**Universidade Federal de Uberlândia**
**Faculdade de Computação**
**Programação Orientada a Objetos 2**
**Prof. Fabiano Dorça**

# Padrões de Projeto

## Padrão Strategy

## Padrão Strategy

- As vezes, apenas herança não resolve…
- Exemplo:
  - Uma classe A implementa dois métodos, m1(a1) e m2(a2).
  - As seguintes sub-classes de A implementam os métodos:

    B [m1(a1) e m2(a2)];
    C [m1(a1) e m2(a3)];
    D [m1(a4) e m2(a2)];
    E [m1(a4) e m2(a3)].

## Padrão Strategy

- Permite definir famílias de comportamentos, que podem ser (re)utilizados de forma intercambiável.

## Padrão Strategy

- Permite que o algoritmo varie independentemente dos clientes que o usam.

## Padrão Strategy

- Cada comportamento é encapsulado em **uma** classe.

## Padrão Strategy

Deve-se perceber o **que varia nas subclasses e encapsular à parte** (famílias de comportamentos reutilizáveis).

## Padrão Strategy

- É útil quando se tem-se **operações comuns a uma série de sub-classes**, *mas não é possível o uso de herança de forma eficiente*.
    - Desta forma, o padrão strategy permite configurar em tempo de execução cada subclasse com os comportamentos adequados

## Padrão Strategy

O padrão Strategy nos conduz a seguinte orientação:

1 - Programe sempre para interfaces;

2 - Dê preferência a composição ao invés de herança.

– Desta forma consegue-se o reuso e intercâmbio de comportamentos entre diversas classes, facilitando a expansão e manutenção.

## Padrão Strategy

- Benefícios

  - Alto nível de reuso.
  - Facilidade de expansão sem modificar o que está pronto (aberto para extensão, fechado para modificação).
  - Alto nível de Flexibilidade.

## Padrão Strategy

Esse padrão tem como elementos participantes

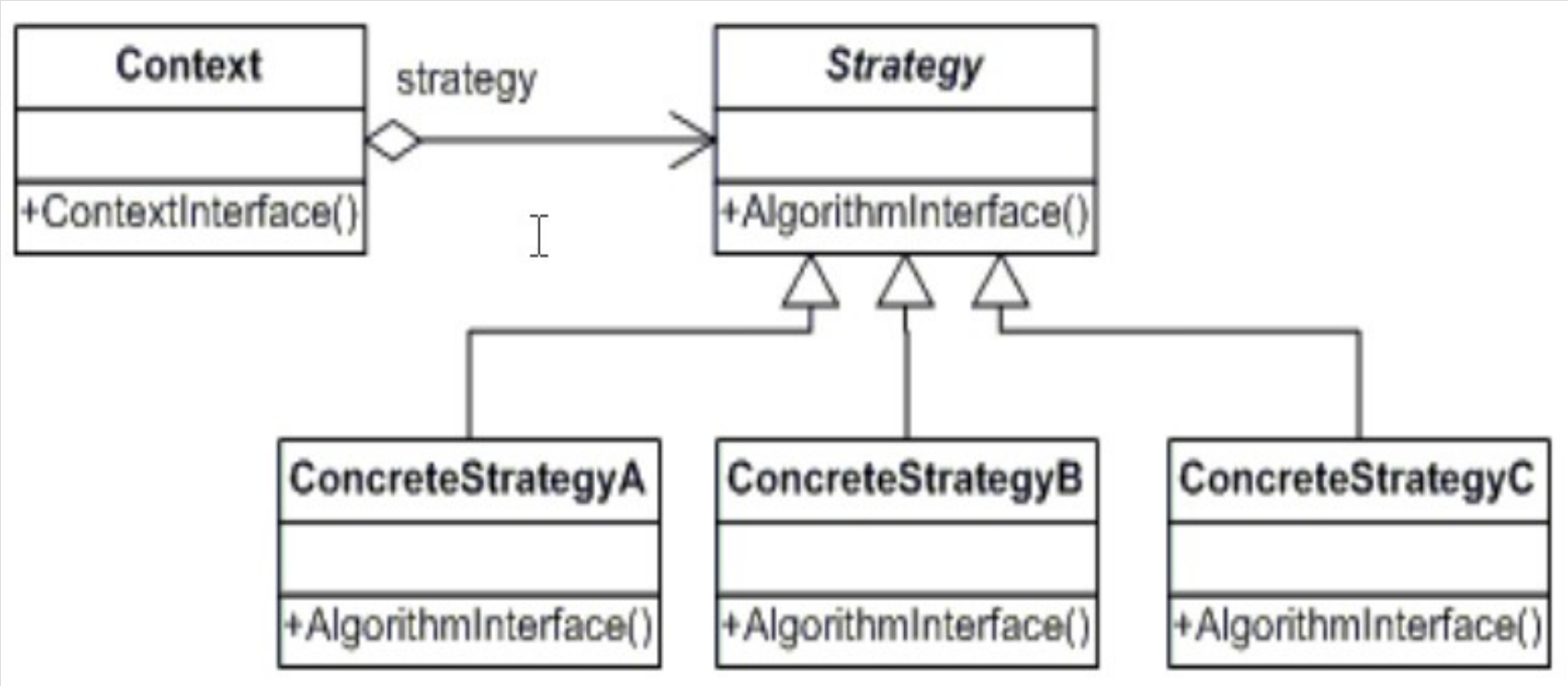
- **Context**, que tem seu "comportamento" ou parte dele definido pelo algoritmo implementado por uma Strategy;

