



# Programação Orientada a Objetos

# Olá!

## Eu sou Willian Brito

- ❖ Desenvolvedor Full Stack na Msystem Software
- ❖ Formado em Analise e Desenvolvimento de Sistemas.
- ❖ Pós Graduado em Segurança Cibernética.
- ❖ Certificação SYCP (Solyd Certified Pentester) v2018





*Antes de falarmos sobre  
programação **Orientada a  
Objetos** devemos falar sobre  
**paradigmas** de programação.*

# Paradigmas de Programação

Um **paradigma de programação** é um **estilo** ou **forma** de programar é o modelo no qual o desenvolvedor vai organizar as ideias e abstrair os problemas do mundo real em código e assim elaborar uma solução para este problema.

## Exemplo de paradigmas:

Funcional, Procedural, Orientada a Objetos.

# Afinal o que é programação orientada a objetos ?

Programação orientada a objetos é um paradigma de programação baseado no conceito de "objetos", que podem conter dados na forma de campos, também conhecidos como atributos, e comportamentos, também conhecidos como métodos. Uma característica de objetos é que um comportamento de um objeto pode acessar, e modificar, os atributos do objeto com o qual eles estão associados.

# Os 4 pilares da Programação Orientada a Objetos

Para entendermos exatamente do que se trata a orientação a objetos, vamos entender quais são os requerimentos de uma linguagem para ser considerada nesse paradigma. Para isso, a linguagem precisa atender a quatro tópicos bastante importantes:

- ⦿ Abstração
- ⦿ Encapsulamento
- ⦿ Herança
- ⦿ Polimorfismo

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue.

# 1.

# Abstração

**Abstração** é um dos conceitos mais importantes do paradigma orientado a objetos e também é um de seus pilares.



“

No contexto de software,  
**abstração** é uma  
**representação simplificada**  
de **algo complexo**.





O conceito de **abstração** consiste em **isolar a complexidade** do código e torná-lo **independente** para que possa ser chamado por outras partes do software sem que você necessite entender como foi implementado.

No mundo real, utilizamos abstrações o tempo todo. Tudo que não sabemos como funciona por baixo dos panos pode ser considerado uma abstração.

# Entendendo o conceito de abstração

Para exemplificar melhor, vamos tomar como exemplo a concessionária que realiza manutenções no seu carro. Você leva ele até lá com um problema e ele volta funcionando. Em suma, pouco importa os detalhes do que aconteceu durante a manutenção do seu carro, o que importa é que ele voltou funcionando.

# Classes Abstratas

As classes abstratas são as que não permitem realizar qualquer tipo de instância. São classes feitas especialmente para serem modelos para suas classes derivadas. As classes derivadas, via de regra, deverão sobrescrever os métodos para realizar a implementação dos mesmos. As classes derivadas das classes abstratas são conhecidas como classes concretas.

# Métodos Abstratos

A funcionalidade dos **métodos abstratos** que são herdados pelas classes filhas normalmente é atribuída de acordo com o objetivo ou o propósito dessas classes, eles estão presentes somente em **classes abstratas**, e são aqueles que não possuem implementação.

A sintaxe deste tipo de método é a seguinte:

```
public abstract void Gravar();
```

# Interfaces

**Interface** é um conjunto de operações externamente visíveis no contexto de uma classe.

Uma interface é o tipo mais abstrato em **orientação a objetos**, é utilizada para especificar um comportamento que as classes devem implementar. Eles são semelhantes aos contratos. As interfaces são declaradas usando a palavra-chave da interface e podem conter apenas assinatura de método e declarações constantes, uma classe pode ser implementada com múltiplas interfaces, diferente da classe abstrata que pode ter apenas uma implementação.

Uma curiosidade é que em algumas linguagens para utilizar a herança em classe abstrata é utilizado a palavra reservada **extends**, e para interfaces é utilizado a palavra **implements**.

# Aplicando o Conceito de Abstrações

As interfaces funcionam como contratos que definem o que as implementações (Classes) devem conter. Este conceito de abstração por interfaces nos leva ao princípio OCP (Princípio do Aberto/Fechado) e DIP (Princípio da Inversão de Dependência) do SOLID.

Vamos imaginar um cenário em que precisamos desenvolver um app que realize orçamentos, neste exemplo vamos desenvolver sem utilizar conceitos de abstrações. No exemplo adiante temos um método que recebe o produto e quantidade como parâmetro e adiciona em uma lista de itens.

```
class Produto
{
    private $descricao;
    private $estoque;
    private $preco;

    public function __construct($descricao, $estoque, $preco)
    {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco = $preco;
    }

    public function getDescricao()
    {
        return $this->descricao;
    }

    public function getPreco()
    {
        return $this->preco;
    }
}
```

```
class Orcamento
{
    private $itens;

    public function adiciona(Produto $item, $qtde)
    {
        $this->itens[] = [$qtde, $item];
    }

    public function calculaTotal()
    {
        $total = 0;
        foreach ($this->itens as $item)
        {
            $total += ($item[0] * $item[1]->getPreco());
        }

        return $total;
    }
}
```

# Aplicando o Conceito de Abstrações

Agora é só adicionar produtos no orçamento e chamar o método de calcular o total e finalizamos nosso app de realizar orçamentos.

```
require_once 'classes/Orcamento.php';  
require_once 'classes/Produto.php';  
  
$orc = new Orcamento;  
$orc->adiciona( new Produto('Máquina de café', 10, 299), 1);  
$orc->adiciona( new Produto('Barbeador elétrico', 10, 170), 1);  
$orc->adiciona( new Produto('Barra de chocolate', 10, 7), 3);  
  
print $orc->calculaTotal();
```



