FATEC OSASCO

PROFESSOR PREFEITO HIRANT SANAZAR

ANDERSON MOREIRA BARBOSA

LEONARDO LEAL E SILVA

MARCOS PAULO FERREIRA DE CARVALHO

CURSO DSM – 3º SEMESTRE

TÉCNICAS DE PROGRAMAÇÃO II

PROF. VICKYBERT PESSOA FREIRE

[REVISÃO] DESIGN PATTERN

OSASCO – SP

2024.1

CONTEXTUALIZAÇÃO

O sistema apresentado é um programa de gerenciamento de vendas de produtos escolares com conceito POO usando diagrama UML sem framework, desenvolvido em Técnicas de Programação I com linguagem PHP *(TPI=1 grupo de 6 | TPII=2 grupos de 3)*.

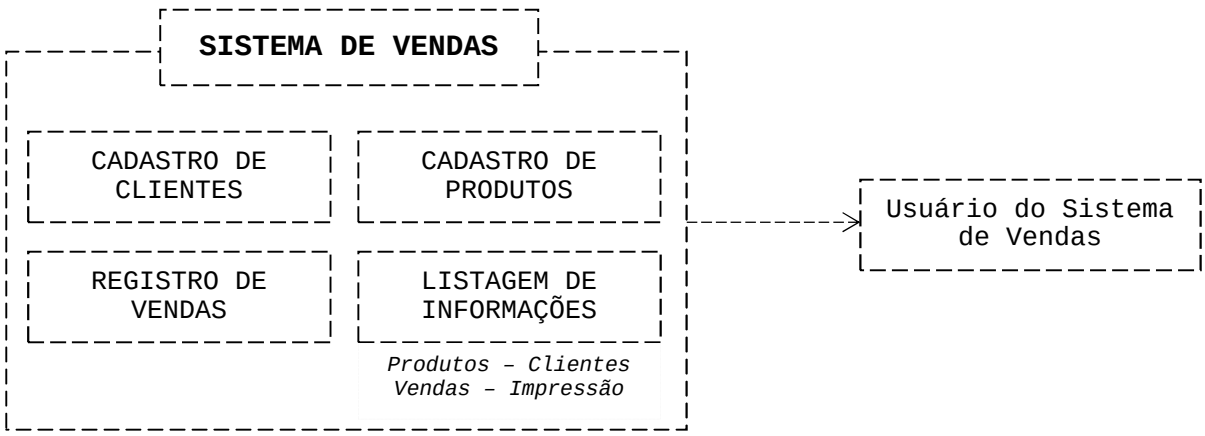
Este controle de vendas permite aos usuários registrar informações detalhadas sobre os clientes e produtos, realizar vendas associando itens aos clientes, calculando o total de vendas e gerando relatórios de vendas e clientes. Desta forma, o sistema resolve os seguintes problemas de forma organizada, automatizada e eficiente:

- ***Cadastro e Gerenciamento de Produtos –*** Permite aos usuários adicionarem novos produtos ao sistema, armazenar informações como descrição, estoque, preço e unidade de medida.
- ***Cadastro e Gerenciamento de Clientes –*** Permite o cadastro de clientes, armazenando informações pessoais e de contato.
- ***Registro de Vendas –*** Permite registrar vendas associadas a clientes cadastrados, adicionando produtos às vendas, aplicando descontos e calculando o valor total da venda.
- ***Listagem de Registros –*** Permite a listagem de todos os produtos, clientes e vendas cadastrados, facilitando a visualização e gerenciamento desses registros.
- ***Impressão de Pedidos –*** Permite a impressão de detalhes específicos de pedidos, incluindo informações do cliente e dos produtos comprados.