- **Strategy**, que define a interface comum para todos os comportamentos;

- **ConcreteStrategy**, que implementa o comportamento definido pela interface Strategy.

# Padrão Strategy

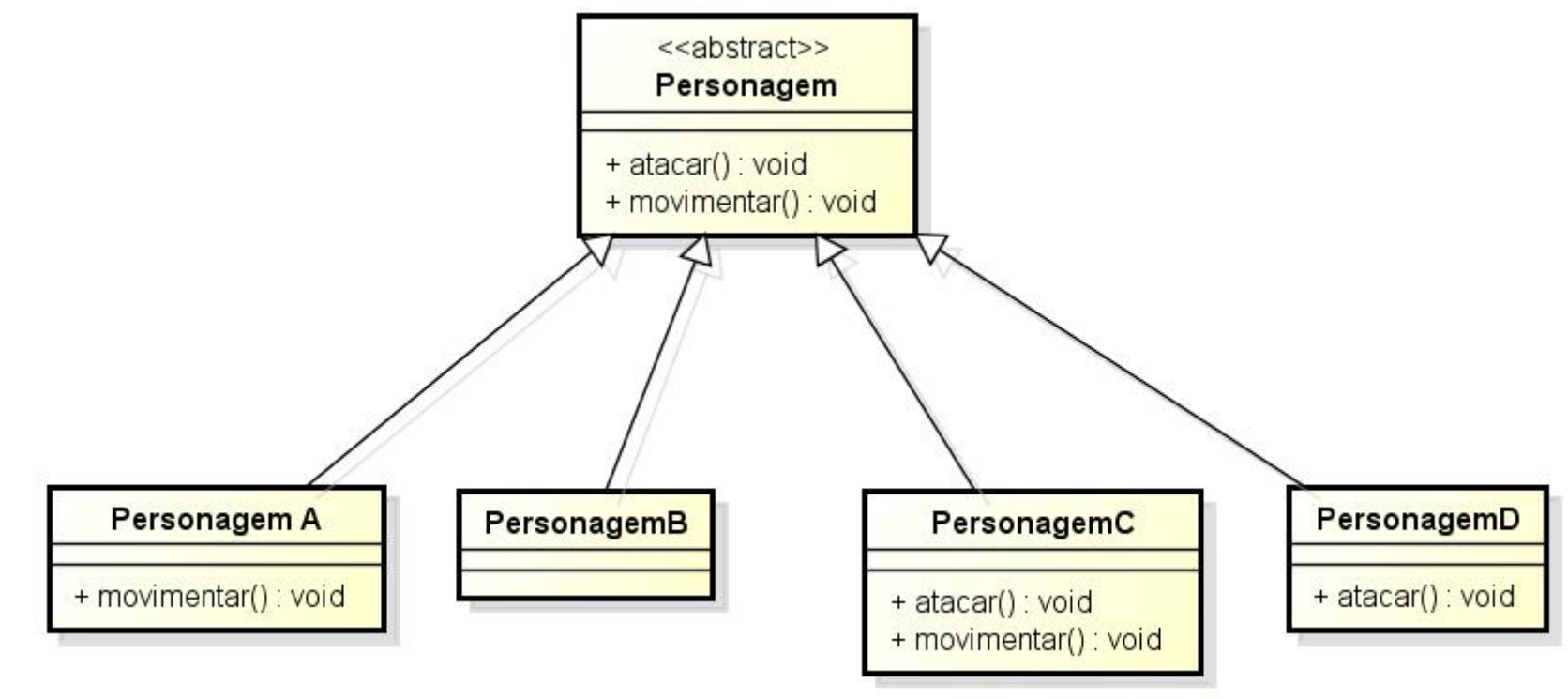
- Diagrama de Classes



## Padrão Strategy

- Problema prático
  - Diferentes personagens de um jogo que exibem diferentes ações e comportamentos.
  - Os personagens A e B implementam o mesmo ataque (forte). Mas os personagens C e D implementam outro tipo de ataque (fraco).
  - Além disto, os personagens A e C implementam o mesmo tipo de movimentação (rápido), e os personagens B e D possuem movimentação normal.
  - Neste caso, só herança não resolve, pois seria necessário replicar a mesma forma de ataque/movimentação.

# Padrão Strategy



# Padrão Strategy

**Solução tradicional com uso de herança**

- **public abstract class Personagem{**

```
...
public void atacar(){
    //implementa ataque forte   }

public void movimentar(){
    //implementa movimentação normal   }

}
```

## Padrão Strategy

```
public class PersonagemA extends Personagem{

      public void movimentar(){
        //implementa movimentação rápida   }
}

public class PersonagemB extends Personagem{


   ...
}
```

## Padrão Strategy

```
public class PersonagemC extends Personagem{

   public void atacar(){
        //implementa ataque fraco   }

      public void movimentar(){
        //implementa movimentação rápida   }
}

public class PersonagemD extends Personagem{

   public void atacar(){
        //implementa ataque fraco   }

}
```