# Aplicando o Conceito de Abstrações

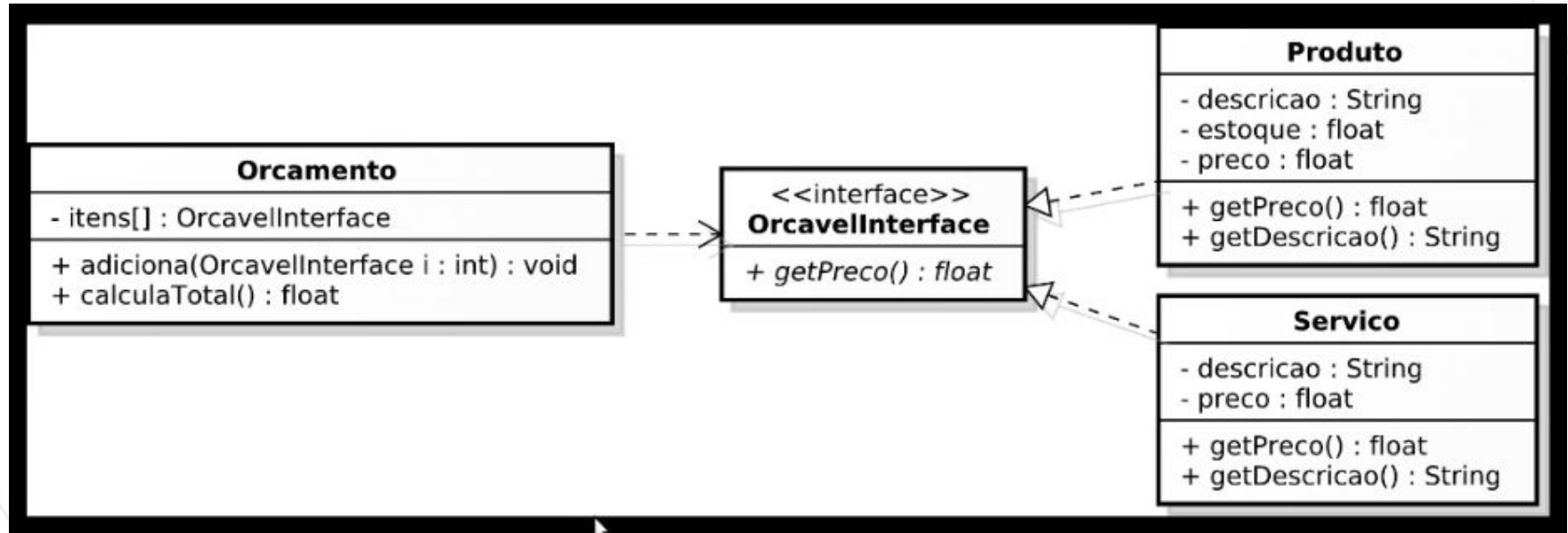
Imagine que agora nosso cliente precisa adicionar serviço e não apenas produtos no orçamento, como faríamos ?

É neste momento que utilizamos nossos conhecimentos em abstrações para tornar nosso app com **menor nível de dependência (baixo acoplamento)** e mais **flexível**.

Para fazermos isso, vamos criar uma interface chamada **OrcavelInterface** para indicar que qualquer classe que implementar esta interface pode fazer parte de um orçamento, basta implementar o método **getPreco()**.

```
interface OrcavelInterface
{
    public function getPreco();
}
```

# Aplicando o Conceito de Abstrações



# Aplicando o Conceito de Abstrações

```
class Servico implements OrcavelInterface ←
{
    private $descricao;
    private $preco;

    public function __construct($descricao, $preco)
    {
        $this->descricao = $descricao;
        $this->preco = $preco;
    }

    public function getDescricao()
    {
        return $this->descricao;
    }

    public function getPreco()
    {
        return $this->preco;
    }
}
```

```
class Produto implements OrcavelInterface ←
{
    private $descricao;
    private $estoque;
    private $preco;

    public function __construct($descricao, $estoque, $preco)
    {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco = $preco;
    }

    public function getDescricao()
    {
        return $this->descricao;
    }

    public function getPreco()
    {
        return $this->preco;
    }
}
```

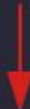
# Aplicando o Conceito de Abstrações

```
class Orcamento
{
    private $itens;

    public function adiciona(OrcavelInterface $item, $qtde)
    {
        $this->itens[] = [$qtde, $item];
    }

    public function calculaTotal()
    {
        $total = 0;
        foreach ($this->itens as $item)
        {
            $total += ($item[0] * $item[1]->getPreco());
        }

        return $total;
    }
}
```



# Aplicando o Conceito de Abstrações

Repare em como deixamos nosso app muito mais flexível utilizando abstração, agora qualquer coisa que queremos adicionar em nosso orçamento, basta criar uma classe que implemente a interface [OrcavelInterface](#) e pronto, não precisamos fazer grandes modificações em nosso app. Este simples processo resulta na possibilidade de criação de várias implementações de um mesmo contrato.

```
require_once 'classes/OrcavelInterface.php';
require_once 'classes/Orcamento.php';
require_once 'classes/Servico.php';
require_once 'classes/Produto2.php';

$orc = new Orcamento;
$orc->adiciona( new Produto('Máquina de café', 10, 299), 1);
$orc->adiciona( new Produto('Barbeador elétrico', 10, 170), 1);
$orc->adiciona( new Produto('Barra de chocolate', 10, 7), 3);

$orc->adiciona( new Servico('Conserto', 20), 1);
$orc->adiciona( new Servico('Manutenção', 30), 2);

print $orc->calculaTotal();
```

# Aplicando o Conceito de Abstrações

## ❑ Estratégias de uso de Abstrações:

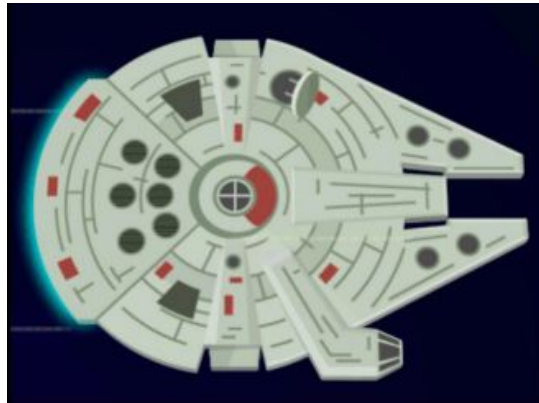
- Esconder a complexidade de uma feature e disponibilizar uma interface para consumo.
- Projetar features mais flexíveis.
- Diminuir o acoplamento entre classes.
- Utilizar classes abstratas como modelo para outras classes.

# Classe Abstrata ou Interface ?

- ❑ **Classe Abstrata:** Quando a abstração for um **conceito** ou uma **base estrutural** (algo que precisa ser refinado ou especializado).
- ❑ **Interface:** Quando a abstração for um **comportamento** (algo que uma classe deve saber fazer).