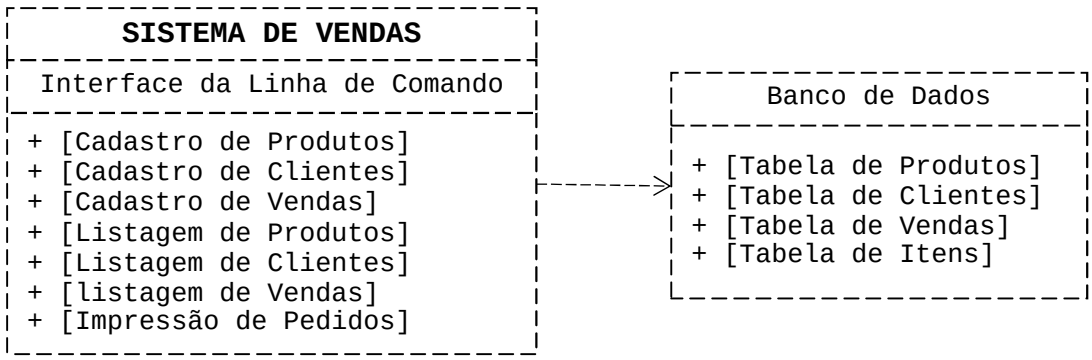
DIAGRAMAS

Estrutura do Sistema Atual

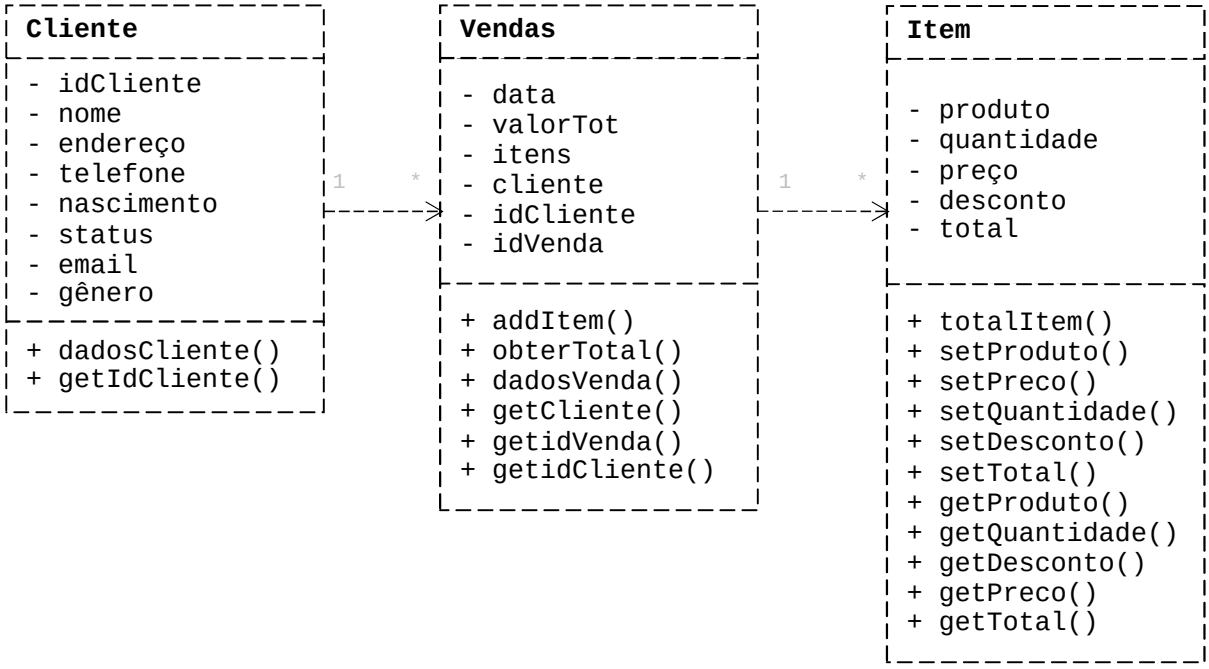
- Diagrama de Contexto (C4 Model - Nível 1)

