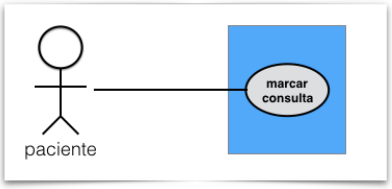
CONTINUAÇÃO SOBRE UML (Diagrama de caso de uso)

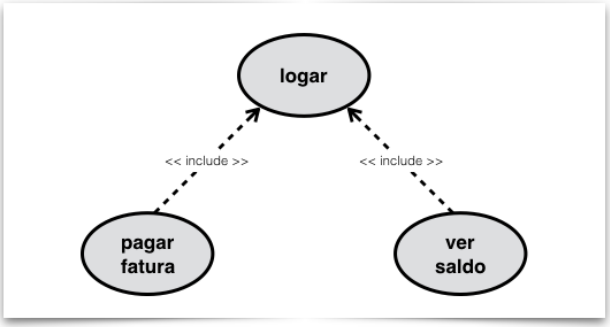
Tipos de Relacionamentos

- **Relacionamento de Comunicação** ou Associação: representa a interação entre um ator e um caso de uso por meio de mensagens. É representado por uma linha sólida.



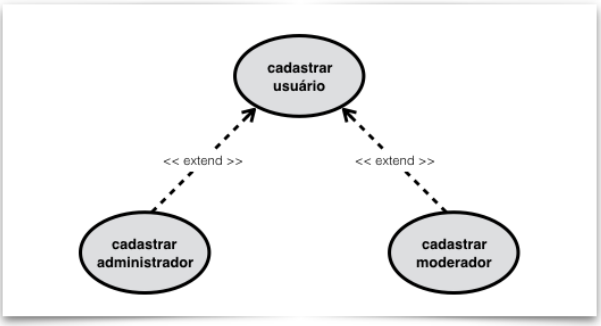
Funcionalidade do ponto de vista do usuário.

- **Relacionamento de Inclusão**: utilizado quando um comportamento se repete em mais de um caso de uso. Por exemplo, num internet banking, um cliente que vai realizar um pagamento precisa se logar., assim como um cliente que vai visualizar o saldo também precisa se logar.



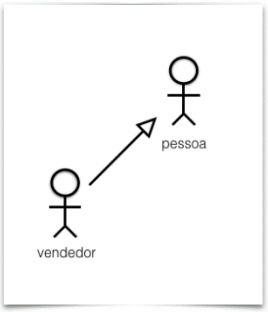
Logar é essencial para pagar fatura e para ver saldo. Ou Logar é parte de pagar fatura e também é parte de ver saldo.

- **Relacionamento de Extensão**: utilizado quando se deseja modelar um relacionamento alternativo. Por exemplo, ao "cadastrar usuário" num sistema de forum, podemos "cadastrar um administrador" ou "cadastrar um moderador".



Cadastrar administrador e cadastrar moderador são extensões de cadastrar usuário. Eles não são essenciais, só contém eventos adicionais sob certas condições.

- **Relacionamento de Herança**: é um relacionamento entre atores, utilizado quando queremos representar uma especialização/generalização. Na figura a seguir, vendedor é especialização de pessoa (ou pessoa é generalização de vendedor), é representado por um alinha com um triângulo.



Os casos de uso de pessoa são também casos de uso de vendedor. Vendedor tem seus próprios casos de uso.

UML — Diagrama de Classe

Diagramas de classes são os componentes básicos da UML. Os diversos componentes em um diagrama de classes podem representar as classes que serão realmente programadas, os principais objetos ou as interações entre classes e objetos.