# Classe Abstrata ou Interface ?

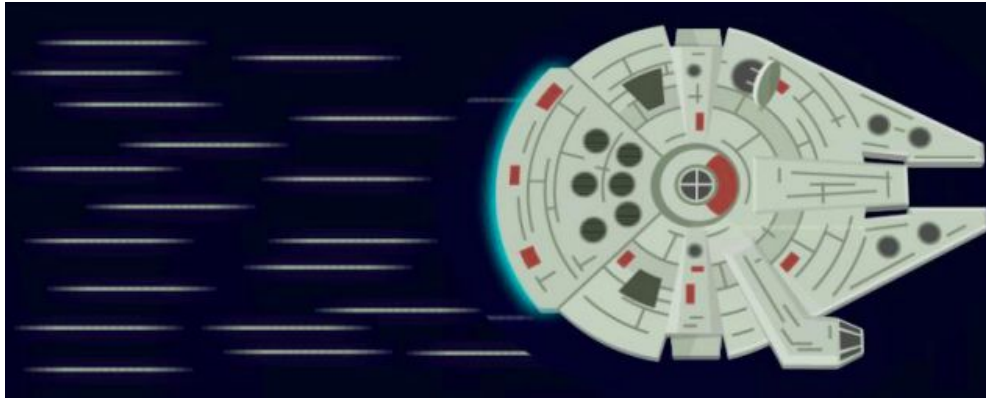
Imagine um jogo no qual existem naves que se movem. Se sua abstração representa uma nave, então você está representando um **conceito** ou uma **base estrutural** e deve utilizar uma **classe abstrata**.





# Classe Abstrata ou Interface ?

Imagine um jogo no qual existem naves que se movem. Mas, se sua abstração representa algo que se move, então o que está sendo abstraído é um **comportamento** e a melhor solução é usar uma interface.



# Conclusão

Em um cenário **orientado a objetos**, a **abstração** representa uma entidade do mundo real. Mas sua importância não para por aí, a abstração é uma peça chave não só para o cenário OO, como também, para ter um cenário desacoplado, saber utilizar abstrações, é saber colocá-las em lugares estratégicos que venham viabilizar um baixo acoplamento. Quando são utilizadas classes concretas ao invés de abstrações, você cria o que é chamado de forte acoplamento, e que em caso de mudança, causa um grande impacto e esforço para que esta mudança aconteça.

A decorative network diagram in the top-left corner, consisting of a complex web of interconnected nodes and lines, rendered in a light gray color.

# 2.

# Encapsulamento

O **Encapsulamento** serve para controlar o acesso aos atributos e métodos de uma classe.



***Encapsulamento** é o nome que damos à ideia da classe **esconder** os uma parte da implementação, ou seja, como o método faz o trabalho dela.*

# Boas práticas de encapsulamento

Sempre que um desenvolvedor ouve “código ruim”, ele logo imagina uma implementação complicada, cheia de ifs e variáveis com maus nomes. Sim, implementações ruins são problemáticas, mas quando se pensa em manter um sistema grande, a longo prazo, temos problemas maiores.

# Boas praticas de encapsulamento

Um dos principais problemas em software é justamente a propagação de alterações. Quantas vezes você, ao por a mão em um sistema legado, precisou fazer a mesma alteração em pontos diferentes? E como você fez pra achar esses pontos de alteração? Se você apelou para algum tipo de CTRL+F ou GREP, você já passou pelo problema que estou discutindo. Em sistemas mal projetados, os pontos de alteração são implícitos. O desenvolvedor precisa busca-los manualmente. E, obvio, ele vai deixar passar algum.

```
class CalculadoraDeSalario {  
    public double calcula(Funcionario funcionario) {  
        if(DESENVOLVEDOR.equals(funcionario.getCargo())) {  
            return dezOuVintePorcento(funcionario);  
        }  
  
        if(DBA.equals(funcionario.getCargo()) ||  
           TESTER.equals(funcionario.getCargo())) {  
            return quinzeOuVinteCincoPorcento(funcionario);  
        }  
  
        throw new RuntimeException("funcionario invalido");  
    }  
}
```

# Boas praticas de encapsulamento

- ⦿ Nessa implementação, DESENVOLVEDOR, DBA e TESTER são enums. Sempre que um cargo novo surgir, o desenvolvedor é obrigado a adicionar um novo item nesse enum e alterar a classe **CalculadoraDeSalario** e fazê-la suportar esse novo cargo. Mas como sabemos disso? E se tivermos outras classes similares a essa calculadora?
- ⦿ Precisamos deixar essa decisão de design mais clara. O desenvolvedor deve saber rapidamente que, ao criar um cargo novo, uma regra de cálculo deve ser associada a ele. Precisamos **encapsular** melhor todo esse problema, para que a mudança, quando feita em um único ponto, seja propagada naturalmente.

# Boas praticas de encapsulamento

- Entendendo a motivação e o problema, é fácil ver que existem muitas implementações diferentes. Aqui, optarei por fazer uso da facilidade de enums. O enum Cargo receberá no construtor a regra de calculo. Dessa forma, qualquer novo cargo deverá, obrigatoriamente, passar uma regra de calculo. Veja:

```
public enum Cargo {  
    DESENVOLVEDOR(new DezOuVintePorCento()),  
    DBA(new QuinzeOuVinteCincoPorCento()),  
    TESTER(new QuinzeOuVinteCincoPorCento());  
  
    private RegraDeCalculo regra;  
  
    Cargo(RegraDeCalculo regra) {  
        this.regra = regra;  
    }  
  
    public RegraDeCalculo getRegra() {  
        return regra;  
    }  
}
```



# Boas praticas de encapsulamento

Novamente, o principal aqui é entender o problema resolvido e não tanto a implementação. Se seu código exige que uma mudança seja feita em vários pontos diferentes para que ela seja propagada, talvez você esteja passando por um problema de projeto. Refatore seu código e encapsule esse comportamento em um único lugar. Lembre-se que o programador não deve nunca usar CTRL+F para programar e buscar pelos pontos de mudança.