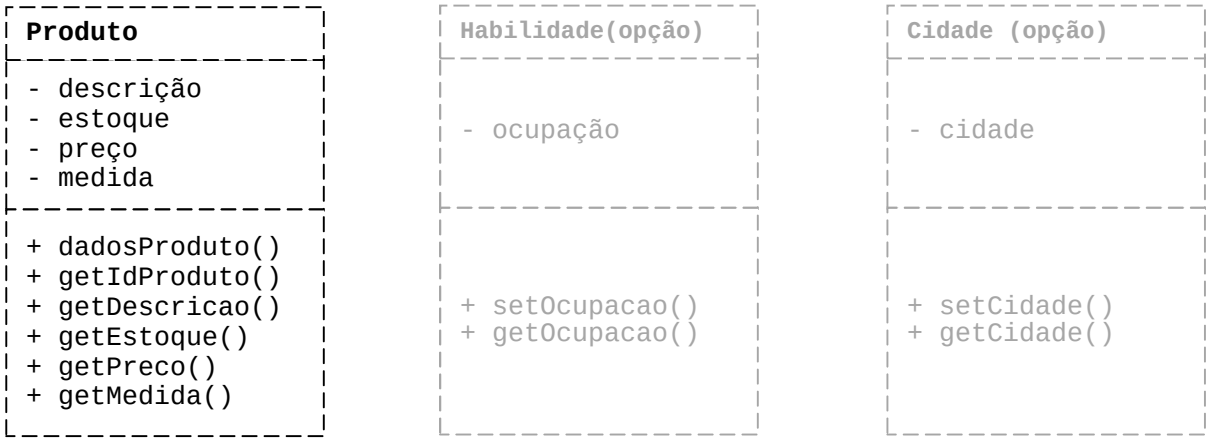


- Diagrama de Containers (C4 Model - Nível 2)



- Diagrama de Classes do Sistema Atual





Padrões Patterns que serão empregados

- Factory Pattern
- Singleton Pattern
- Strategy Pattern

IMPLEMENTAÇÃO

Refatoramos o script para implementar os padrões *FACTORY*, *SINGLETON* e *STRATEGY*, com separação de responsabilidades, melhorando a manutenção e extensão do código.

Criada a classe `*DataManager*` para gerenciar e centralizar o armazenamento dos dados relacionados aos clientes, produtos e vendas na aplicação. Seguindo o padrão *SINGLETON*, em vez de ter múltiplos arrays ou listas dispersos pelo código, todas essas informações serão armazenadas em um único lugar, evitando inconsistências nos dados e garantindo que todas as operações de leitura e escrita ocorram no mesmo conjunto de dados. Em resumo, garantimos que apenas uma instância dessa classe exista durante a execução do programa.

Também implementamos a interface `*PricingStrategy*` definindo como todas as estratégias de precificação deverão seguir. Ela força todas as classes que a implementam a fornecer uma implementação para o método `*calculateTotal*`. Isso promove a flexibilidade e a extensibilidade do código.

Além disso, foram realizadas algumas correções e adaptações em todo o código para suportar as novas classes.

