METODOLOGIA E LINGUAGEM DE PROGRAMAÇÃO AVANÇADA

Herança vs Composição

Ciências da Computação

Material Baseado nas Notas do Prof. Ms. Daniel Brandão

Prof. Esp. Renato Leite

renato.leite@unipe.br

OBJETIVOS

- Conceito Herança;
- Conceito Composição;
- Distinção entre as duas abordagens;

CONCEITO HERANÇA

- A herança é um mecanismo da Orientação a Objeto que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida.

- Este mecanismo é muito interessante, pois promove um grande reuso e reaproveitamento de código existente.

CONCEITO COMPOSIÇÃO

- Use composição para estender as responsabilidades pela delegação de trabalho a outros objetos:

- Em vez de codificar um comportamento estaticamente, definimos e encapsulamos pequenos comportamentos padrão e usamos composição para delegar comportamentos

COMPOSIÇÃO VS HERANÇA

- Composição e herança são dois mecanismos para reutilizar funcionalidade;

- A herança foi considerada a ferramenta básica de extensão e reuso de funcionalidade por muito tempo;

- A composição estende uma classe pela **delegação** de trabalho para outro objeto;

COMPOSIÇÃO VS HERANÇA

- A herança estende atributos e métodos de uma classe

- A composição é muito superior à herança na maioria dos casos, **por quê?**

- A herança deve ser utilizada em alguns (poucos) contextos

COMPOSIÇÃO

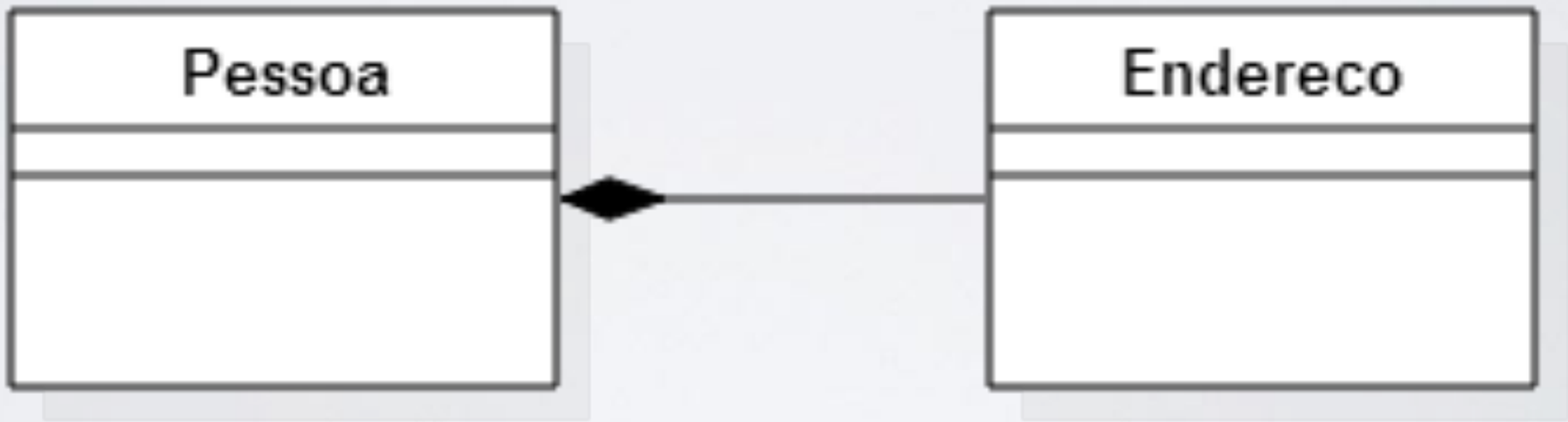
- Use composição para estender as responsabilidades pela delegação de trabalho a outros objetos:

- O objeto Pessoa « tem » um objeto Endereco

- Podemos deixar o objeto Pessoa responsável pelo objeto Endereco;

- Podemos deixar o objeto Pessoa responsável pelo objeto Endereco e temos composição