# Tell, Don't Ask

Um conhecido principio de Orientação a Objetos e o Tell, Don't Ask, ou seja, “Diga, não pergunte”. Mas, como assim, diga e não pergunte? No exemplo abaixo, perceba que a primeira coisa que fazemos para o objeto e uma pergunta (ou seja, um if) e, de acordo com a resposta, damos uma ordem para esse objeto: ou calcula o valor de um jeito ou calcula de outro.

```
NotaFiscal nf = new NotaFiscal();  
double valor;  
if (nf.getValorSemImposto() > 10000) {  
    valor = 0.06 * nf.getValor();  
}  
else {  
    valor = 0.12 * nf.getValor();  
}
```

# Tell, Don't Ask

- ⦿ Quando temos códigos que perguntam uma coisa para um objeto, para então tomar uma decisão, e um código que não está seguindo o **Tell, Don't Ask**. A ideia é que devemos sempre dizer ao objeto o que ele tem que fazer, e não primeiro perguntar algo a ele, para depois decidir. O código que refatoramos faz isso direito. Perceba que estamos dando uma ordem ao objeto: calcule o valor do imposto. Lá dentro, obviamente, a implementação será o if anterior, não há como fugir disso. Mas ele está **encapsulado** no objeto:

```
NotaFiscal nf = new NotaFiscal();  
double valor = nf.calculaValorImposto();
```

# Conclusão

Esconda os detalhes da implementação, e diminua pontos de mudança. E isso que tornará seu sistema fácil de ser mantido. Lembre-se que precisamos sempre diminuir a quantidade de pontos de mudança. Quanto menos, mais fácil. Quanto mais claras, melhor. Pense no seu sistema agora como um daqueles grandes abajures, em que, se você tocar em uma das partes, as partes mais abaixo balançam. Seu sistema deve ser idêntico: se você tocar em uma classe, você precisa ver facilmente as outras classes que deverão ser alteradas. E, claro, quanto menos, melhor. No fim, não é tão difícil quanto parece.

A decorative network diagram in the top-left corner, featuring a complex web of interconnected nodes and lines, with some nodes highlighted in blue and others in grey.

# 3.

# Herança

**Herança**, permite que classes sejam organizadas em uma hierarquia que representa relacionamentos “é um”.



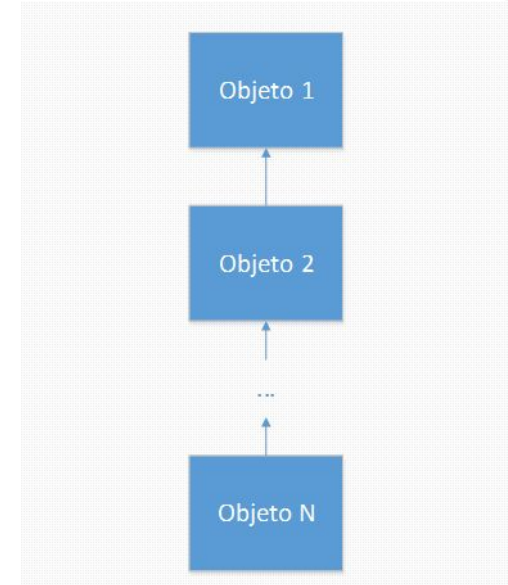
*A **herança** é um tipo de relacionamento que define que uma classe "**é uma**" de outra classe como, por exemplo, a classe Funcionário que é uma Pessoa, assim um Funcionário tem um relacionamento de herança com a classe Pessoa, ou seja a classe funcionário herda todos os **atributos** e **comportamentos** da classe pessoa.*

# Herança

O **reuso de código** é uma das grandes vantagens da programação orientada a objetos. Muito disso se dá por uma questão que é conhecida como **herança**. Essa característica otimiza a produção da aplicação em tempo e linhas de código.

# Herança

Para entendermos essa característica, vamos imaginar uma família: a criança, por exemplo, está herdando características de seus pais. Os pais, por sua vez, herdam algo dos avós, o que faz com que a criança também o faça, e assim sucessivamente. Na orientação a objetos, a questão é exatamente assim, como mostra a figura ao lado. O objeto abaixo na hierarquia irá herdar características de todos os objetos acima dele, seus “ancestrais”. A herança a partir das características do objeto mais acima é considerada herança direta, enquanto as demais são consideradas heranças indiretas. Por exemplo, na família, a criança herda diretamente do pai e indiretamente do avô e do bisavô.

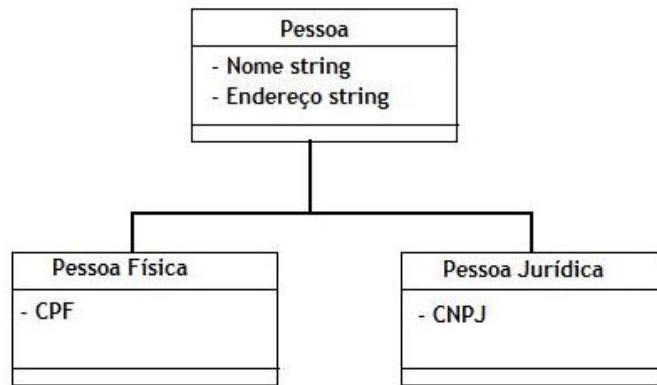




# Herança

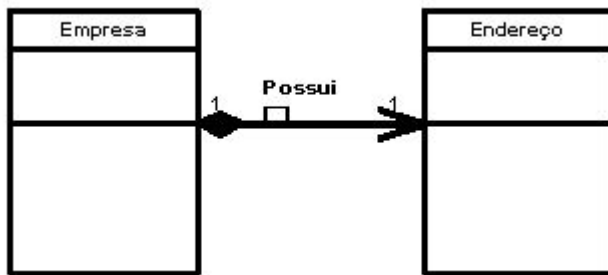
Outro exemplo clássico de herança é uma superclasse *Pessoa* com suas subclasses *PessoaFísica* e *PessoaJurídica*.

A subclasse *PessoaFísica* possui os dois atributos de sua superclasse (nome e endereço) e possui o seu atributo particular CPF. O mesmo acontece com a subclasse *PessoaJurídica*, porém sua particularidade é o CNPJ.