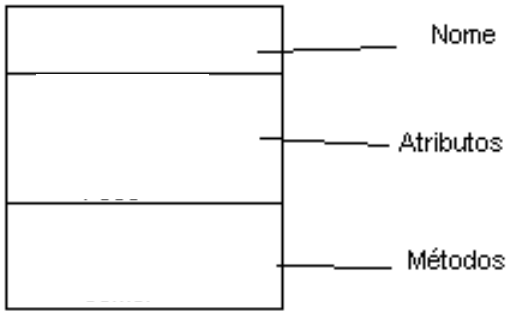
Os benefícios de diagramas de classes

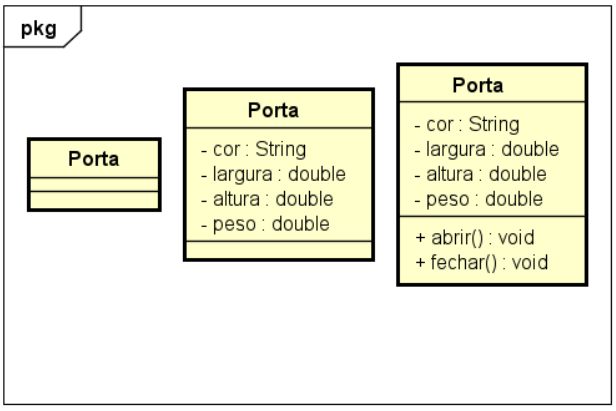
Diagramas de classes oferecem uma série de benefícios para qualquer organização. Use diagramas de classes UML para:

- Ilustrar modelos de dados para sistemas de informação, não importa quão simples ou complexo.
- Entender melhor a visão geral dos esquemas de uma aplicação.
- Expressar visualmente as necessidades específicas de um sistema e divulgar essas informações por toda a empresa.
- Criar gráficos detalhados que destacam qualquer código específico necessário para ser programado e implementado na estrutura descrita.
- Fornecer uma descrição independente de implementação de tipos utilizados em um sistema e passados posteriormente entre seus componentes.

Os diagramas de classe são formados por uma série de elementos gráficos. A seguir veremos alguns desses elementos:

Classe





Relação entre classes

Existem vários tipos de relação entre classes na UML (Unified Modeling Language). Alguns dos tipos mais comuns incluem:

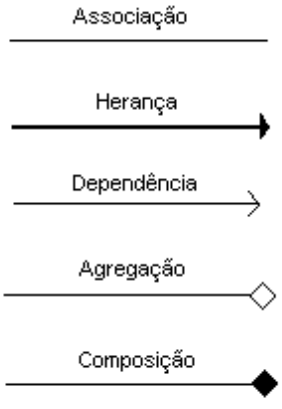
1. **Associação**: é a relação mais básica entre duas classes. Ela representa um relacionamento entre duas classes onde um objeto de uma classe é associado a um ou mais objetos de outra classe. Por exemplo, a classe "Pessoa" pode estar associada à classe "Endereço".

2. **Agregação**: é uma associação especial que indica uma relação de "parte-de" entre duas classes, onde uma classe é a classe proprietária e a outra é a classe parte. Por exemplo, uma classe "Carro" pode ter uma agregação com a classe "Motor".

3. **Composição**: é uma agregação forte que indica uma relação de "todo-parte" entre duas classes, onde a parte não pode existir sem o todo. Por exemplo, uma classe "Carro" pode ter uma composição com a classe "Roda".

4. **Herança**: é uma relação onde uma classe (a subclasse) é derivada de outra classe (a superclasse). A subclasse herda todos os atributos e métodos da superclasse. Por exemplo, a classe "Gerente" pode ser uma subclasse da classe "Funcionário".

5. **Dependência**: é uma relação onde uma classe depende de outra classe para realizar alguma operação. Por exemplo, a classe "Pedido" pode depender da classe "Cliente" para obter informações do cliente.

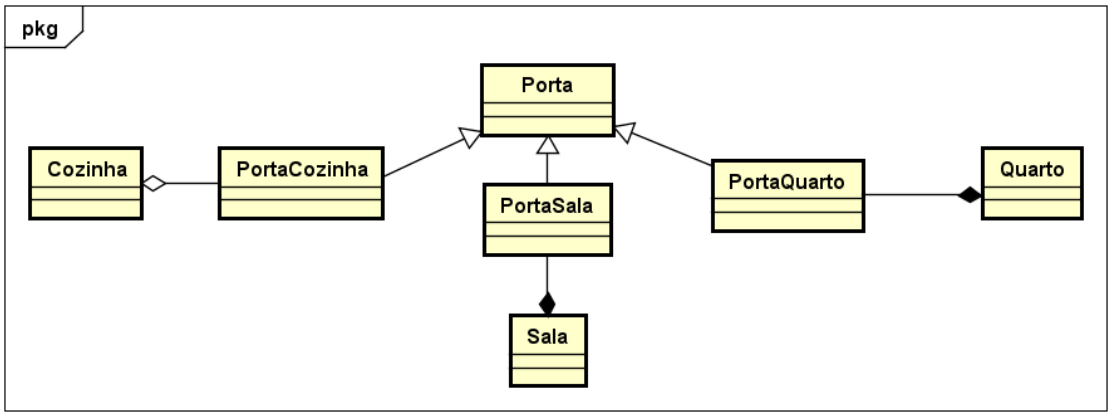


Esses são apenas alguns exemplos dos tipos de relação entre classes que podem ser definidos na UML. É importante escolher o tipo de relação correto para representar corretamente o relacionamento entre as classes em seu modelo.