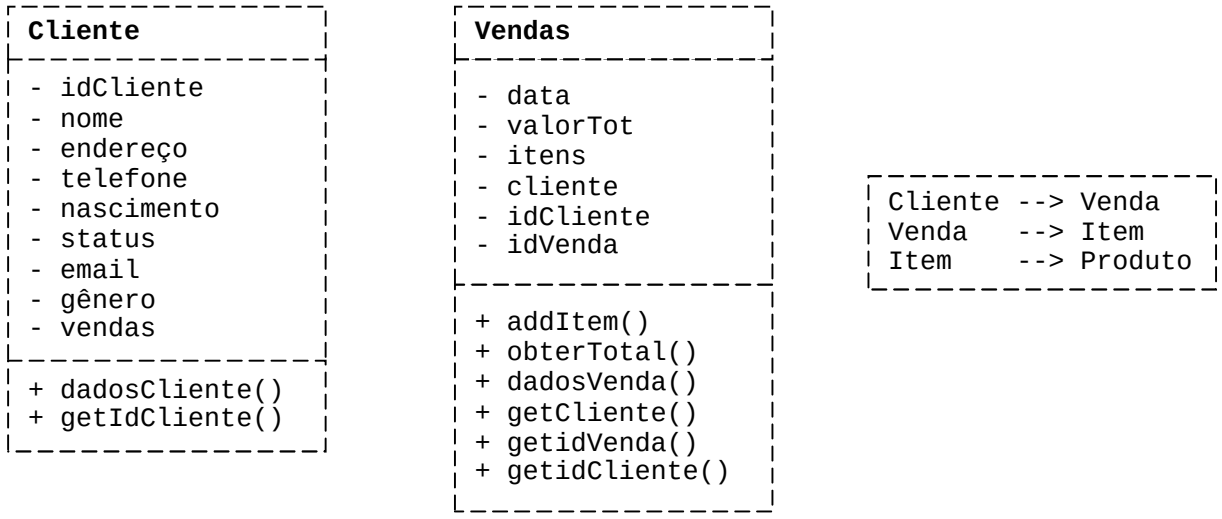
Descrição da funcionalidade

A classe `*DataManager*` é privada, evitando que outras instâncias sejam criadas e adicionada nesta classe. Também pela adoção do método `*getInstace*` é verificado se uma instância já existe, criando uma nova se necessário ou retornando a existente. Em conjunto, o uso de arrays privados para armazenamento de dados e os métodos públicos para adicionar e recuperar tais dados. Em resumo, todos os dados serão acessados e modificados através de uma única instância.

Já o uso da interface `*PricingStrategy*` fornecerá uma maneira flexível e extensível de calcular o total de vendas, permitindo que a aplicação se adapte facilmente a diferentes requisitos de precificação sem modificar a estrutura existente do código.