# Composição

Na **composição**, estendemos uma classe e delegamos o trabalho para o objeto dessa classe. Para facilitar, quando temos um sentido de que uma classe “**tem uma**” outra classe, usa-se composição (diferentemente de herança que tem o sentido de “**é uma**”). Imagine o exemplo de uma classe **Empresa** que tem um **Endereço**. Pode-se deixar a classe **Empresa** responsável pela classe **Endereço** (composição)



# Herança x Composição

## Diferença entre os comportamentos de herança e composição:

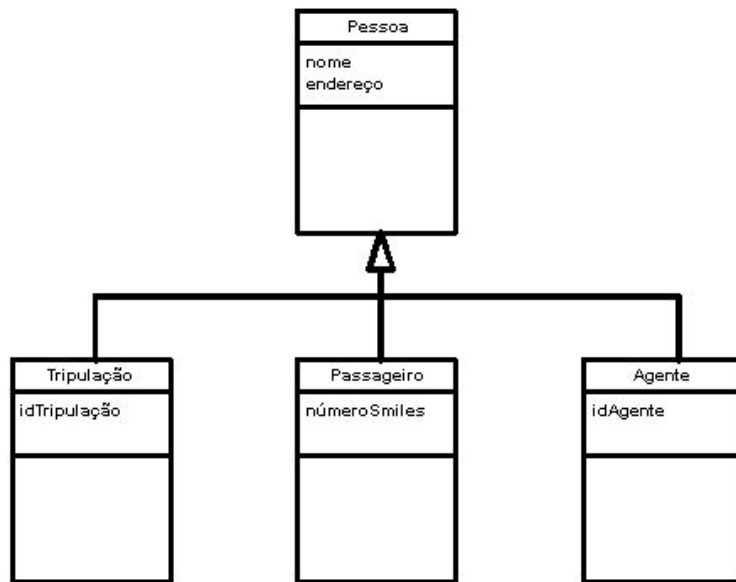
- ⦿ **Herança** herda comportamentos em tempo de compilação ou seja comportamentos herdados se tornam estáticos.
- ⦿ **Composição** você pode utilizar os comportamentos em tempo de execução e até adicionar novos comportamentos sem alterar o código existente.

# Herança x Composição

A herança tem a vantagem de capturar o que é comum e o isolar daquilo que é diferente, além de ser vista diretamente no código OO. Porém, ela gera um alto acoplamento, pois uma pequena mudança numa superclasse afeta todas as suas subclasses. Essa característica da herança viola um dos princípios da Orientação a Objetos que é manter o baixo acoplamento. Além disso, a herança é um relacionamento estático. Algumas vezes, os objetos do mundo OO precisam sofrer mutações, e algumas delas não são possíveis quando utilizamos herança (por exemplo, se um funcionário mudou de cargo dentro de uma empresa, seu objeto deveria mudar de classe, o que não ocorre).

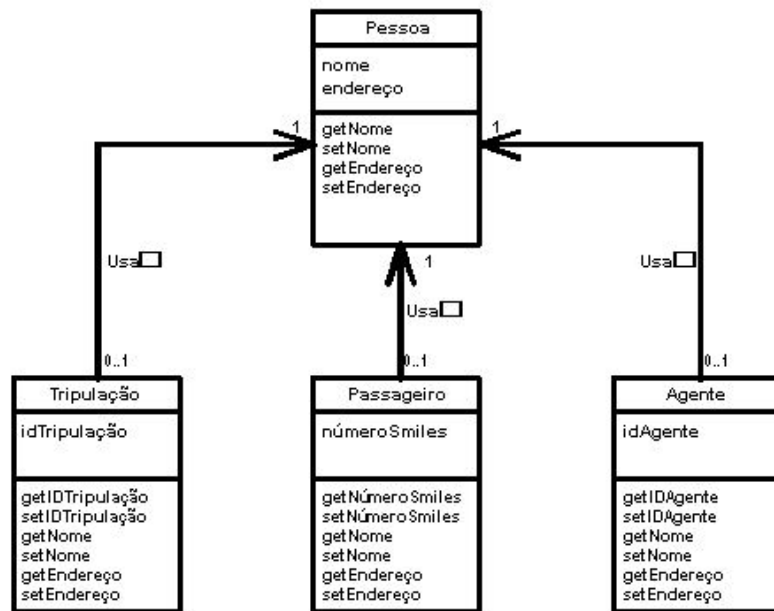
# Herança x Composição

Considere o exemplo a seguir. Nele, por usar herança, se um **Agente** for viajar, ele será um **Passageiro**, mas, com tal disposição das classes, realizar a mudança de classe do objeto se torna uma tarefa complicada.



# Herança x Composição

Agora, observe o mesmo exemplo usando composição. Note que ficou muito mais fácil permitir que uma **Pessoa** possua vários papéis. Usando **composição**, nós estendemos a funcionalidade de **Pessoa**, sem usar **herança**. Com esse mecanismo, o comportamento de um objeto pode ser definido em tempo de execução (não ficando preso ao tempo de compilação).



# Conclusão

É possível perceber que existem várias diferenças entre **Herança** e **Composição**. Enquanto herança é mais fácil de entender, permite um maior reuso de código e isola o comportamento comum do comportamento diferenciado, já a composição permite um maior dinamismo dos objetos do programa, permitindo que objetos tenham comportamentos diferentes ao longo de sua vida. Mas, como nem tudo são flores, cada uma tem suas desvantagens. A herança permite um forte acoplamento (o que viola os princípios OO). Já a composição, apesar de todas as suas vantagens já descritas, dificulta o entendimento por parte dos programadores, uma vez que é um código dinâmico e parametrizado.

Dessa forma, cabe ao programador OO avaliar, de acordo com os requisitos de seu projeto, se é melhor utilizar herança ou composição.

A decorative network diagram in the top-left corner, consisting of a complex web of interconnected nodes and lines, rendered in a light gray color.

# 4.

# Polimorfismo

O **Polimorfismo** é um mecanismo por meio do qual selecionamos as funcionalidades utilizadas de forma dinâmica por um programa no decorrer de sua execução.





***Polimorfismo** significa ter “**muitas formas**”, que significa um único nome representando um código diferente, selecionado por algum mecanismo automático, ou seja “**Um nome, vários comportamentos**”.*

# Entendendo o conceito de polimorfismo

Podemos assumir que uma bola de futebol e uma camisa da seleção brasileira são artigos esportivos, mais que o cálculo deles em uma venda é calculado de formas diferentes.

# Entendendo o conceito de polimorfismo

Outro exemplo: podemos dizer que uma classe chamada **Vendedor** e outra chamada **Diretor** podem ter como base uma classe chamada **Pessoa**, com um método chamado **CalcularVendas**. Se este método (definido na classe base) se comportar de maneira diferente para as chamadas feitas a partir de uma instância de **Vendedor** e para as chamadas feitas a partir de uma instância de **Diretor**, ele será considerado um método polimórfico, ou seja, um método de várias formas.

