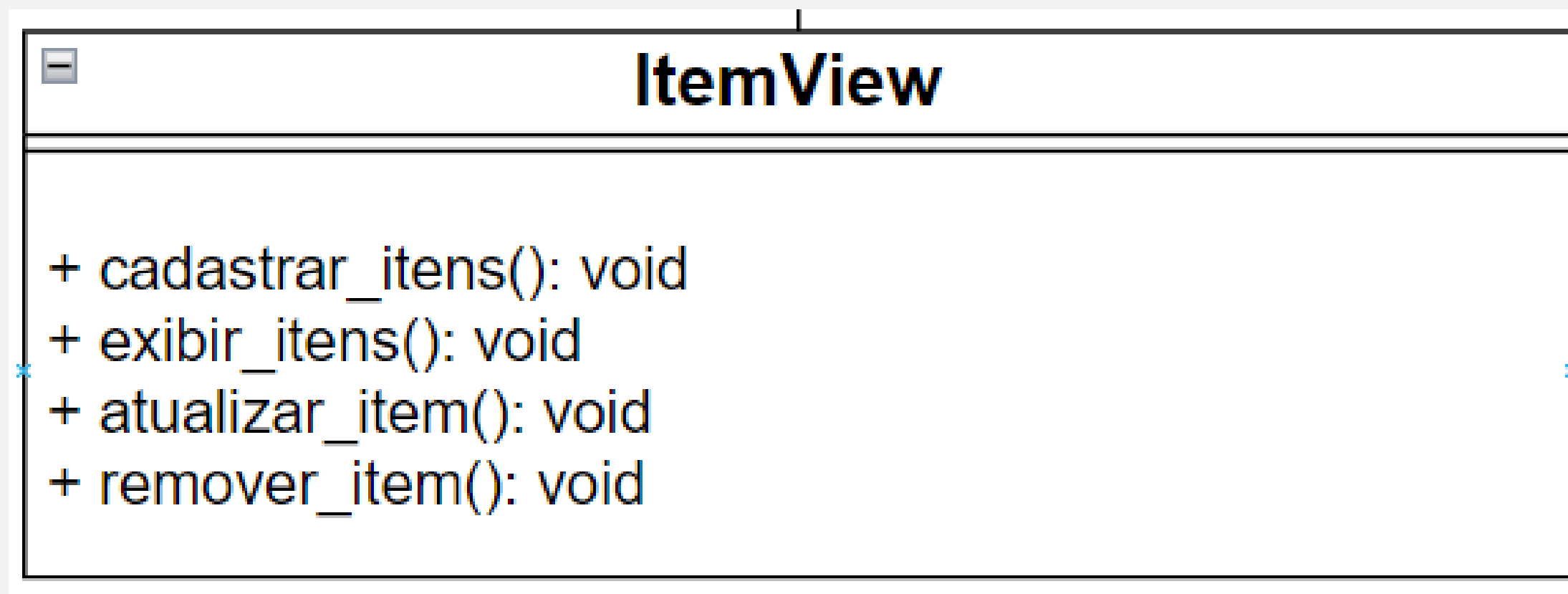
Atributos: vendas, data_inicial, data_final.

Métodos: gerar_relatorio(), calcular_total_vendas().



Controla as operações de Item

Métodos: `adicionar_item()`, `atualizar_item()`, `aplicar_desconto()`, `listar_itens()`.



Interface de interação do usuário com os itens

Métodos: `cadastrar_itens()`, `exibir_itens()`, `atualizar_itens()`, `remover_item()`

Relação entre as Classes

Untitled - TextEdit



File Edit View Help

ItemController → Item

o ItemController gerencia operações
de CRUD em Item

Relação entre as Classes

Untitled - TextEdit



File Edit View Help

ItemView → ItemController

ItemView chama os métodos de
ItemController para manipular os
itens

Venda → Item

**Relação
entre as
Classes**

Venda contém uma lista de Item

Relatorio → Venda

**Relação
entre as
Classes**

Relatorio depende de Venda para
gerar os relatórios



Princípios SOLID no Projeto

S- Single Responsibility Principle (SRP)

Cada classe possui uma única responsabilidade
(ex.: Item só representa um item)

0 - Open/Closed Principle (OCP)

As classes podem ser estendidas sem serem modificadas diretamente
(ex.: ItemController pode ser aprimorado sem alterar Item)

L - Liskov Substitution Principle (LSP):

Não é tão diretamente aplicável neste projeto,
mas o uso de interfaces em Item poderia seguir
esse princípio

... **I - Interface Segregation Principle (ISP):**

Cada classe tem uma interface específica (ex.: ItemView só lida com a visualização de itens).

D - Dependency Inversion Principle (DIP)

ItemController depende de abstrações, permitindo que outras implementações de Item sejam usadas sem alterar a lógica do controlador

```
class Item(models.Model):
    nome = models.CharField(max_length=100)
    categoria = models.CharField(max_length=50)
    preco = models.DecimalField(max_digits=10, decimal_places=2)
    quantidade = models.IntegerField()

    def atualizar_quantidade(self, nova_quantidade):
        self.quantidade = nova_quantidade
        self.save()

    def calcular_total(self, quantidade):
        return self.preco * quantidade
```

```
class Venda(models.Model):
    itens = models.ManyToManyField(Item)
    data = models.DateField(auto_now_add=True)
    total = models.DecimalField(max_digits=10, decimal_places=2, default=0)

    def adicionar_item(self, item, quantidade):
        # Lógica para adicionar item à venda
        self.itens.add(item)
        self.calcular_total()

    def calcular_total(self):
        self.total = sum(item.preco for item in self.itens.all())
        self.save()
```

```
class Relatorio:
    def __init__(self, vendas):
        self.vendas = vendas

    def gerar_relatorio(self, data_inicial, data_final):
        return [venda for venda in self.vendas if data_inicial <= venda.data <= data_final]
```

Boas Práticas que estão sendo aplicadas



Encapsulamento: Atributos privados e métodos públicos protegem a integridade dos dados.

Reuso de Código: Métodos como `calcular_total` são reutilizáveis em várias partes da aplicação.

Simplicidade e Clareza: Cada classe foca em uma única responsabilidade, facilitando o entendimento do código por outros desenvolvedores.