HERANÇA x COMPOSIÇÃO

Introdução

Herança e composição são dois conceitos importantes na programação orientada a objetos. A herança permite que uma classe filha herde propriedades e métodos de uma classe pai. A classe filha pode adicionar ou modificar comportamentos, mas também pode acessar os comportamentos da classe pai. A herança é usada para estender as funcionalidades da classe pai e reutilizar o código.

Já a composição é outra maneira de construir classes. Neste caso, uma classe é composta por outras classes e não há uma relação hierárquica entre elas. Em vez de herdar propriedades e métodos, a classe composta cria instâncias de outras classes para usar suas funcionalidades. A composição é usada para construir objetos complexos com funcionalidades específicas.

Exemplo de código em Java mostrando o conceito de herança.

Nesse exemplo, a classe `Cahorro` herda da classe `Animal`. Isso significa que a classe `Cahorro` recebe todos os métodos e propriedades da classe `Animal`, além de ter seus próprios métodos e propriedades. Aqui, o método `comer()` é definido na classe `Animal` e é herdado pela classe `Cahorro` . A classe `Cahorro` também define seu próprio método `latir()`.

```
class Animal {
   public void comer() {
      System.out.println("O animal está comendo.");
   }
}
```

```
class Cachorro extends Animal {
   public void latir() {
      System.out.println("O cachorro está latindo.");
   }
}

public class Main {
   public static void main(String[] args) {
      Cachorro toto = new Cachorro ();
      toto.comer();
      toto.latir();
   }
}
```

Exemplo de código em Java mostrando o conceito de composição.

Nesse exemplo, a classe `Carro` é composta pela classe `Motor`. Em vez de herdar da classe `Motor`, a classe `Carro` tem uma instância da classe `Motor` como uma propriedade privada. A classe `Carro` define seu próprio método `start()`, que chama o método `start()` da instância da classe `Motor`. Aqui, a classe `Motor` é encapsulada dentro da classe `Carro` .

```
class Motor {
   public void start() {
      System.out.println("O motor está ligado.");
   }
}

class Carro {
   private Motor motor;
```

```
    public Carro() {
       this.motor = new Motor ();
    }

    public void start() {
       this.motor .start();
    }
}

public class Main {
    public static void main(String[] args) {
       Carro carro = new Carro ();
       carro.start();
    }
}
```

Tanto a herança quanto a composição são estratégias importantes em Programação Orientada a Objetos (OOP), cada uma com suas próprias vantagens e desvantagens. Vamos analisar cada uma:

Herança:

Vantagens:

1. Reutilização de código: Permite que classes filhas herdem atributos e métodos da classe pai.
2. Hierarquia clara: Cria uma estrutura hierárquica bem definida entre classes.
3. Polimorfismo: Facilita o uso de polimorfismo, permitindo que objetos de classes filhas sejam tratados como objetos da classe pai.

Desvantagens:

1. Acoplamento forte: As classes filhas ficam fortemente acopladas à classe pai.
2. Fragilidade: Mudanças na classe pai podem afetar todas as classes filhas.
3. Hierarquia rígida: Pode levar a hierarquias complexas e difíceis de manter.

Composição:

Vantagens:

1. Flexibilidade: Permite combinar comportamentos de várias classes de forma mais flexível.
2. Acoplamento fraco: As classes são menos dependentes umas das outras.
3. Manutenção mais fácil: Mudanças em uma classe têm menos impacto em outras.
4. Favorece a reutilização: Encoraja a criação de componentes menores e mais reutilizáveis.

Desvantagens:

1. Mais código: Pode requerer mais código para implementar funcionalidades complexas.
2. Menos intuitivo: A estrutura do código pode ser menos óbvia à primeira vista.

Estratégia recomendada:

A escolha entre herança e composição depende do contexto específico do problema. No entanto, há uma tendência na comunidade de desenvolvimento para favorecer a composição sobre a herança, seguindo o princípio "Prefira composição sobre herança".

Uma boa estratégia é:

1. Use herança quando houver uma relação clara "é um" entre as classes (por exemplo, "Gato é um Animal").
2. Opte por composição quando a relação for mais do tipo "tem um" (por exemplo, "Carro tem um Motor").
3. Considere usar interfaces com composição para obter flexibilidade e polimorfismo sem os problemas de acoplamento forte da herança.
4. Utilize herança com cuidado, especialmente em hierarquias profundas.
5. Aproveite os benefícios da composição para criar sistemas mais modulares e flexíveis.

Ambas as estratégias têm seu lugar, mas a composição geralmente oferece maior flexibilidade e menor acoplamento, tornando-a frequentemente a escolha preferida em designs de software modernos.