# Aplicando o polimorfismo

Assim podemos ter na **classe base** o método **CalcularVendas**:

```
public decimal CalcularVendas()
{
    decimal valorUnitario = decimal.MinValue;

    decimal produtosVendidos = decimal.MinValue;

    return valorUnitario * produtosVendidos;
}
```

# Aplicando o polimorfismo

Na classe `Vendedor` temos o mesmo método, mais com a codificação diferente:

```
public decimal CalcularVendas()

{

    decimal valorUnitario = 50;

    decimal produtosVendidos = 1500;

    return valorUnitario * produtosVendidos;

}
```

# Aplicando o polimorfismo

Na classe **Diretor** acrescentamos ao calculo uma **taxa adicional**:

```
public decimal CalcularVendas()

{

    decimal valorUnitario = 150;

    decimal produtosVendidos = 3800;

    decimal taxaAdicional = 100;

    return taxaAdicional + (valorUnitario * produtosVendidos);

}
```



# Conclusão

É importante observar que, quando **polimorfismo** está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. O polimorfismo também é usado em uma série de **refatorações**, como substituir condicional por polimorfismo para deixar o código mais limpo.

A decorative network diagram in the top-left corner, consisting of a complex web of interconnected nodes and lines, rendered in a light gray color.

# 5.

## Outros Conceitos

Outros conceitos muito importantes na programação orientada a objetos que não podemos deixar de falar.



# Outros Conceitos

- © **Classes:** As classes são abstrações de algum elemento do mundo real, que servira de base para a construção de um objeto aonde está classe têm características e comportamentos que permite armazenar propriedades e métodos dentro dela. Exemplos de classes: uma pessoa, um lugar, algo que seja “abstrato”.
- © **Atributos:** Os atributos são as propriedades de um objeto, também são conhecidos como variáveis ou campos. Essas propriedades definem o estado de um objeto, fazendo com que esses valores possam sofrer alterações.
- © **Objetos:** Os objetos são características definidas pelas classes. Neles é permitido instanciar objetos da classe para inicializar os atributos e invocar os métodos.

# Outros Conceitos

- ◎ **Visibilidade:** São tipos de acesso aos atributos e métodos de uma classe.
  - **public:** permite que sejam acessados diretamente de qualquer classe;
  - **private:** permite que sejam acessados apenas dentro da classe;
  - **protected:** permite sejam acessados apenas dentro da própria classe e em classes filhas;
- ◎ **Construtores:** O construtor de um objeto é um método especial, pois inicializa seus atributos toda vez que é instanciado (inicializado). Toda vez que é digitada a palavra reservada **new**, o objeto solicita para a memória do sistema armazená-lo, onde chama o construtor da classe para inicializar o objeto. A identificação de um construtor em uma classe é um método com o mesmo nome da classe.

# Outros Conceitos

- © **Módulos ou Pacotes:** Módulos ou Pacotes de Software é o termo utilizado para descrever o elemento de software que encapsula uma série de funcionalidades. Um Módulo ou Pacote é uma unidade independente, que pode ser utilizado com outros Módulos ou Pacotes para formar um sistema mais complexo.
- © **Getters and Setters:** Get e Set, são métodos que classes públicas, deve fornecer aos campos privados, eles servem para acessar os dados (**get - Acesso**) da classe e caso a classe seja mutável é utilizado os métodos (**set - modificadores**) que serve para atribuir um valor ao atributo da classe.

# Outros Conceitos

- ◎ **Delegates:** Na linguagem C# um delegate é um elemento que permite que você faça referência a um método. Então um delegate é semelhante a um ponteiro de função (a vantagem é que é um ponteiro seguro). Usando um delegate você pode encapsular a referência a um método dentro de um objeto de delegação. Usando delegates você tem a flexibilidade para implementar qualquer funcionalidade em tempo de execução.
  
- ◎ Existem três etapas na definição e uso de delegates :
  1. Declaração
  2. Instanciação
  3. Invocação

# Outros Conceitos

```
using System;

namespace Macoratti.SimplesDelegate
{
    // Declaração
    public delegate void SimplesDelegate();

    class ExemploDeDelegate
    {
        public static void minhaFuncao()
        {
            Console.WriteLine("Eu fui chamada por um delegate ...");
        }

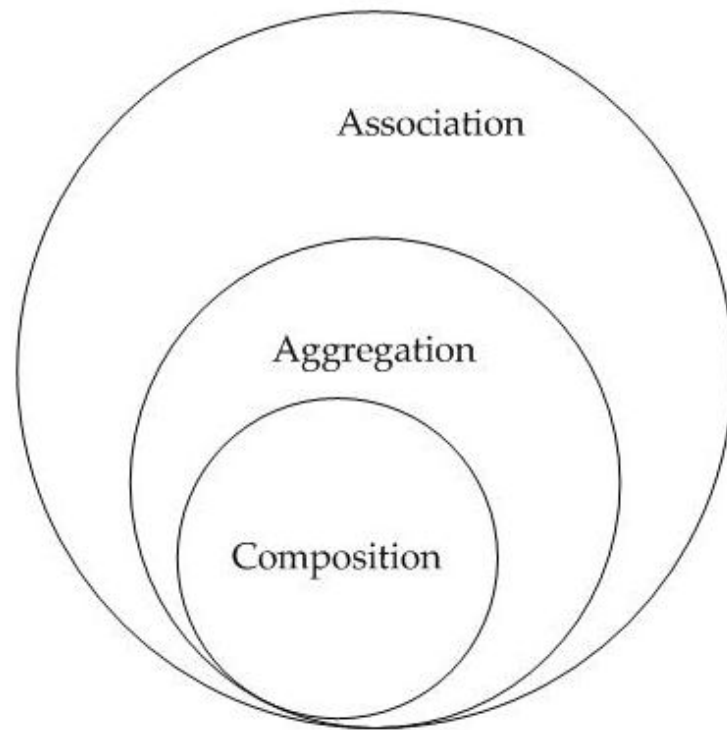
        public static void Main()
        {
            // Instanciação
            SimplesDelegate simplesDelegate = new SimplesDelegate(minhaFuncao);

            // Invocação
            simplesDelegate();
            Console.ReadKey();
        }
    }
}
```