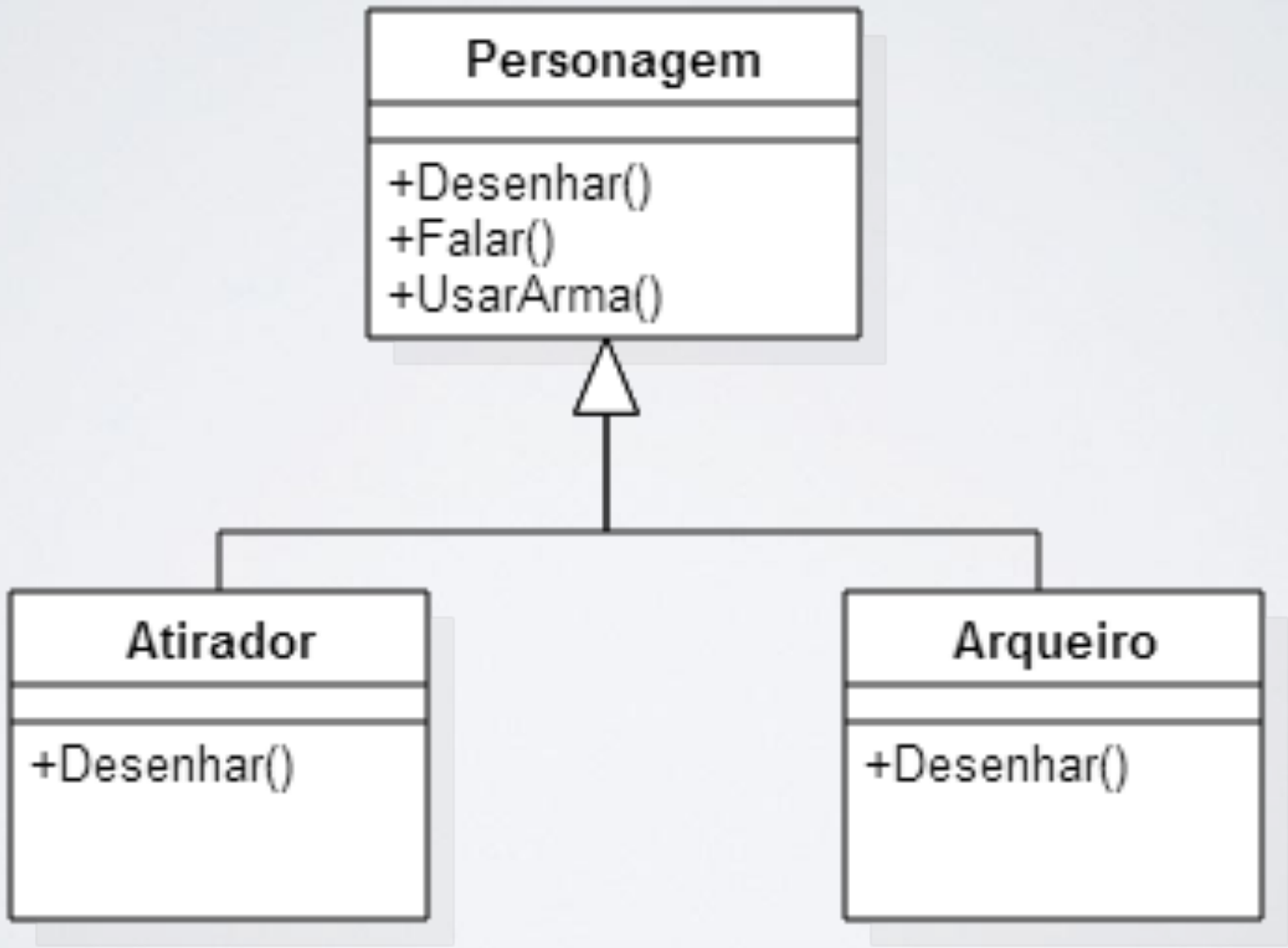
COMPOSIÇÃO - UML



SITUAÇÃO PROBLEMA

- Imagine a modelagem de um sistema para um jogo de guerra para computador com personagens:
- Existem dois tipos de personagens
- Atirador
- Arqueiro

SITUAÇÃO PROBLEMA



SITUAÇÃO PROBLEMA

- Como deveria ser implementada a solução do problema usando herança?

