HERANÇA VS COMPOSIÇÃO

27/10/15

Francisco Barretto – francisco.barretto@udf.edu.br

Reuso de Código

- Mecanismo de Reuso:
  - Diminui a necessidade de re-escrever código: menos trabalho para o programador;
  - Permite o aproveitamento de código pré-existente (possivelmente livre de erros e otimizado): menos chances de cometer erros;

- Em linguagens orientadas a objeto permite a criação de um banco (biblioteca) de classes reutilizáveis.

Reuso de Código em POO

- Reuso de Classes em POO:
  - Herança no mundo da POO, o termo é associado com uma das formas de reutilização de software. Através da herança, novas classes podem ser derivadas das classes existentes. A nova classe herda propriedades e métodos da classe base. A nova classe também pode adicionar suas próprias propriedades e métodos;
  - Composição é uma maneira alternativa de estender a funcionalidade de uma classe agregando à ela, funcionalidades de outras classes.

Reuso de Código em Java

- Quando você precisa de uma clase em Java, podemos escolher entre:
  - Uma classe Java, já desenvolvida, que realize a função específica (API, classe nossa, colega, etc);
  - Escrever a classe “do zero”;
  - Reutilizar uma classe existente ou estrutura (hierarquia) de classes através de Herança;
  - Reutilizar uma classe existente com Composição.

Herança

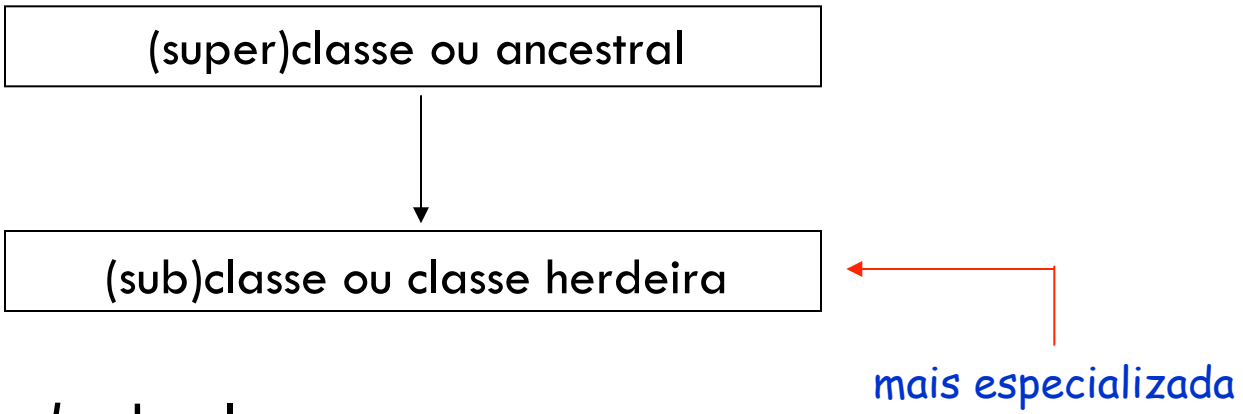
Reuso com Herança em Java

Herança

- Permite reutilizar as características de uma classe (superclasse) na definição de outra classe (subclasse);
  - Classe mais generalizada: superclasse;
  - Classe mais especializada: subclasse;
- Classes ligadas à uma hierarquia.

Herança

- Relacionamento hierárquico entre classes:



- A subclasse *herda* da classe:
  - todos os atributos
  - todos os métodos
- A subclasse pode conter atributos e métodos adicionais