# Relacionamento entre objetos

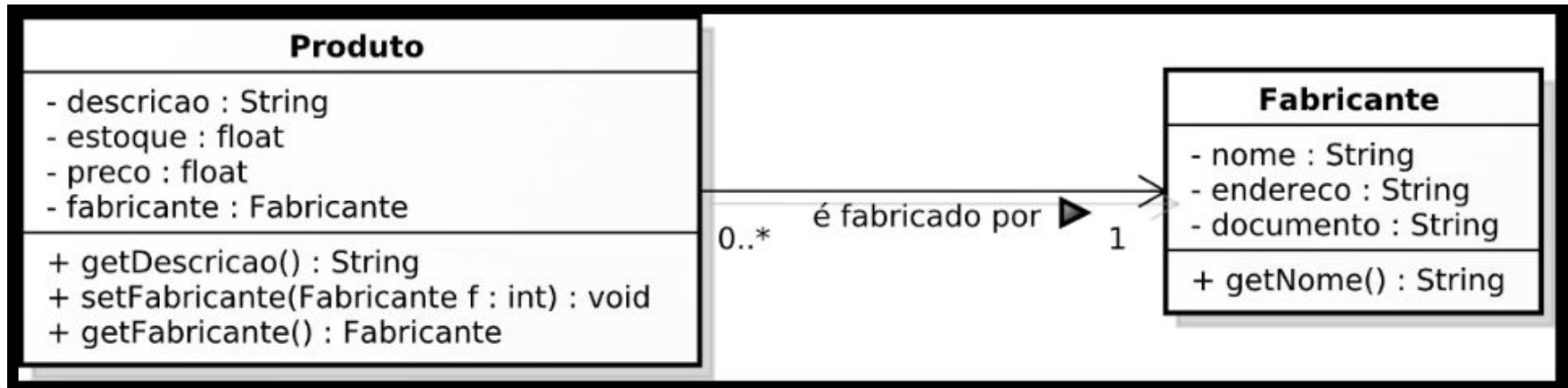
O **relacionamento entre objetos** define como os objetos vão interagir para executar uma operação em uma aplicação.

- ⦿ **Composição** e **Agregação** são as duas formas de **Associação**



# Associação

A **Associação** acontece quando um objeto possui um **ponteiro** ou **referência** para outro objeto. OBS: Composição e agregação são as duas formas de associação.



```
class Produto
{
    private $descricao;
    private $estoque;
    private $preco;
    private Fabricante $fabricante;

    public function __construct($descricao, $estoque, $preco)
    {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco = $preco;
    }

    public function getDescricao()
    {
        return $this->descricao;
    }

    public function setFabricante( Fabricante $fabricante )
    {
        $this->fabricante = $fabricante;
    }

    public function getFabricante()
    {
        return $this->fabricante;
    }

    public function getPreco()
    {
        return $this->preco;
    }
}
```

```
class Fabricante
{
    private $nome;
    private $endereco;
    private $documento;

    public function __construct($nome, $endereco, $documento)
    {
        $this->nome = $nome;
        $this->endereco = $endereco;
        $this->documento = $documento;
    }

    public function getNome()
    {
        return $this->nome;
    }
}
```

```
require_once 'classes/Fabricante.php';
require_once 'classes/Produto.php';

$p1 = new Produto('Chocolate', 10, 7);
$f1 = new Fabricante('Fabrica de chocolate', 'Rua tal...', '93.3823393.333');

$p1->setFabricante($f1);

$descricao = $p1->getDescricao();
$nome_fabr = $p1->getFabricante()->getNome();

print "O Fabricante do produto {$descricao} é {$nome_fabr}";
```



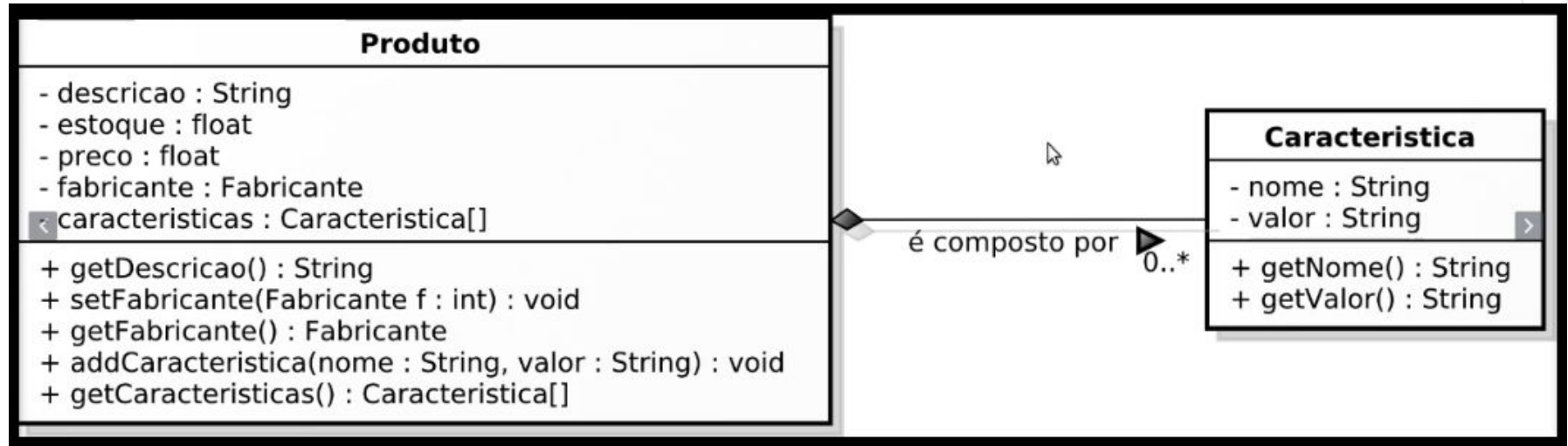
# Composição

**Composição** é um tipo específico de **Associação**, porém é um relacionamento chamado de **todo-parte**, a ideia deste relacionamento é de termos um objeto maior composto por partes menores, onde teremos um objeto principal que é o **todo**, que são compostos por objetos menores que é a **parte**.

## ❏ Características:

- Os objetos **parte** são **criados** dentro do objeto **todo**.
- Os objetos **parte** só existem dentro do objeto **todo** e nunca estará disponível sozinho no restante do sistema.
- O objeto **todo** é responsável pelo ciclo de vida do objeto **parte**, ou seja, construção e destruição do objeto.
- No contexto de banco de dados é um relacionamento **1** para **N**, onde teremos a tabela **todo** e a tabela **parte**.