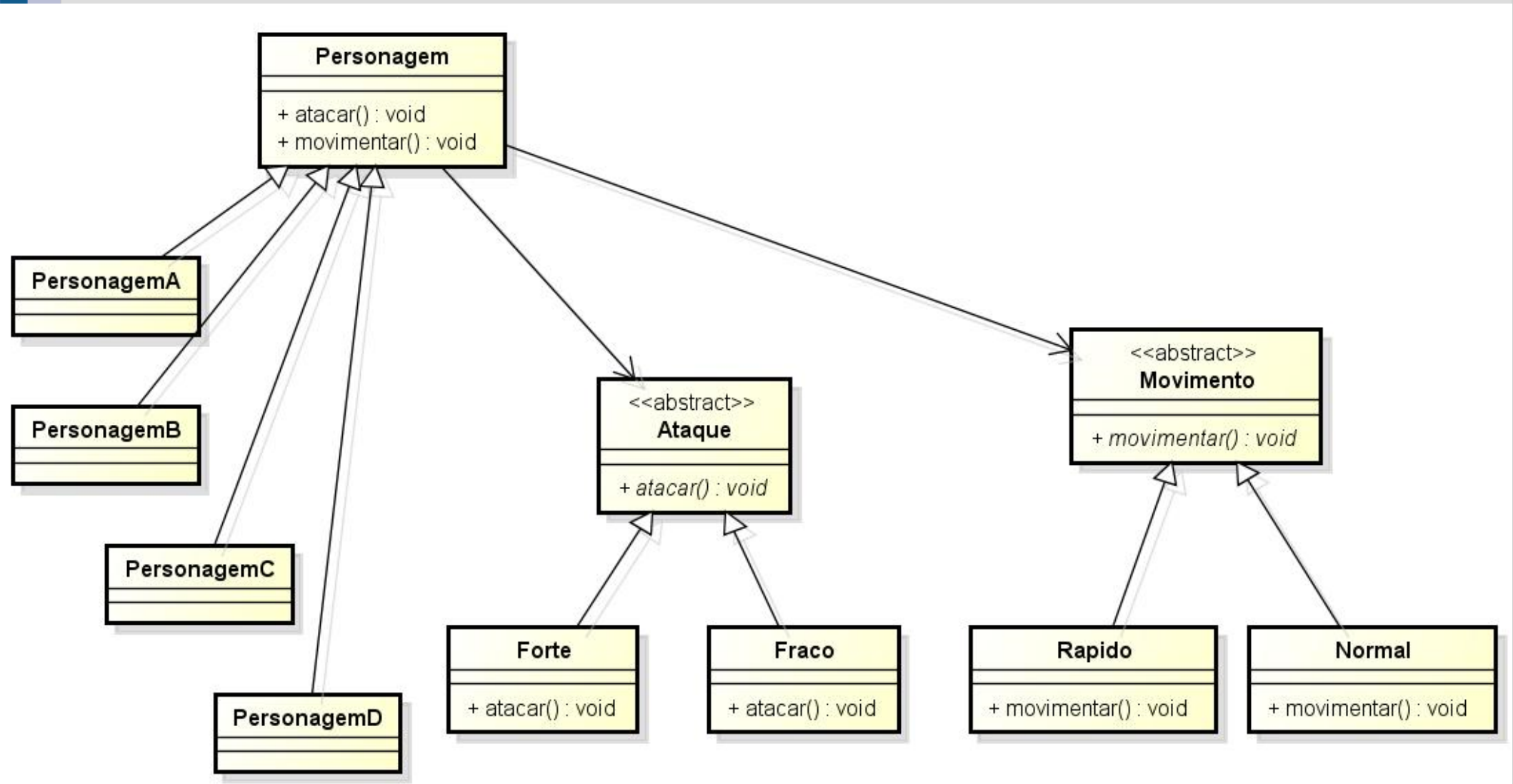
## Padrão Strategy

- Então, é importante identificar os aspectos que variam e sapará-los do que é comum à todas as sub-classes.
- E usar composição ao invés da herança.

→ Qual o aspecto que varia? Quantas variações temos?

# Padrão Strategy

- Aplicando o pradrão strategy...



## Padrão Strategy

- Aplicando o Padrão Strategy para resolver o problema

*Strategy (estratégias abstratas)*

```
public abstract class Ataque{
   public abstract void atacar();
}

public abstract class Movimento{
   public abstract void movimentar();
}
```

## Padrão Strategy

*ConcreteStrategies*

```
public class Forte extends Ataque{
   public void atacar(){
      //implementa ataque forte;
   }
}

public class Fraco extends Ataque{
   public void atacar(){
      //implementa ataque fraco;
   }
}
```

## Padrão Strategy

```
public class Rapido extends Movimento{
   public void movimentar(){
      //implementa movimentação rápida;
   }
}

public class Normal extends Movimento{
   public void movimentar(){
      //implementa movimentação normal;
   }
}
```

# Padrão Strategy

*Context*

```
public abstract class Personagem{
   private Ataque a;
   private Movimento m;

   public void setAtaque(Ataque a) {    this.a = a;  }

   public void setMovimento(Movimento m) {  this.m = m;  }

   public void atacar() {    a.atacar();  }

   public void movimentar() {  m.movimentar();  }
}
```

## Padrão Strategy

*ConcreteContext*

```
public class PersonagemA extends Personagem{
     public PersonagemA() {
        setAtaque(new Forte() );
        setMovimento(new Rapido() );