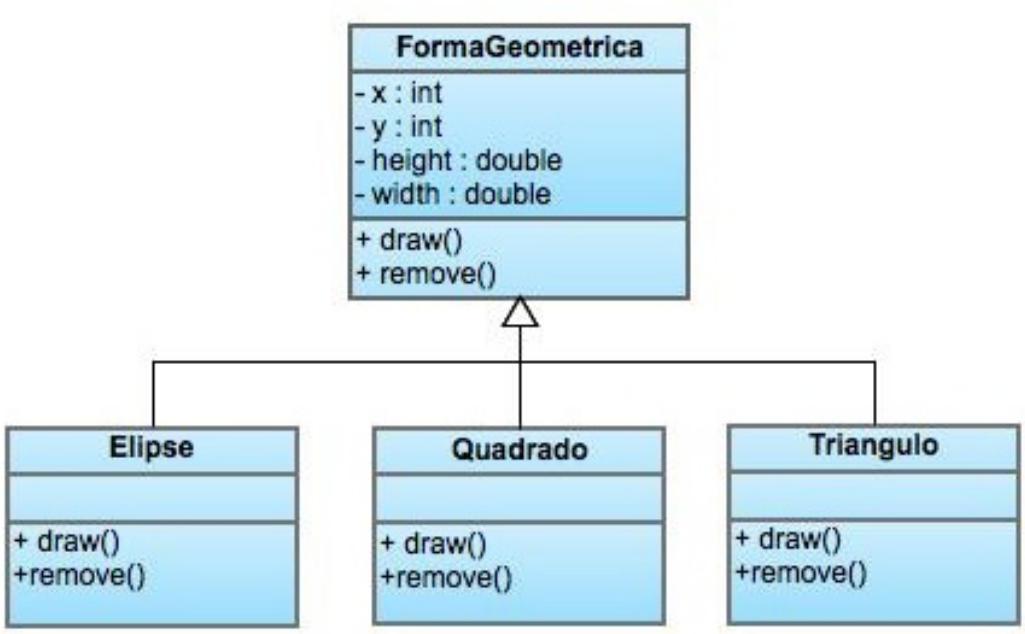
Herança

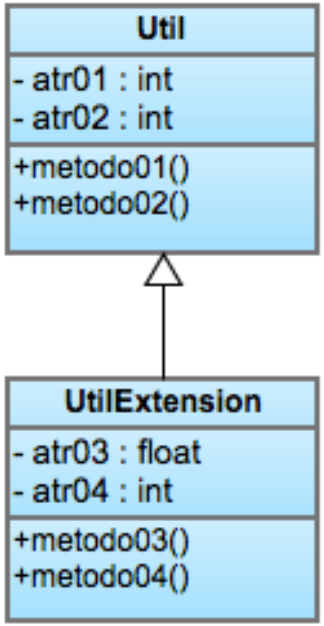
- Propriedades, conexões a objetos e métodos comuns ficam na superclasse (classe de generalização):
  - Estendemos os atributos e métodos nas subclasses (classes de especialização);
- A herança viabiliza a construção de sistemas a partir de componentes facilmente reutilizáveis;
- A classe descendente não tem trabalho para receber a herança: basta usar *extends* e redefinir os métodos quando necessário (para gerar comportamentos mais especializados).

Herança Pura vs Extensão

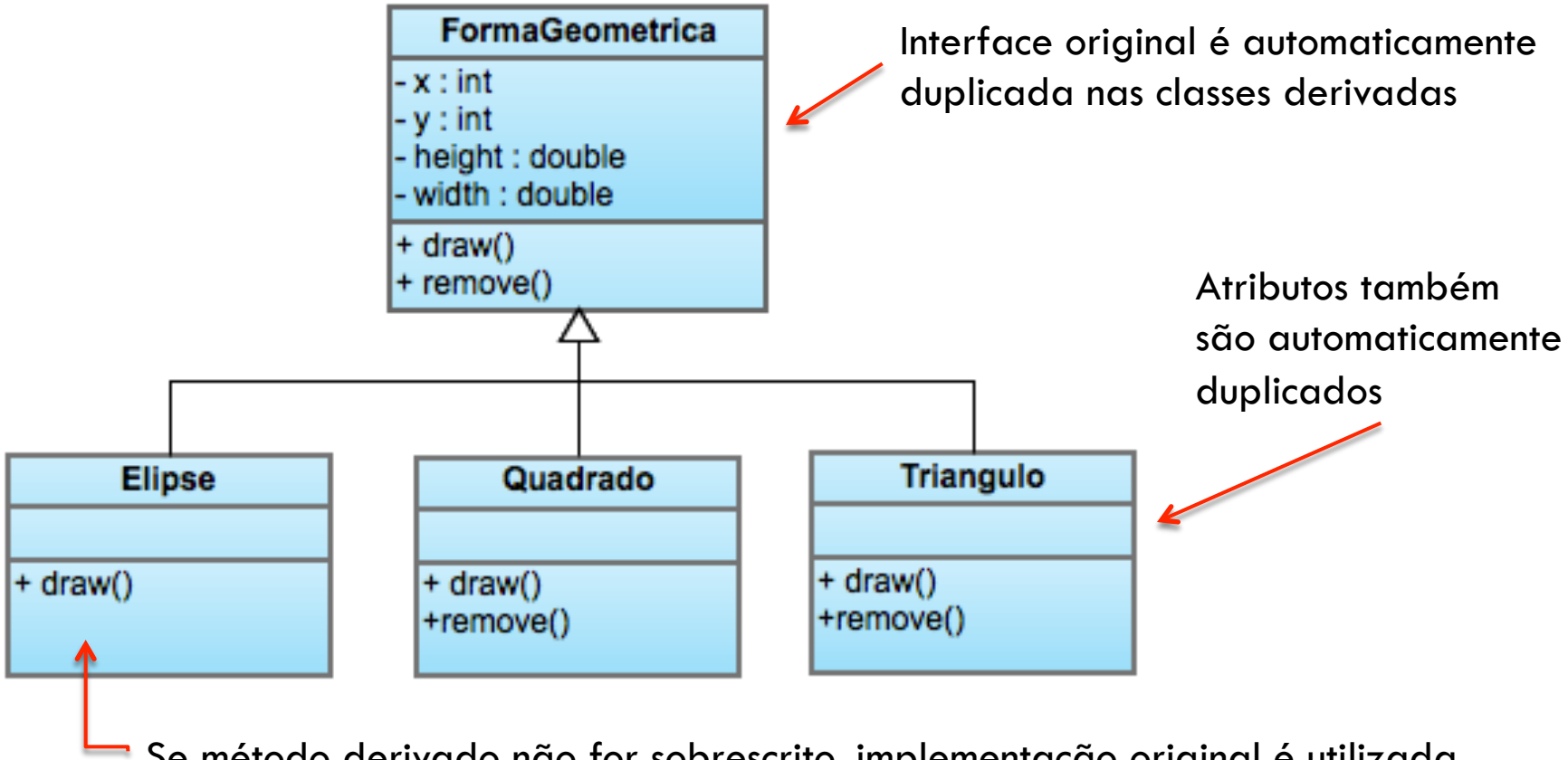
Herança Pura: métodos genéricos sobrepostos