# Composição



```
class Produto
{
    private $descricao;
    private $estoque;
    private $preco;
    private Caracteristica $caracteristicas;

    public function __construct($descricao, $estoque, $preco)
    {
        $this->descricao = $descricao;
        $this->estoque = $estoque;
        $this->preco = $preco;
        $this->caracteristicas = [];
    }

    public function addCaracteristica( $nome, $valor )
    {
        $this->caracteristicas[] = new Caracteristica( $nome, $valor);
    }

    public function getCaracteristicas()
    {
        return $this->caracteristicas;
    }

    public function getDescricao()
    {
        return $this->descricao;
    }

    public function getPreco()
    {
        return $this->preco;
    }
}
```

```
class Caracteristica
{
    private $nome;
    private $valor;

    public function __construct( $nome, $valor )
    {
        $this->nome = $nome;
        $this->valor = $valor;
    }

    public function getNome()
    {
        return $this->nome;
    }

    public function getValor()
    {
        return $this->valor;
    }
}
```

# Composição

```
require_once 'classes/Produto.php';
require_once 'classes/Caracteristica.php';

$p1 = new Produto('Chocolate', 10, 7);
$p1->addCaracteristica( 'Cor', 'Branco');
$p1->addCaracteristica( 'Peso', '500gr');

print 'Produto: ' . $p1->getDescricao() . '<br>';
foreach ($p1->getCaracteristicas() as $caracteristica)
{
    $nome = $caracteristica->getNome();
    $valor = $caracteristica->getValor();

    print "Característica {$nome} = {$valor} <br>";
}
```

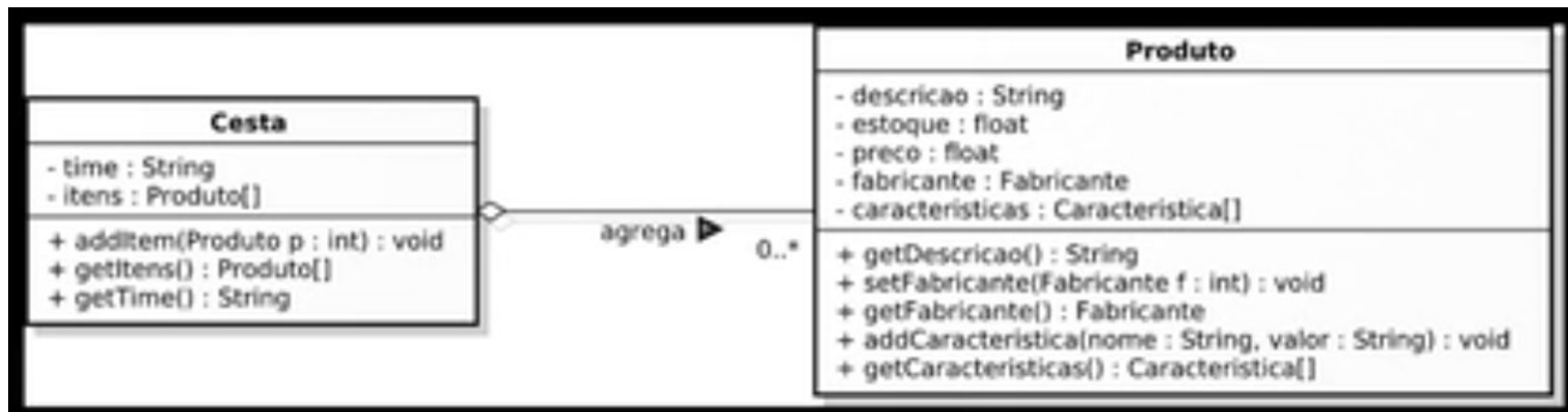
# Agregação

**Agregação** também é um tipo específico de **Associação** que também é considerado um relacionamento de **todo-parte**, porém a diferença entre a composição é que o objeto todo e parte são criados de forma separada (diferente da composição) e depois são reunidos.

## ❏ Características:

- Os objetos **parte** são **injetados** dentro do objeto **todo**.
- Os objetos **parte** estão disponíveis fora do objeto **todo**.
- Os objetos **parte**, pode estar referenciado em vários objetos **todo**.
- Ao deletar o objeto **todo** o objeto **parte**, não será excluído.
- No contexto de banco de dados é um relacionamento **N** para **N**, onde existe a tabela **todo**, **parte** e uma tabela no **meio** de relacionamento entre o todo e parte.

# Agregação



```
class Cesta
{
    private $hora;
    private Produto $itens;

    public function __construct()
    {
        $this->hora = date('Y-m-d H:i:s');
        $this->itens = [];
    }

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

    public function getItens()
    {
        return $this->itens;
    }
}
```

```
require_once 'classes/Cesta.php';
require_once 'classes/Produto.php';
```

```
$c1 = new Cesta;
$p1 = new Produto('Chocolate', 10, 5);
$p2 = new Produto('Café', 100, 7);
$p3 = new Produto('Mostarda', 50, 3);
```

```
$c1->addItem( $p1 );
$c1->addItem( $p2 );
$c1->addItem( $p3 );
```

```
foreach ($c1->getItens() as $item)
{
    print "Item: {$item->getDescricao()} <br>";
}
```



# Outros Conceitos

## 🎯 Diferença entre procedimento, função e métodos:

1. **Procedimento:** Sequência de instruções específica de um programa, seja com intuito ou não de retornar algum valor, que pode ser invocada a partir de outros locais.
2. **Método:** Procedimento ou função que pertencente a uma classe.
3. **Função:** É um procedimento que obrigatoriamente retorna um valor.



# Conclusão

A programação orientada a objetos é um paradigma predominante no desenvolvimento de software. Quase todas as linguagens de programação implementam o conceito. Entre suas características, permite abstrair os conceitos do mundo real e representar os mesmos através de classes, atributos, métodos e relacionamentos. Outra característica da POO, é que os objetos são autônomos. Têm seu comportamento encapsulado isolando as regras de negócio dentro da classe. Isso melhora a manutenibilidade, a legibilidade e a qualidade do software.



# Obrigado!

## Alguma Pergunta?

Você pode me encontrar em:

- 🕒 [willianbrito05@gmail.com](mailto:willianbrito05@gmail.com)
  - 🕒 [www.linkedin.com/in/willian-ferreira-brito](https://www.linkedin.com/in/willian-ferreira-brito)
  - 🕒 [github.com/Willian-Brito](https://github.com/Willian-Brito)
- 