Quando fazemos o uso de um diagrama de classes no dia a dia da produção de software, nem sempre é necessário/relevante representar cada classe no menor nível de detalhe , ou seja, com os três compartimentos, e com todo rigor nas especificações dos atributos e operações.

Abaixo temos três exemplos de um mesmo diagrama, em níveis de detalhes diferentes. No último exemplo vamos ver detalhes de cada classe e seus relacionamentos.

Exemplo 1



No diagrama cima temos relacionamentos de Associação, Agregação, Composição e Generalização (Herança). A explicação a seguir aplica-se a todos os três exemplos, pois foca apenas nos relacionamentos:

- Cozinha pode ter ou não uma PortaCozinha, podendo existir se não tiver. (Agregação)

- PortaCozinha generaliza Porta, possuindo todas as características que Porta têm, além das suas específicas. (Generalização)

- Quarto deve ter PortaQuarto, não podendo existir se não tiver. (Composição)

- PortaQuarto generaliza Porta, que tem todas as características que Porta têm, além das suas específicas. (Generalização)

- Sala deve ter PortaSala, não podendo existir se não tiver. (Composição)

- PortaSala generaliza Porta, que tem todas as características que Porta têm, além das suas específicas. (Generalização)

- Sala pode ter ou não uma Porta que não seja uma PortaSala, mas se tiver ou não isso não fará diferença, pois Porta pode existir sem Sala, e Sala pode existir sem Porta. (Associação).

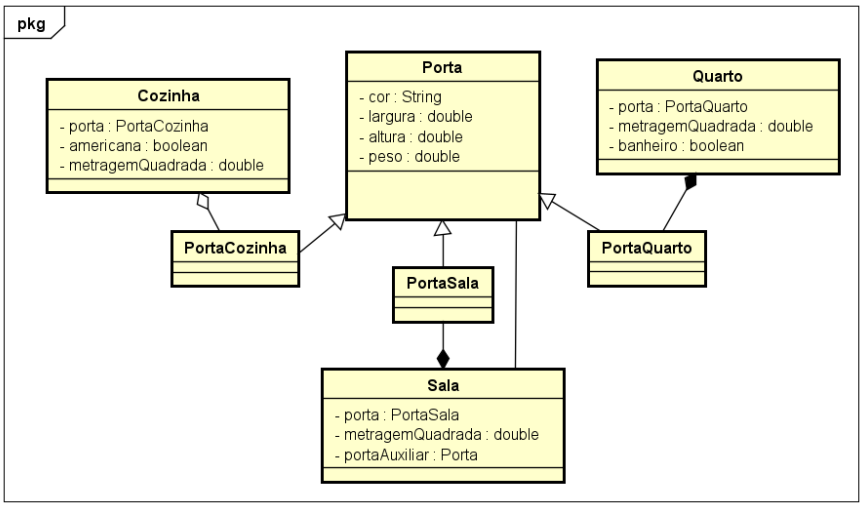
Este exemplo de representação acima é muito usado/recomendado quando:

- Uma equipe está discutindo um problema e algum profissional quer esboçar (esboço = rascunho, “rabisco”) como pensa na solução, em termos de arquitetura.

- Um profissional quer mostrar apenas as dependências entre as classes do sistema, para uma análise de impacto ou contextualização da arquitetura.

- Não há necessidade de entrar em maiores detalhes sobre as classes, apenas identificá-las e ilustrar suas relações.