Extensão: novos comportamentos adicionados





Árvore de Herança



Herança


```
class FormaGeometrica {
        public int x, y;
        public double height, width;

        public void draw() {
        }
        public void remove() {
        }
}
```

Assinatura do método sobrescrito deve ser idêntico ao declarado na superclasse.

```
public class Elipse extends FormaGeometrica {
        public void draw() {
        //desenha elipse
        }
}
```

Árvore de Herança

- União das classes que herdam entre si gera uma árvore de herança (hierarquia de classes relacionadas)
  - Todos as classes herdam características (gerais) definidas em FormaGeometrica;
  - Elipse, Quadrado e Triângulo são especializações de FormaGeometrica ("Elipse é uma FormaGeometrica");
- Em todos os casos, cada subclasse possui uma única superclasse: herança simples;
  - Em algumas linguagens é possível herdar a partir de diversas superclasses mas em Java, não.

Herança

- Se B é uma subclasse de A, então:
  - Os objetos de B suportam todas as operações suportadas pelos objetos de A, exceto aquelas que foram sobrescritas;
  - Os objetos de B incluem todas as variávies de instância de B + todas as variáveis de instância de A;
  - Métodos e Atributos declarados como *private* não serão herdados;
- Construtores também não serão herdados:
  - Deverão ser chamados (em cascata) na construção de objetos especializados via super();

Herança

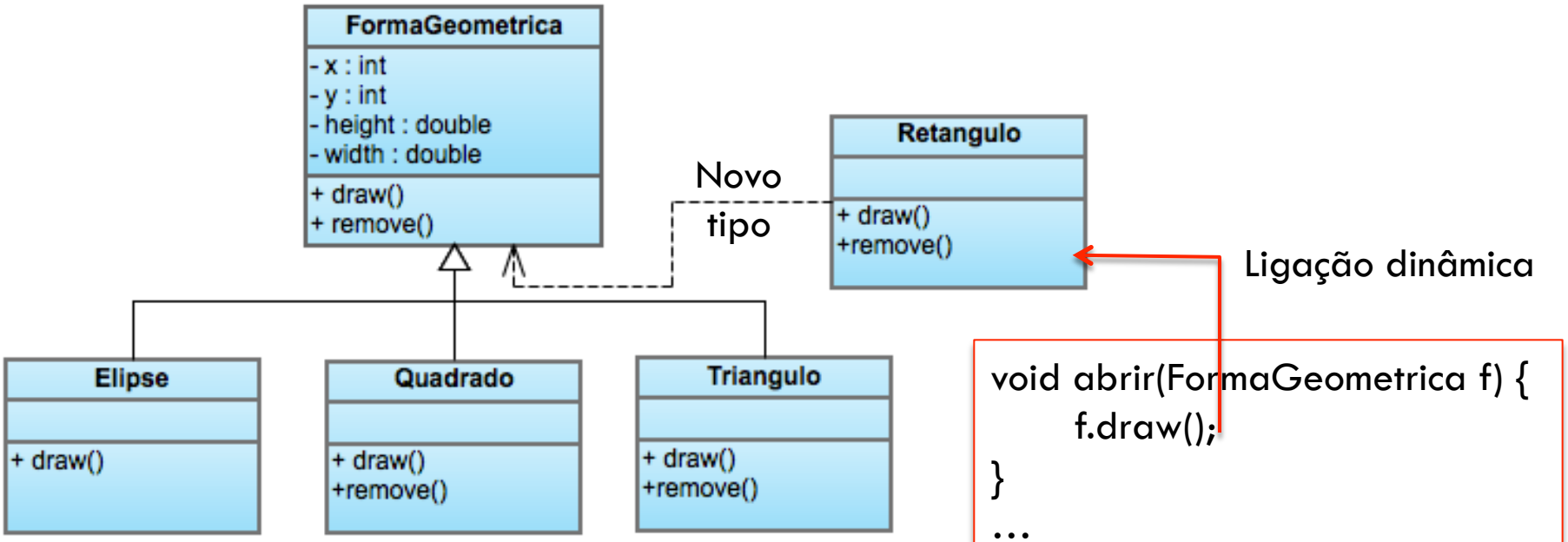
- Técnica para prover suporte a especialização:
  - Uma classe mais abaixo na hierarquia deve especializar comportamentos (“ é um tipo mais especializado de…”);
  - Métodos e atributos internos são herdados por todos os objetos dos níveis abaixo (com exceção do que for declarado como *private*);