SITUAÇÃO PROBLEMA

```java
public abstract class Personagem {

    public abstract void desenhar();
    public void falar(){
        /*código comum para falar*/
    }
    public void usarArma(){
        /*código comum para usar arma*/
    }

}
```

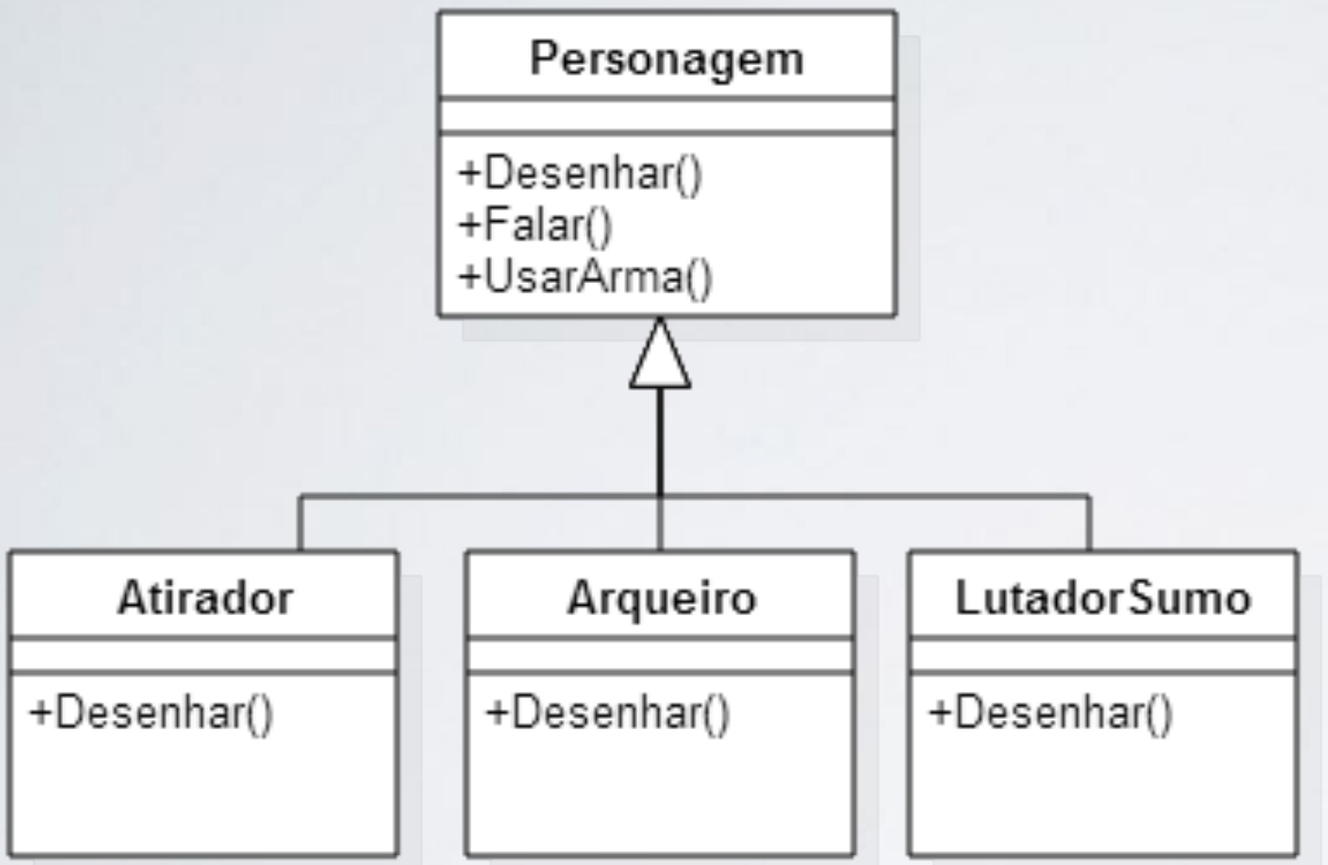
```java
public class Atirador extends Personagem{

    @Override
    public void desenhar() {
        /*Desenha o atirador*/
    }

}
```

```java
public class Arqueiro extends Personagem{

    @Override
    public void desenhar() {
        /*Desenha o atirador*/
    }

}
```

PROBLEMAS?

- Encapsulamento entre classes e subclasses forte (forte acoplamento)
- Mudanças na superclasse podem afetar todas a subclasses;
- Falta flexibilidade pra mudanças! Não delega de forma total.
- Todo lutador tem armas?!
- Um lutador só utiliza 1 tipo de arma?

