Instituto Federal de Educação, Ciência e Tecnologia

Triângulo Mineiro – Campus Uberlândia Centro

Curso Superior de Tecnologia em Sistemas para Internet

Programação Orientada a Objetos 2

Slide 3 – Herança e Interface

Prof. Carlos Eduardo de Carvalho Dantas

carloseduardodantas@iftm.edu.br

https://carloseduardoxp.github.io/

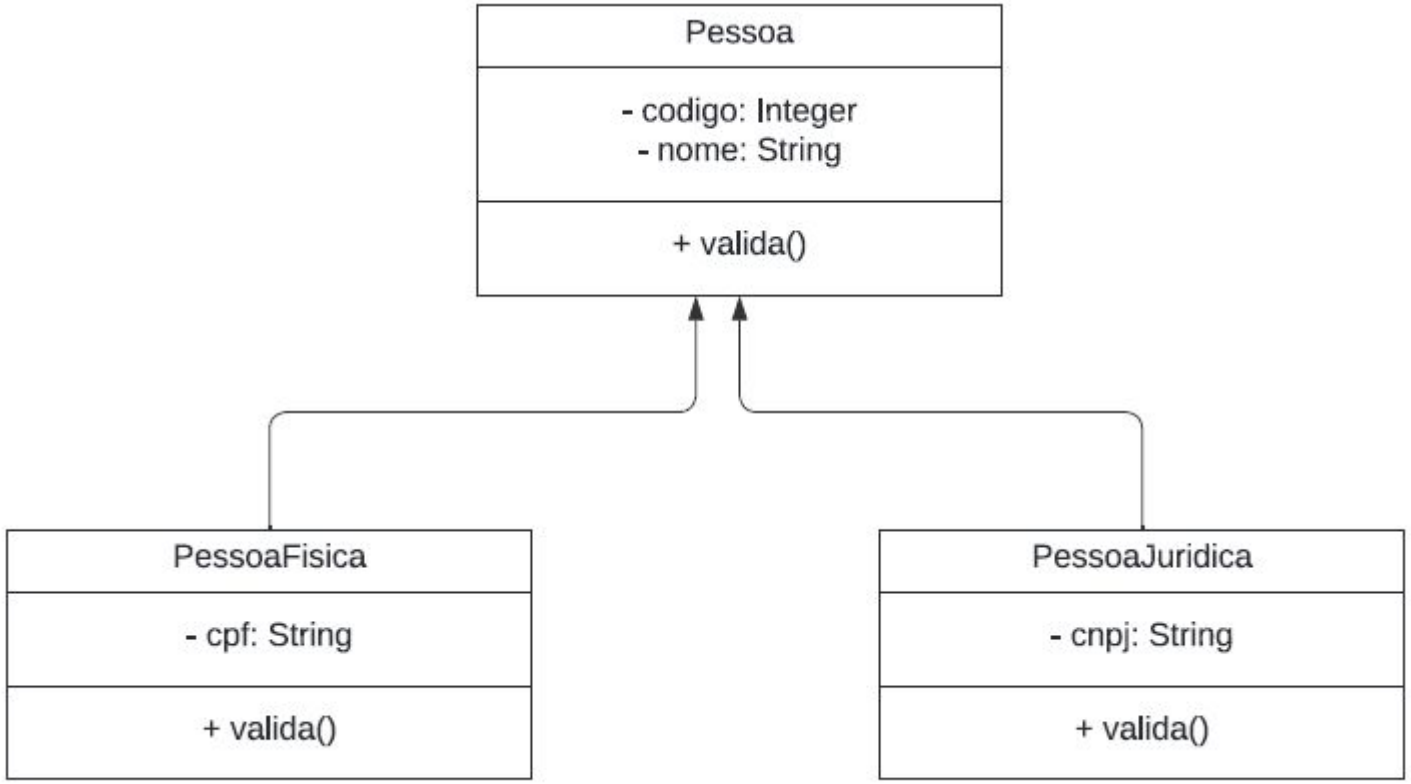
INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Herança - Introdução

- Recurso em Programação Orientada a Objetos que permite
  - **Reutilização de Código** – Classes “filhas” irão reutilizar o comportamento da classe “mãe”.
  - **Organização e Estruturação** – Criação de uma hierarquia de classes
  - **Polimorfismo** – Objetos de diferentes classes sendo tratados como instâncias de uma classe comum
  - **Encapsulamento e Abstração** – Manter detalhes de implementação escondidos, e focar no que a classe faz ao invés de como faz.
  - **Extensibilidade** – Incluir novos recursos sem modificar as classes existentes.