Herança: vantagens

- Como o código pode ser facilmente reutilizado, a quantidade de código a ser adicionado em uma subclasse pode diminuir bastante;
  - Subclasses provêem comportamentos especializados, tomando como base os elementos comuns;
- Potencializa a manutenção de sistemas:
  - Maior legibilidade do código existente;
  - A herança é vista facilmente no código.

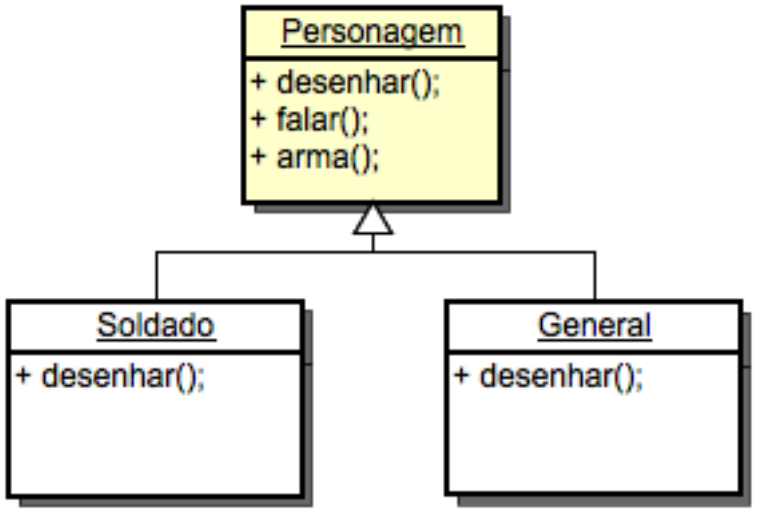
Herança: vantagens

- Quando relacionamos duas classes via herança, podemos ter polimorfismo com ligação dinâmica (*dynamic binding*):
  - Se um trecho de código usa uma referência de uma superclasse (FormaGeometrica), então este trecho pode manipular novos tipos concretos futuros (Retangulo, e.g.):



Cenário de Uso 1: Jogo de Ação

- Imagine uma modelagem de um sistema para jogo de ação com personagens;
- Neste jogo existem dois tipos de personagens:
  - Soldado;
  - General.



Cenário de Uso 1: Jogo de Ação


```
public abstract class Personagem {
        public abstract void desenhar();

        public void falar(){
        /* código comum para falar */ }

        public void arma(){
        /* código comum para atirar */ }
}
```