Exemplo 2



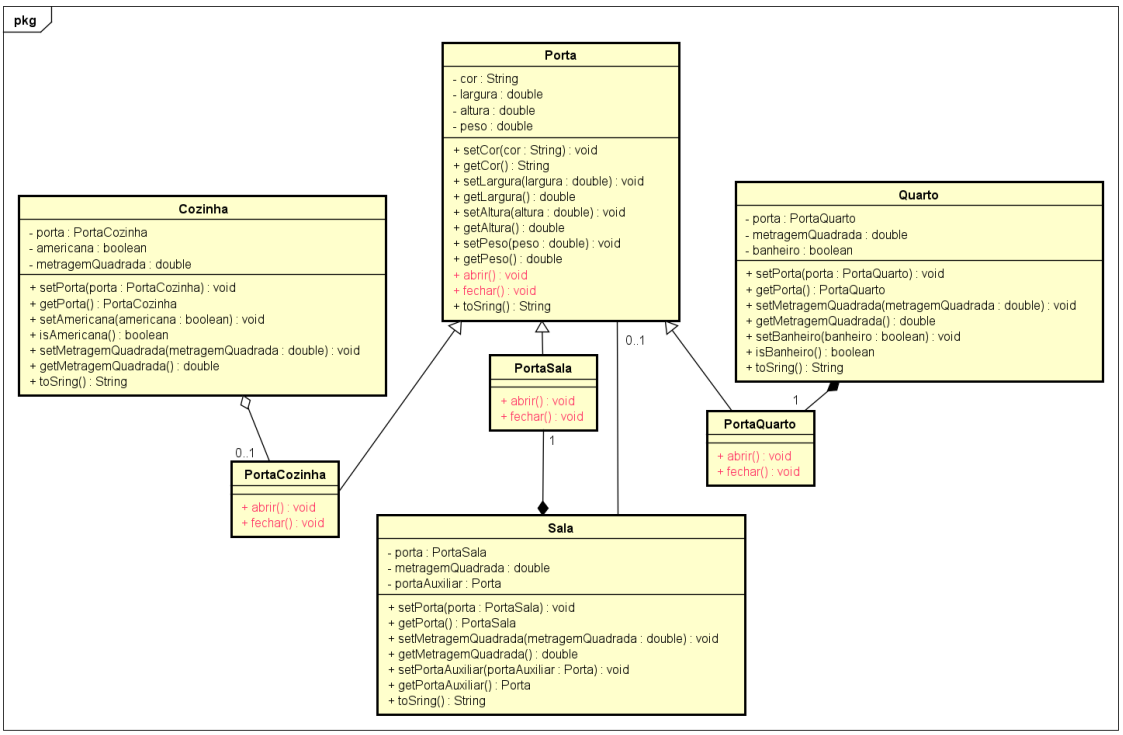
O diagrama acima já é um pouco diferente do primeiro, pois além dos compartimentos com os nomes das classes, possui também um outro compartimento contendo os atributos da classe (as classes sem atributos são classes “filhas” de outras [generalização], que, portanto, implicitamente possuem todos os atributos que a classe “mãe” possui).

Este exemplo de representação acima é muito usado/recomendado quando:

- O objetivo é demonstrar as classes, seus relacionamentos, seus atributos e não há necessidade de detalhar as operações da classe.

- O profissional precisa (ou entende ser necessário) dar mais contexto às classes, detalhando seus atributos, para que se compreenda melhor o escopo de cada classe do modelo e como isso compõe o entendimento sobre as relações entre as classes.

Exemplo 3



O diagrama acima já é bem diferente do primeiro e do segundo, pois além dos compartimentos com os nomes das classes e atributos, possui também um outro compartimento contendo as operações da classe.

Este exemplo de representação acima é muito usado/recomendado quando:

- O objetivo é demonstrar as classes, seus relacionamentos, e cada classe com seu escopo completo.

- Quando a empresa realiza projeto formal do software, utilizando ferramentas case, modelos de classes que serão utilizados para outra empresa (ou a mesma até) para entendimento sobre o software a ser construído.

- Muito cobrado em empresas que prestam serviço para órgãos públicos através de licitação.

- O profissional precisa (ou entende ser necessário) dar 100% de contexto às classes, detalhando seus atributos e suas operações, para que se compreenda melhor o escopo de cada classe do modelo e como isso compõe o entendimento sobre as relações entre as classes.

Vejam a seguir como fica a implementação de um digrama de classe em código, utilizaremos a linguagem Java, mas sempre lembrando que essa implementação poderá ser feita em qualquer linguagem que permita o desenvolvimento de softwares no paradigma Orientado a Objeto (OO).