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Herança



Herança

```java
package domain;

public class Pessoa {

    private Integer codigo;

    private String nome;

    public boolean valida() {
        boolean validouNome = true;

        return validouNome;
    }

}
```

Extends significa que PessoaFisica é "filha" de "Pessoa".

```java
package domain;

public class PessoaFisica extends Pessoa {

    private String cpf;

    @Override
    public boolean valida() {
        super.valida();

        boolean validouCpf = true;
        return validouCpf;
    }

}
```

super.valida() significa que irá executar o método valida() da classe "pai" e depois continuar a execução do método valida() da classe filha

@Override significa que o método valida() da classe "filha" está sobrescrevendo o método de mesmo nome da classe "pai"

```java
1   package domain;
2
3   public class PessoaJuridica extends Pessoa {
4
5       private String cnpj;
6
7       @Override
8       public boolean valida() {
9           super.valida();
10
11          boolean validouCnpj = true;
12          return validouCnpj;
13      }
14
15  }
```

INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Herança

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        Pessoa pessoa1 = new Pessoa();
        Pessoa pessoa2 = new PessoaFisica();
        Pessoa pessoa3 = new PessoaJuridica();

        List<Pessoa> pessoas = new ArrayList<>(Arrays.asList(pessoa1,pessoa2,pessoa3));

        for (Pessoa pessoa: pessoas) {
            chamaValidacao(pessoa);
        }

    }

    private static void chamaValidacao(Pessoa pessoa) {
        pessoa.valida();
    }
}
```

A assinatura do método e o loop aceitarão “qualquer” pessoa, seja a classe pai ou as filhas.

O código está genérico:
- Se a instância for de Pessoa: Valida o nome
- Se a instância for de PessoaFisica: Valida o nome e o cpf
- Se a instância for de PessoaJuridica: Valida o nome e o CNPJ

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

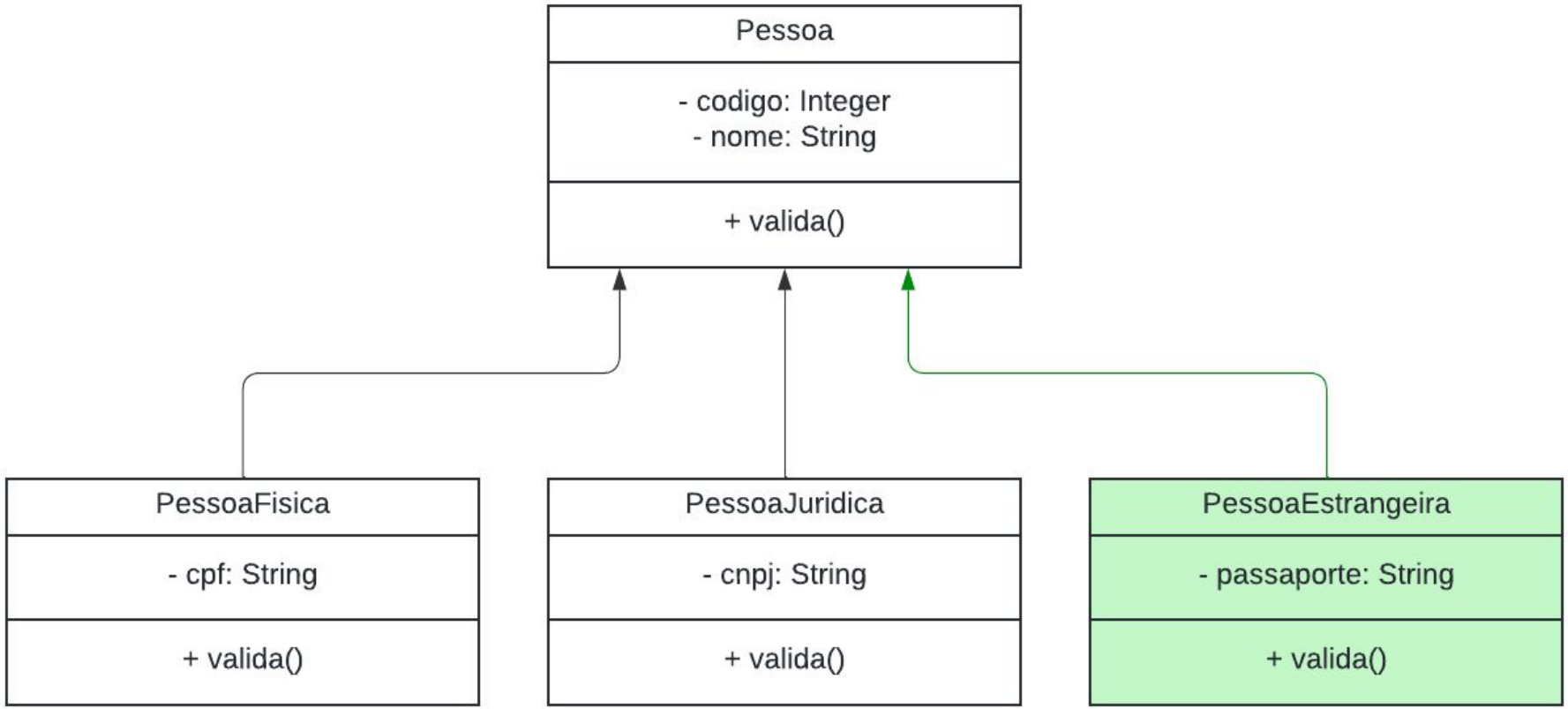
Só que ...

- Software constantemente demanda por mudança, pois novos requisitos são solicitados pelos usuários.

- Então o design do Software precisa ser flexível o bastante para que novas funcionalidades sejam incluídas no sistema **sem precisar de alterar “o sistema inteiro”.**

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Herança – Adicionando um novo “filho”



Herança – Adicionando um novo “filho”