Classe abstrata que possui a interface comum a todos os personagens

Subclasses redefinem comportamentos específicos

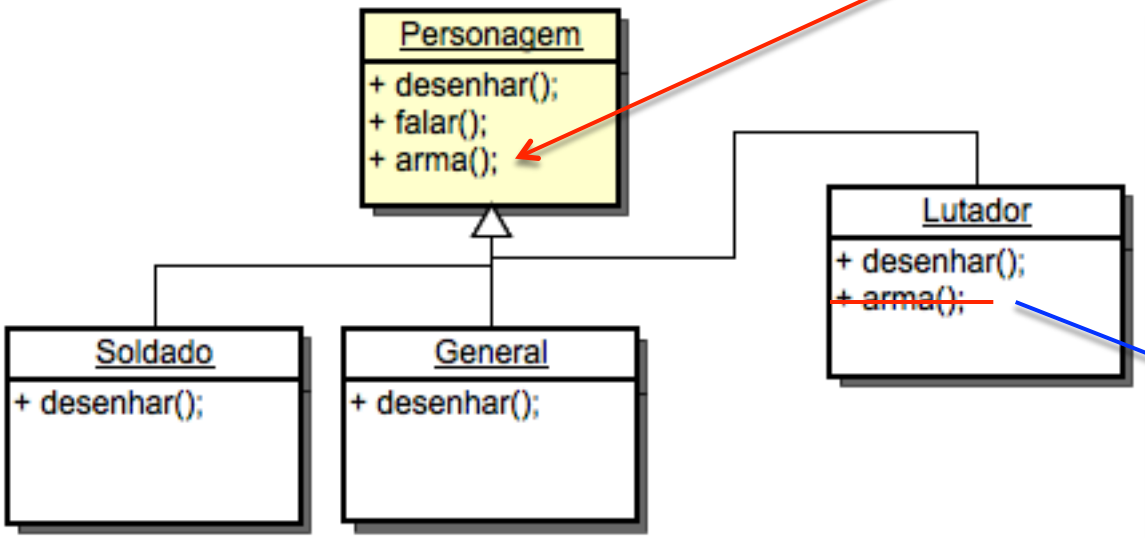
```
public class Soldado extends
Personagem {
        public void desenhar() {
                /*desenha soldado*/
        }
}
```

```
public class General extends
Personagem {
        public void desenhar() {
                /*desenha general*/
        }
}
```

Cenário de Uso 1: Jogo de Ação

- Adicionando um novo personagem: um LUTADOR;
  - A implementação de falar pode ser herdada sem problemas;
  - Desenhar pode ser sobrescrito;
  - O lutador não usa arma.

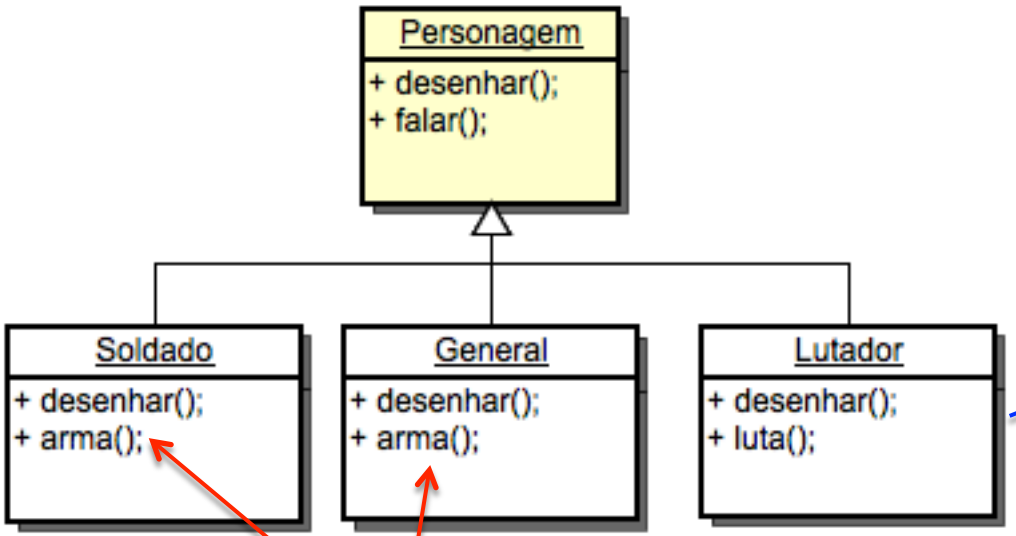
Agora temos comportamentos que não são mais comuns a todos os objetos!




```
public class Lutador extends
Personagem {
        public void desenhar() {
                /*desenha lutador*/
        }
        public void arma() {
                /*deixa em branco?*/
        }
}
```

Cenário de Uso 1: Jogo de Ação

- E se retirarmos o método arma() da superclasse?
  - Define-se, então, o método arma() em cada uma das subclasses?




```
public class Lutador extends
Personagem {
        public void desenhar() {
                /*desenha lutador*/
        }
        public void luta() {
                /*karatê!*/
        }
}
```