Diagramas de Classes Pré-Modificação

https://onlinegdb.com/3yRAMLo6G



Descrição dos Patterns Aplicados

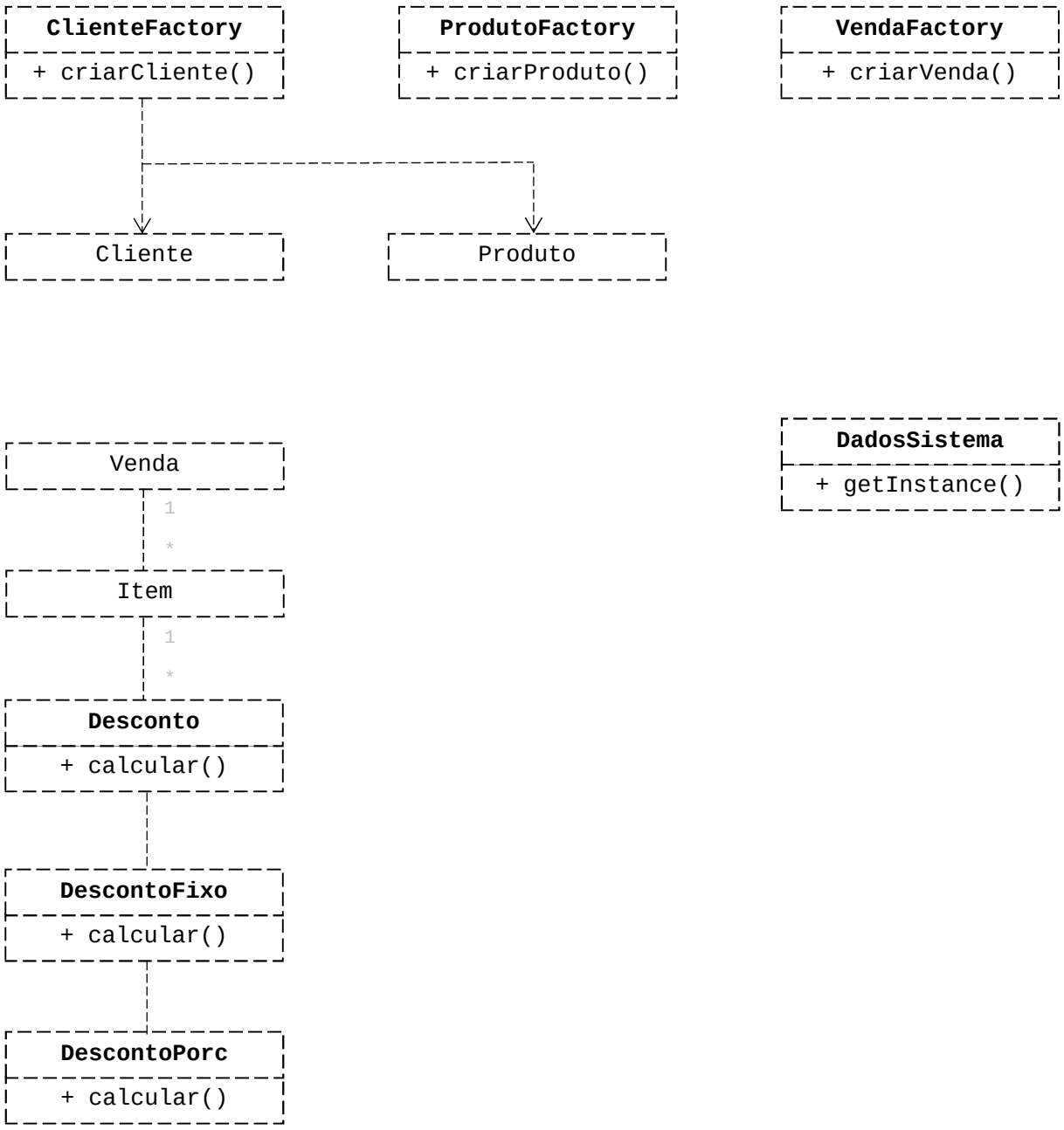
- **Factory Pattern –** Facilita a criação de objetos Cliente e Produto, evitando a duplicação de código e promovendo a flexibilidade na criação de instâncias. É particularmente adequado porque separa a lógica de criação do resto do sistema, permitindo mudanças nas classes concretas sem afetar o código cliente.

- **Singleton Pattern –** Será implementado para garantir que o gerenciamento de clientes, produtos e vendas seja feito através de uma única instância centralizada de cada um desses componentes.

- **Strategy Pattern –** Será empregado para implementar diferentes estratégias de cálculo de desconto nos itens das vendas.

Diagramas de Classes Pós-Modificação

https://onlinegdb.com/Gof_70S-T



➢ Trechos de Código mais Relevantes

```php
//FACTORY para criação de objetos Cliente e Produto
class Factory {
    public static function createCliente($nome, $endereco, $telefone, $nascimento, $status, $email, $genero) {
        return new Cliente($nome, $endereco, $telefone, $nascimento, $status, $email, $genero);
    }

    public static function createProduto($descricao, $estoque, $preco, $medida) {
        return new Produto($descricao, $estoque, $preco, $medida);
    }

    public static function createVenda($cliente, $idCliente) {
        return new Venda($cliente, $idCliente);
    }

    public static function createItem() {
        return new Item();
    }
}
```

```php
//SINGLETON para gerenciar dados dos produtos, clientes e vendas.
class DataManager {
    private static $instance;
    private $clientesCad;
    private $produtosCad;
    private $vendasCad;

    private function __construct() {
        $this->clientesCad = array();
        $this->produtosCad = array();
        $this->vendasCad = array();
    }

    public static function getInstance() {
        if (self::$instance === null) {          //se a instância não foi criada, um novo Gerenciador de Dados será feito.
            self::$instance = new DataManager();
        }
        return self::$instance;
    }
```

```php
//STRATEGY para diferentes estratégias de preços.
interface PricingStrategy {
    public function calculateTotal($quantity, $price, $discount);
}
class StandardPricingStrategy implements PricingStrategy {
    public function calculateTotal($quantity, $price, $discount) {
        return $quantity * $price * (1 - $discount);
    }
}
```