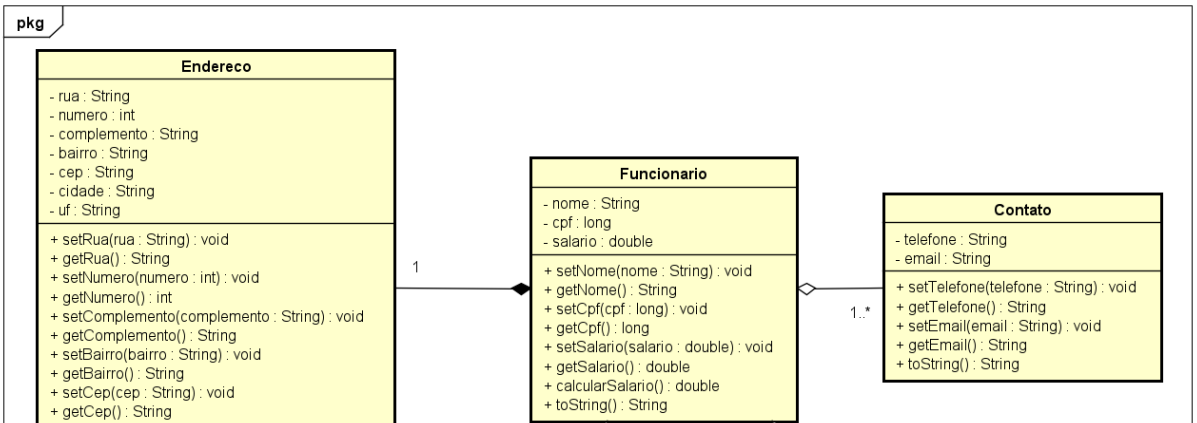


Diagrama acima mostrando como as classes estão relacionadas perante a um sistema de vendas (básico).

Agora veja como esse diagrama fica em Java, classe por classe:

Classe Cliente.java:

```java
public class Cliente{
    private String nome;
    private long cpf;
    private List<Contato> contatos;
    private List<Endereco> enderecos;

    public void setNome(String nome){
        this.nome = nome;
    }

    public String getNome(){
        return nome;
    }

    public void setCpf(long cpf){
        this.cpf = cpf;
    }

    public long getCpf() {
        return cpf;
    }

    public void setContatos(List<Contato> contatos){
        this.contatos = contatos;
    }

    public List<Contato> getContatos(){
        return contatos;
    }

    public void setEnderecos(List<Endereco> enderecos){
        this.enderecos = enderecos;
    }

    public List<Endereco> getEndereco(){
        return enderecos;
    }

    public String toString() {
        return "\n" +
        "Nome: "+nome+"\n" +
        "Cpf: "+cpf+"\n" +
        "\nEndereço: "+enderecos+"\n" +
        "\nContato: "+contatos;
    }
}
```

Classe Funcionario.java:

```java
import java.util.Collection;
public class Funcionario {
    private String nome;
    private long cpf;
    private double salario;
    private Endereco;
    private List<Contato> contatos;

    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
    public String getNome() {
        return nome;
    }

    public void setCpf(long cpf) {
        this.cpf = cpf;
    }

    public long getCpf() {
        return cpf;
    }

    public void setSalario(double salario) {
        this.salario = salario;
    }

    public double getSalario() {
        return salario;
    }

    public void setEndereco(Endereco endereco){
        this.endereco = endereco;
    }
    public Endereco getEndereco(){
        return endereco;
    }

    public void setContatos(List<Contato> contatos){
        this.contatos = contatos;
    }

    public List<Contatos> getContatos(){
        return contatos;
    }

    public double calcularSalario() {
        return 0;//Aqui vai a implementação (código para calcular o salário)
    }

    public String toString() {
        return "\n" +
        "Nome: "+nome+"\n" +
        "Cpf: "+cpf+"\n" +
        "Salário: "+salario+"\n" +
        "\nEndereço: "+endereco+"\n" +
        "\nContato: "+contatos;
    }
}
```