     }
}

public class PersonagemB extends Personagem{
     public PersonagemB() {
         setAtaque(new Forte() );
         setMovimento(new Normal() );
     }
}
```

## Padrão Strategy

```
public class PersonagemC extends Personagem{
    public PersonagemC() {
       setAtaque(new Fraco() );
       setMovimento(new Rapido() );
    }
}

public class PersonagemD extends Personagem{
     public PersonagemD() {
        setAtaque(new Fraco() );
        setMovimento(new Normal() );
     }
}
```

- Neste caso, o construtor do personagem concreto é responsável por definir na instanciação seus comportamentos

## Padrão Strategy

- Testando…

```
public static void main(){

        PersonagemA p = new PersonagemA();
        p.atacar();
        p.movimentar();
        …
        p.setMovimento(new Normal() );
        p.movimentar();
        …
        p.setAtaque(new Fraco() );
        p.atacar();
        …
        //o comportamento do objeto é alterado em tempo de
          execução, como se ele tivesse mudado de classe
```

## Padrão Strategy

**- Alto nível de reuso:**

- É fácil implementar novos personagens utilizando os comportamentos disponíveis, e é fácil implementar e utilizar novos comportamentos
    - Facilidade de expansão sem modificar o que está pronto:
    - Implementar novos personagens que utilizam novos tipo de ataque
    - Note que o código pronto não é mexido na expansão

**- Alto nível de Flexibilidade:**

- Implementar um novo personagem que inicia com determinado tipo de comportamento, mas em determinado momento pode ter seu comportamento alterado em tempo de execução.