➢ Classes originais refatoradas

```php
//Menu Principal do Programa
$dataManager = DataManager::getInstance();
```

```php
class Cliente {
    protected $nome;
    protected $endereco;
    protected $telefone;
    protected $nascimento;
    protected $status;
    protected $email;
    protected $genero;

    private static $contador = 0;      //para gerar um novo ID a cada novo cliente
    protected $idCliente;              //gerado com a propriedade estática acima
    protected $vendas;                 //array que receberá todas as vendas do cliente

    function __construct($nome, $endereco, $telefone, $nascimento, $status, $email, $genero) {
        $this->nome = $nome;
        $this->endereco = $endereco;
        $this->telefone = $telefone;
        $this->nascimento = $nascimento;
        $this->status = $status;
        $this->email = $email;
        $this->genero = $genero;
        $this->vendas = array();                    //inicia como array vazio
        self::$contador++;                          //contador é incrementado
        $this->idCliente = 'C' . self::$contador;  //ID recebe o novo valor do contador
    }
```

```php
class Produto {
    protected $descricao;
    protected $estoque;
    protected $preco;
    protected $medida;

    private static $contador = 0;  //variável estática para manter o contador de produtos
    protected $idProduto;          //gerado com a propriedade estática acima

    function __construct($descricao, $estoque, $preco, $medida) {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco = $preco;
        $this->medida = $medida;
        self::$contador++;                          //contador é incrementado
        $this->idProduto = 'P' . self::$contador;  //ID recebe o novo valor do contador
    }
```

```php
class Venda {
    protected $cliente;              //recebe o cliente cadastrado
    protected $itens;                //array para guardar os produtos vendidos [todo]
    private static $contador = 0;  //para gerar um novo ID a cada nova venda
    protected $idVenda;              //recebe o ID pela propriedade contador
    protected $data;
    protected $valorTot;

    public function __construct(Cliente $cliente, $idCliente) {
        $this->cliente = $cliente;
        $this->idCliente = $idCliente;
        $this->data = date('d-m-Y H:i:s');

        $this->itens = array();                          //cria o array para guardar os produtos vendidos
        self::$contador++;                               //contador é incrementado
        $this->idVenda = 'PED' . self::$contador;  //ID recebe o novo valor do contador
    }

    public function addItem(Item $item) {
        $this->itens[] = $item;
    }

    //calcula o total geral da venda [soma todos os itens vendidos]
    public function obterTotal() {
        $total = 0;
        foreach ($this->itens as $item) {
            $total += $item->getTotal();
        }
        $this->valorTot = $total;  //armazena o total na propriedade $valorTot
        return $total;
    }
```

```php
class Item {
    protected $produto;
    protected $quantidade;
    protected $preco;
    protected $desconto;
    protected $total;

    //associa o produto ao método addItem
    public function setProduto(Produto $produto) {
        $this->produto = $produto;
    }

    //setters e getters
    public function setPreco($preco) {
        $this->preco = $preco;
    }
    public function setQuantidade($quantidade) {
        $this->quantidade = $quantidade;
    }
    public function setDesconto($desconto) {
        $this->desconto = $desconto;
    }
    public function setTotal($total) {
        $this->total = $total;
    }
    public function getTotal() {
        return $this->total;
    }
```

PROJETO ORIGINAL
https://onlinegdb.com/3yRAMLo6G

PROJETO COM IMPLEMENTAÇÃO
https://onlinegdb.com/Gof_70S-T