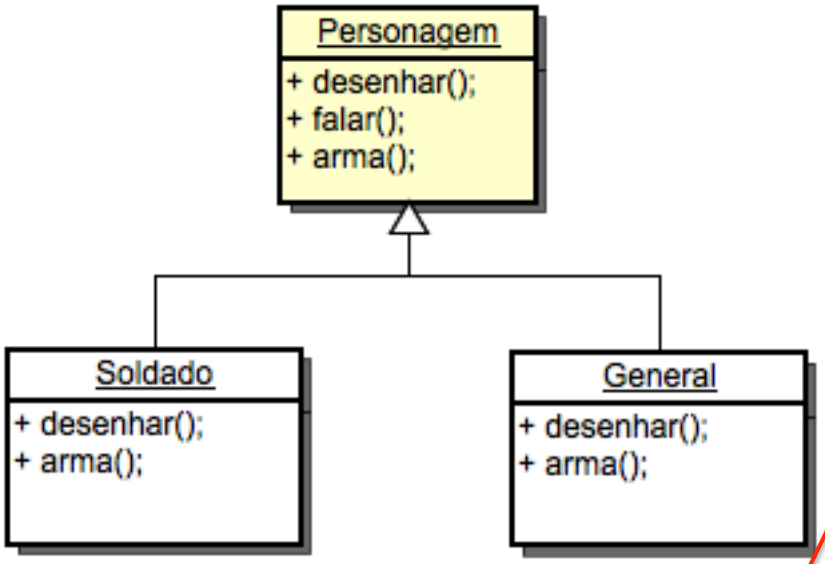
Agora temos uma duplicação desnecessária de código!

Problemas com a Herança

- O encapsulamento entre classes e subclasses é fraco, gerando forte acoplamento;
  - Mudanças na superclasse afetam todas as subclasses, podendo gerar um retrabalho enorme (*weak base-class problem*);
- Outro problema é a mudança de comportamento dinamicamente (em tempo de execução)

Cenário de Uso 2: Jogo de Ação

- Mesma modelagem com General e Soldado usando armas de fogo distintas:




```
public class Soldado extends Personagem {
        public void desenhar() {
                /*desenha soldado*/
        }
        public void arma() {
                println("Tiro");
        }
}
```

```
public class General extends Personagem {
        public void desenhar() {
                /*desenha general*/
        }
        public void arma() {
                println("Rajada");
        }
}
```

Subclasses redefinem métodos para um comportamento específico

Cenário de Uso 2: Jogo de Ação

- Problemas:
  - E se quisermos adicionar novas armas?
  - Como trocar a arma dinâmicamente?

```
public class UsaPersonagem {
        public static void main (String[] args) {
                Personagem p;
                p = new Soldado();
                p.arma(); //imprime "Tiro":
}
```

```
public class Soldado extends Personagem {
        public void desenhar() {
                /*desenha soldado*/
        }
        public void arma() {
                println("Tiro");
        }
}
```

```
public void arma(int arma) {
        if (arma==0)
                println("Tiro);
        else if (arma==1)
                println("Rajada);
        ... }
```

A arma está acoplada no código da classe soldado!

Problemas com Herança

- Não estamos tendo sucesso nestes cenários do Jogo com herança. Por que?
  1. No primeiro cenário, existem comportamentos na superclasse que não são comuns a todos os personagens do jogo;
  2. No segundo cenário, o código da arma específica está acoplado a cada uma das classes. Isto dificulta a criação de novas armas para o jogo e não permite que um personagem mude de arma em tempo de execução.
- Solução?

Composição

Reuso com Composição em Java

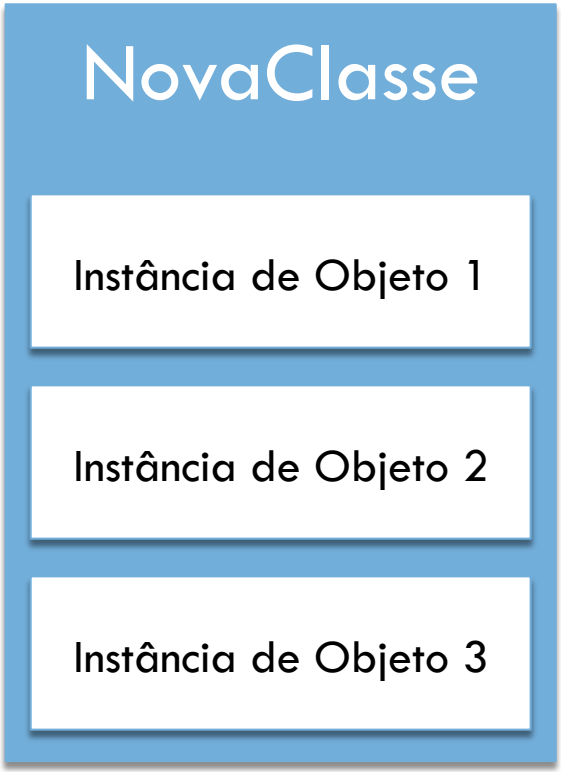
Como melhorar o reuso?

- Separar as partes que podem mudar, das partes que não mudam:
  - Ao descobrir que algo irá mudar, a idéia é encapsular isso, trabalhar com uma interface e usar o código sem a preocupação de ter que reescrever tudo caso surjam versões futuras (fraco acoplamento)
- Programe para uma interface, 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.