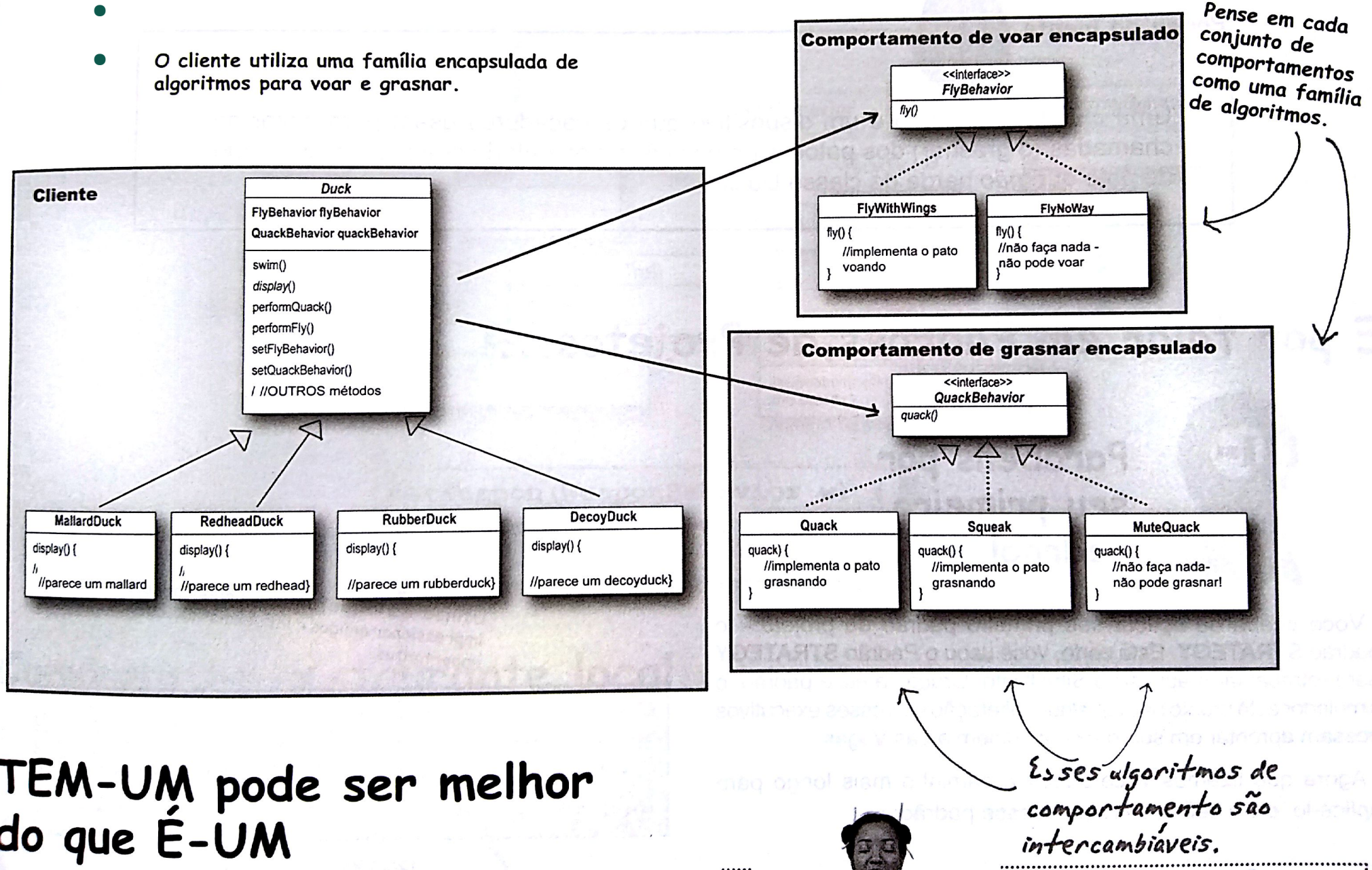
**- Aberto para extensão, fechado para modificação.**

## Padrão Strategy

- Deve-se produzir uma família de classes para cada comportamento que varia.
- Deve-se ter uma estrutura de herança em que cada subclasse implementa uma variação do algoritmo.
- Possibilita que um algortimo seja substituído por outro em tempo de execução.
- Possibilita alto nível de reuso (vários contexts utilizamento a mesma família de estratégias).