Classe Vendedor.java:

```
public class Vendedor extends Funcionario {
    private List<Venda> vendas;

    public void setVendas(List<Venda> vendas){
        this.vendas = vendas;
    }

    public List<Venda> getVendas(){
        return vendas;
    }

    public double receberComissao() {
        return 0; //Aqui vai a implementação (código para calcular a comissão)
    }

    public String toString() {
        return "\n" +
        "Nome: "+getNome()+"\n" +
        "Cpf: "+getCpf()+"\n" +
        "Salário: "+getSalario()+"\n" +
        "\nEndereço: "+getEndereco()+"\n" +
        "\nContato: "+getContatos()+"\n"+
        "Vendas: "+vendas;
    }
```

```
}
```

Classe Gerente.java:

```
public class Gerente extends Funcionario {
    public void demitir(Funcionario) {
        "Aqui vai a implementação (código) para excluir um funcionário"
    }

    public String toString() {
        return "\n" +
        "Nome: "+getNome()+"\n" +
        "Cpf: "+getCpf()+"\n" +
        "Salário: "+getSalario()+"\n" +
        "\nEndereço: "+getEndereco()+"\n" +
        "\nContato: "+getContatos();
    }
}
```

Classe Produto.java:

```
public class Produto {
    private String nome;
    private String descricao;
    private double valor;
    private Venda;

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }

    public String getDescricao() {
        return descricao;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

    public double getValor() {
        return valor;
    }

    public String toString() {
        return "\n" +
        "Nome: "+nome+"\n" +
        "Descrição: "+descricao+"\n" +
        "Valor: "+valor+"\n" +;
    }
}
```

Classe Endereco.java:

```
public class Endereco {
    private String rua;
    private int numero;
    private String complemento;
    private String bairro;
    private String cep;
    private String cidade;
    private String uf;
```

```
    public void setRua(String rua) {
        this.rua = rua;
    }

    public String getRua() {
        return rua;
    }

    public void setNumero(int numero) {
        this.numero = numero;
    }

    public int getNumero() {
        return numero;
    }

    public void setComplemento(String complemento) {
        this.complemento = complemento;
    }

    public String getComplemento() {
        return complemento;
    }

    public void setBairro(String bairro) {
        this.bairro = bairro;
    }

    public String getBairro() {
        return bairro;
    }

    public void setCep(String cep) {
        this.cep = cep;
    }

    public String getCep() {
        return cep;
    }

    public void setCidade(String cidade) {
        this.cidade = cidade;
    }

    public String getCidade() {
        return cidade;
    }

    public void setUf(String uf) {
        this.uf = uf;
    }

    public String getUf() {
        return uf;
    }

    public String toString() {
        return "\n" +
        "Rua: "+rua+"\n" +
        "Número: "+numero+"\n" +
        "Complemento: "+complemento+"\n" +
        "Bairro: "+bairro+"\n" +
        "Cep: "+cep+"\n" +
        "Cidade: "+cidade+"\n" +
        "UF: "+uf;
    }
}
```

Classe Contato.java:

```
public class Contato {
    private String telefone;
    private String email;

    public void setTelefone(String telefone) {
```

```
        this.telefone = telefone;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return this.email;
    }

    public String toString() {
        return "\n"+
        "Telefone: "+telefone+"\n"+
        "E-mail: "+email;
    }
}
```

Classe Venda.java:

```
import java.util.Collection;

public class Venda {
    private long codigo;
    private String dataVenda;
    private Vendedor;
    private List<Produto> produtos;

    public void setCodigo(long codigo) {
        this.codigo = codigo;
    }

    public long getCodigo() {
        return codigo;
    }

    public void setDataVenda(String dataVenda) {
        this.dataVenda = dataVenda;
    }

    public String getDataVenda() {
        return dataVenda;
    }

    public void setVendedor(Vendedor vendedor){
        this.vendedor = vendedor;
    }

    public Vendedor getVendedor(){
        return vendedor;
    }

    public void setProdutos(List<Produto> produtos){
        this.produtos = produtos;
    }

    public List<Produto> getProdutos(){
        return produtos;
    }

    public double calcularVenda() {
        return 0;//Aqui vai a implementação (código para calcular a venda)
    }

    public String toString() {
        return "\n" +
        "Código: "+codigo+ "\n" +
        "Data de venda: "+dataVenda+ "\n" +
        "Vendedor: "+vendedor+ "\n" +
        "Produtos: "+produtos+ "\n" +;
    }
```

```
}
```