Composição

- A composição estende uma classe pela delegação de trabalho para outro objeto:
  - Em vez de codificar um comportamento estaticamente, definimos e encapsulamos pequenos comportamentos padrão e usamos composição para delegar comportamentos;

Composição em Java




```
class NovaClasse {
    Um u = new Um();
    Dois d = new Dois();
    Tres t = new Tres();
}
```

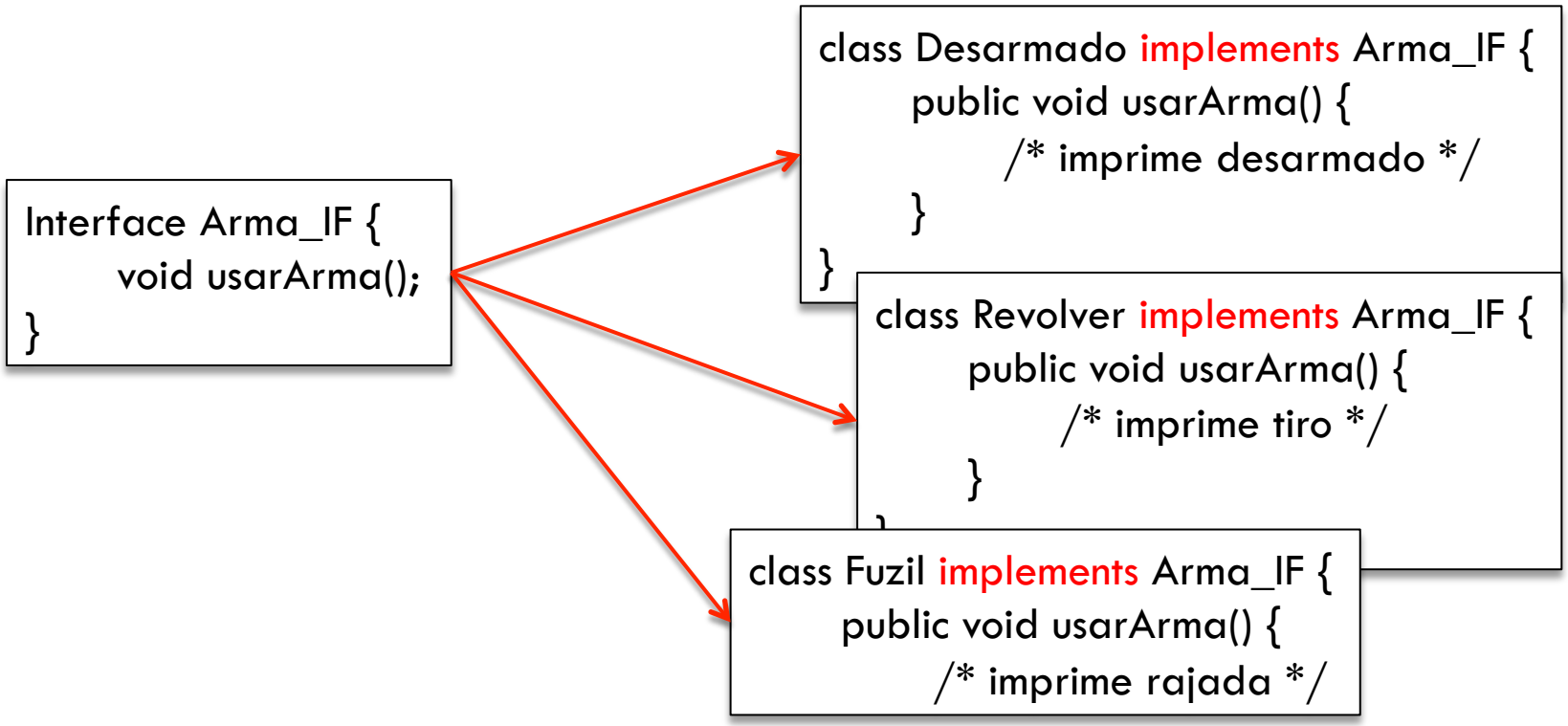
- Objetos podem ser inicializados no construtor;
- Flexibilidade: pode trocar objetos durante a execução;
- Relacionamento: “tem um”.

Qual a idéia para este Cenário?

- Permitir o aparecimento de novas armas no Jogo sem grandes impactos negativos (if-else's) em todas as subclasses concretas;
- Permitir que o personagem mude de arma em tempo de execução ao invés de definir a arma estaticamente no código;
- Alguns comportamentos não devem ser compartilhados por todos os personagens:
  - Um lutador não deve atirar, por exemplo.