PROBLEMAS?




```java
public class LutadorSumo extends Personagem{

    @Override
    public void desenhar() {
        /*Desenha o lutador de sumô*/
    }
    @Override
    public void usarArma(){
        /* Sem arma?*/
    }

}
```

```java
public class Atirador extends Personagem{

    @Override
    public void desenhar() {
        /*Desenha o atirador*/
    }

    @Override
    public void usarArma(){
        System.out.println("Disparo");
    }

}
```

ESTRATÉGIA ERRADA = GAMBIARRA!

```java
public class Atirador extends Personagem{

    @Override
    public void desenhar() {
        /*Desenha o atirador*/
    }

    @Override
    public void usarArma(){
        System.out.println("Disparo");
    }

}
```

E se… novas armas surgirem?

```java
public void usarArma(int arma){
    if(arma == 0){
        System.out.println("Disparo");
    } else {
        System.out.println("Rajada");
    }
}
```

CONCLUSÃO

- Não teremos sucesso nestes cenários do Jogo com herança. Por quê?
- No primeiro cenário, existem comportamentos na superclasse que não são comuns a todos os personagens do jogo
- No segundo cenário, com uma solução de contorno, o código da arma específica está "preso" em cada uma das classes. Isso dificulta a criação de novas armas para o jogo e não permite que um personagem mude de arma em tempo de execução.
- O que fazer?

SOLUÇÃO

- Separar as partes que podem mudar das partes que não mudam;
- Descobrindo o que irá mudar, a idéia é encapsular isso, trabalhar com uma interface e usar o códigos sem a preocupação de ter que reescrever tudo, caso surjam versões futuras (fraco acoplamento.);
- Programe para uma interface, e não para uma implementação (concreta);
- Explorar o polimorfismo e a ligação dinâmica para usar um supertipo e poder trocar objetos distintos em tempo de execução

RESULTADO

- Permitir o aparecimento de novas armas no Jogo sem grandes impactos negativos (if-else's) em todas as subclasses concretas
- Permitir que um personagem mude de arma em tempo de execução
- Em vez de já definir a arma estaticamente no código
- Alguns comportamentos não devem ser compartilhados por todos os personagens
- Ex.: atirar com um revólver ser executado por um objeto do tipo lutador de Sumô!!

IMPLEMENTAÇÃO

- Se iremos programar o comportamento dos personagens dinamicamente, definiremos o que ele irá usar e não COMO irá usar;

- O como faz parte do modo como a arma funciona, e não do uso que o personagem dará a ela;

- Ele passa a TER uma arma e não definir o que uma arma faz;

IMPLEMENTAÇÃO

```java
public interface Arma {

    public void usarArma();
}
```

```java
public class Desarmado implements Arma{

    @Override
    public void usarArma() {
        System.out.println("Sem arma!");
    }
}
```

```java
public class Revolver implements Arma{

    @Override
    public void usarArma() {
        System.out.println("Disparo!");
    }
}
```

```java
public class Fuzil implements Arma{

    @Override
    public void usarArma() {
        System.out.println("Rajada!");
    }
}
```

```java
public class Arco implements Arma{

    @Override
    public void usarArma() {
        System.out.println("Flexada!");
    }
}
```

IMPLEMENTAÇÃO

```
public abstract class Personagem {

    private Arma arma;

    public abstract void desenhar();

    public void falar(){
        /*código comum para falar*/
    }
    public Arma getArma(){
        return this.arma;
    }

    public void setArma(Arma arma){
        this.arma = arma;
    }

}
```

OBJETIVO

- Separar o que muda do que não muda (e encapsular estes pequenos comportamentos)
- Trabalhar com uma interface para manipular estes pequenos comportamentos
- Trocar comportamentos dinamicamente
- Programar para uma interface sempre que possível
- Garante um fraco acoplamento

OBJETIVO

- Quando usar composição ou herança:
    - Identifique os componentes do objeto, suas partes;
    - Essas partes devem ser agregadas ao objeto via composição (relacionamento do tipo "É PARTE DE");
    - Classifique seu objeto e tente encontrar uma semelhança de identidade com classes existentes;
    - Herança só deve ser usada se você puder comparar seu objeto A com outro B dizendo que A "É UM tipo de..." B;

EXERCÍCIO

- Utilize a estratégia de composição para implementação de um cenário de criação de automóveis de três tipos distintos: terrestre, aéreo e fluvial;

- Atenção: todos eles são automóveis e tem necessidades em comum, mas possuem comportamentos distintos;

BIBLIOGRAFIA

- DEITEL, H.; DEITEL, P. **Java: como programar. 6. ed**. São Paulo: Pearson Prentice Hall, 2005. (Disponível na Biblioteca Digital Pearson) - https://centrodeinformacao.unipe.br/bibliotecas-digitais/livros

Java: como programar - 6ª edição

DÚVIDAS?

- Se apresentem: qual seu nome? Quais suas expectativas pra disciplina?