# Exemplo (Freeman, Use a cabeça – Padrões de Projeto):

- O cliente utiliza uma família encapsulada de algoritmos para voar e grasnar.

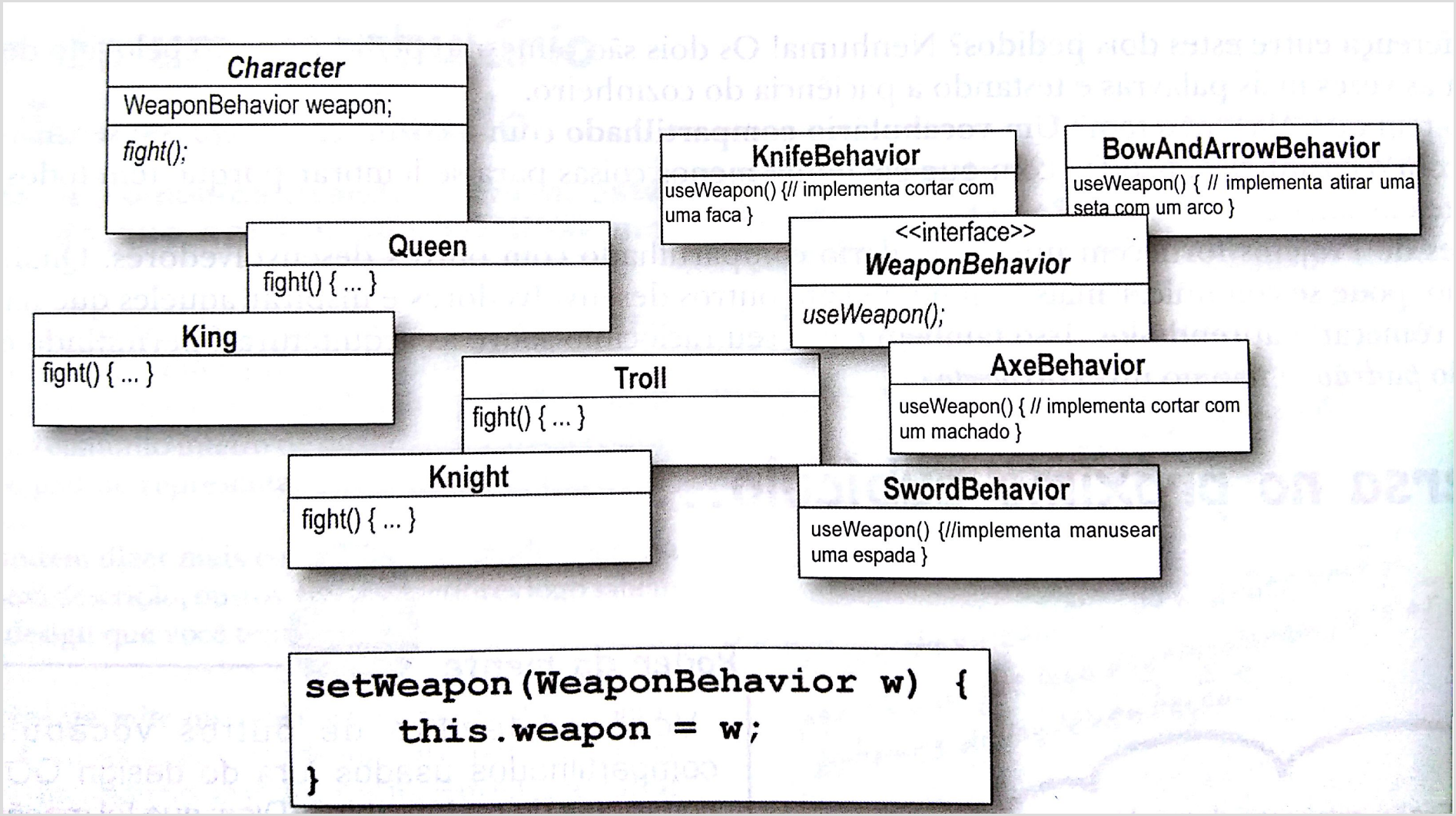



TEM-UM pode ser melhor do que É-UM

## Padrão Strategy

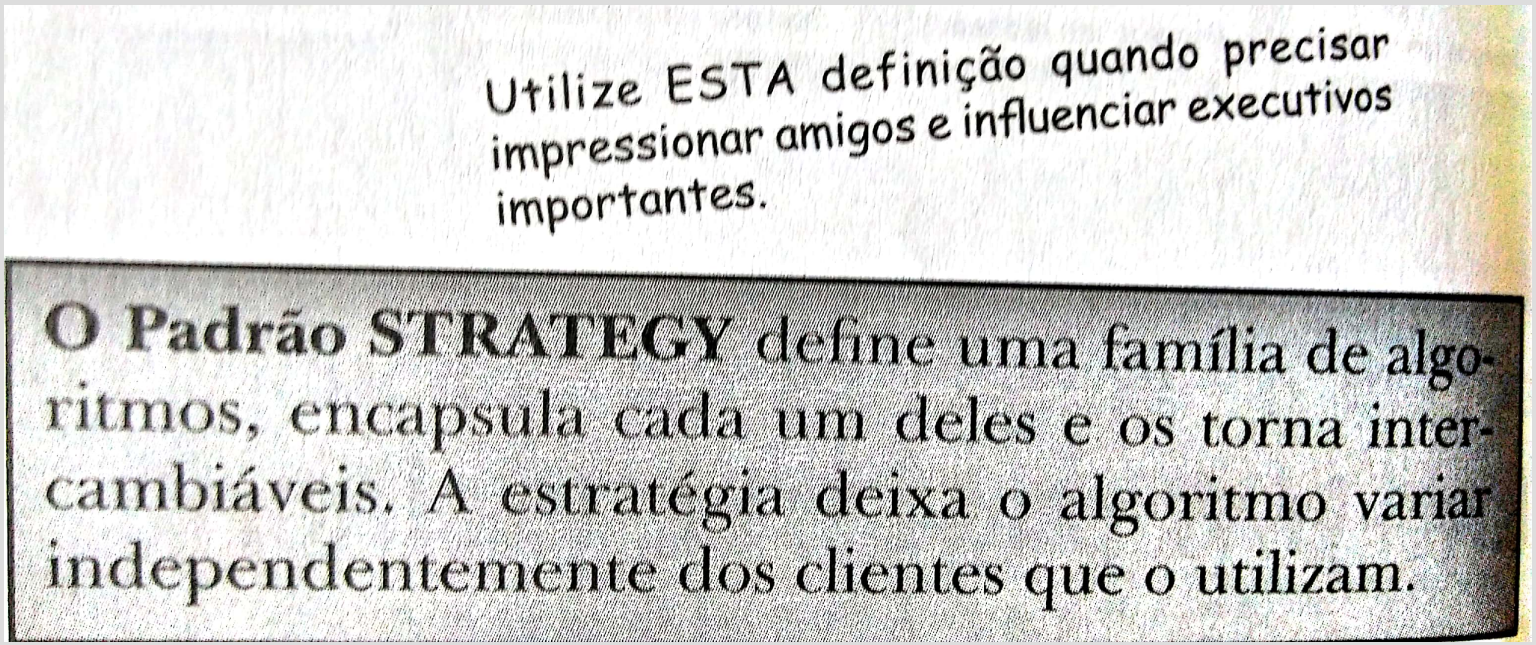
- Exercício:
- A seguir é apresentado um conjunto de classes e interfaces para um jogo de aventura.
- Existem classes que implementam personagens do jogo e classes que implementam comportamentos das armas que podem ser utilizadas por estes personagens.
- Cada personagem pode utilizar uma arma de cada vez, mas pode alterar as armas a qualquer momento durante o jogo.
-
- Pede-se
  - Utilizando estas classes e a interface disponibilizada, utilize o padrão strategy para projetar o jogo.
  - Implemente em java (obs. Simule a implementação dos comportamentos utilizando saídas de texto).

# Padrão Strategy




Freeman, Use a cabeça – Padrões de Projeto.

# Padrão Strategy




Freeman, Use a cabeça – Padrões de Projeto.

# Padrão Strategy

- Fim.