Cenário de Uso 3: Jogo de Ação

- Se as armas podem mudar, devemos então separá-las das classes concretas:



Cenário de Uso 3: Jogo de Ação

- Os personagens concretos devem trabalhar agora como uma referência para algo que implemente a interface de armas:

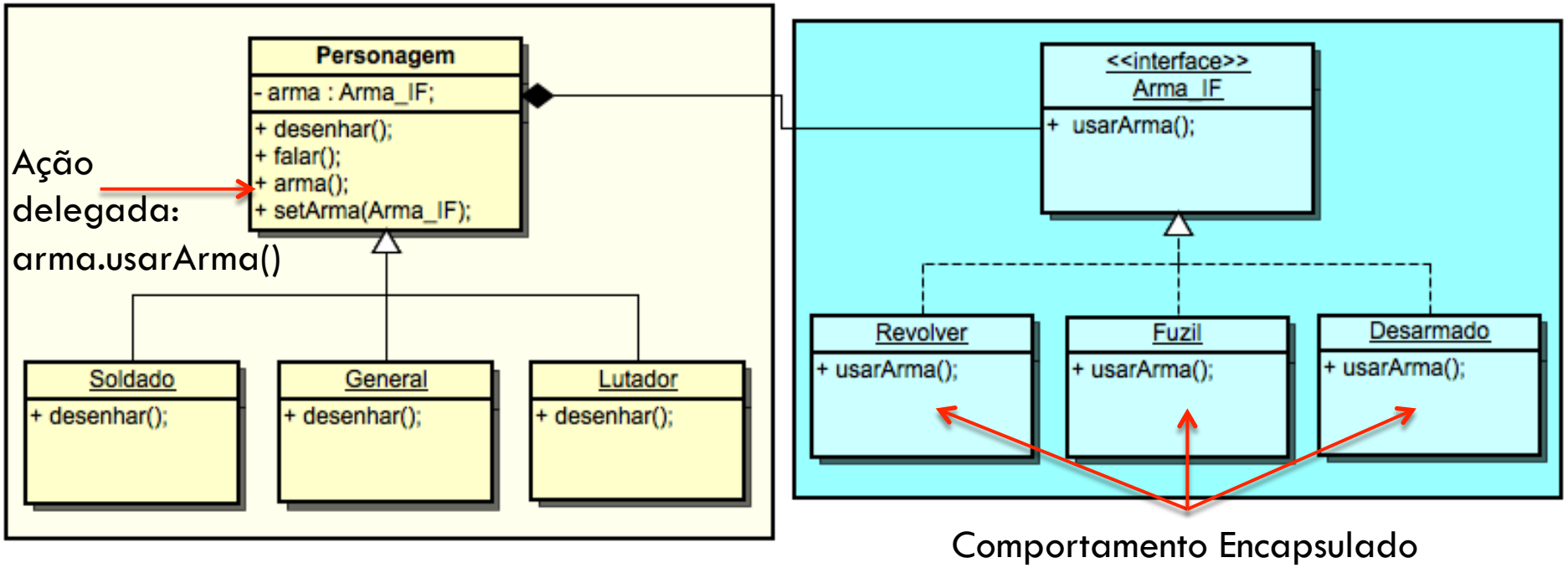
```
public abstract class Personagem {
        Arma_IF arma;

        public abstract void desenhar();
        public void falar() {
                /*código comum para falar*/
        public void setArma (Arma_IF arma) {
                this.arma =arma;
        }
        public void arma() {
                arma.usarArma();
        }
}
```

- Reduz o acoplamento de código já que os personagens interagem com uma interface ao invés de uma implementação;
- Se um personagem deseja utilizar a arma, basta delegar esta tarefa ao objeto (que tem o método usarArma) que está sendo referenciado no momento;

Cenário de Uso 3: Jogo de Ação

- Diagrama de Classes Final:



Considerações

Composição vs Herança

Considerações

- Uma das principais atividades em um projeto orientado a objetos é estabelecer corretamente o relacionamento entre diferentes classes;
  - Duas formas básicas de relacionar classes são herança e composição;
- Composição e Herança não são mutuamente exclusivas;
- A herança, geralmente, ocorre mais no design de tipos (uma subclasse é um tipo de...)

Considerações

- No desenvolvimento, porém, a composição de objetos é a técnica predominante:
  - Separar o que muda do que não muda e encapsular estes comportamentos;
  - Trabalhar com uma interface para manipular estes comportamentos;
  - Trocar comportamentos dinamicamente;
- Programar para uma interface sempre que possível garante um fraco acoplamento

Quando usar Herança e Composição?

- Identifique os componentes do objeto (suas partes)
  - Estas partes devem ser agregadas ao objeto via composição;
  - Este é um 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.

Para Discutir

- O que deve ser alterado no Jogo de Ação para acomodar um personagem Dragão Alado?
- Um dragão pode falar? E qual é a arma do dragão?