```java
package domain;

public class PessoaEstrangeira extends Pessoa {

    private String passaporte;

    @Override
    public boolean valida() {
        super.valida();

        boolean validouPassaporte = true;

        return validouPassaporte;
    }
}
```

Veja que o novo “filho” respeita o contrato, ou seja, adiciona seus campos e implementa o método valida()

Caso o método valida() não fosse implementado, a classe App iria chamar o método valida() da classe Pessoa, ou seja, validaria apenas o nome, sem validar o passaporte.

Herança – Adicionando um novo “filho”

```
10  public class App {
        Run | Debug
11      public static void main(String[] args) throws Exception {
12          Pessoa pessoa1 = new Pessoa();
13          Pessoa pessoa2 = new PessoaFisica();
14          Pessoa pessoa3 = new PessoaJuridica();
15          Pessoa pessoa4 = new PessoaEstrangeira();
16
17          List<Pessoa> pessoas = new ArrayList<>(Arrays.asList(pessoa1,pessoa2,pessoa3,pessoa4));
18
19          for (Pessoa pessoa: pessoas) {
20              chamaValidacao(pessoa);
21          }
22
23      }
24
25      private static void chamaValidacao(Pessoa pessoa) {
26          pessoa.valida();
27      }
28  }
29
```

Veja que a inclusão de um objeto do tipo PessoaEstrangeira não provocou nenhuma alteração no loop for-each ou no método de chamaValidacao()

Essa implementação respeita o princípio SOLID chamado de OCP, ou seja, “aberto para extensão, fechado para modificação”.

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Herança – Resumindo...

- **Reutilização de Código** – O campo “nome” e o seu código de validação foi reaproveitado nas classes filhas, PessoaJuridica, PessoaFisica e PessoaEstrangeira.
- **Organização e Estruturação** – A hierarquia de classes é muito mais genérica do que construir uma classe apenas cheia de ifs e elses.
- **Polimorfismo** – O método chamaValidacao() e o loop funciona de forma exatamente igual para qualquer uma das instâncias de Pessoa, PessoaFisica, PessoaJuridica e PessoaEstrangeira.
- **Encapsulamento e Abstração** – Quem implementou a classe App não faz a menor idéia de como Pessoa e suas filhas implementam a validação.
- **Extensibilidade** – Adicionar PessoaEstrangeira não modificou em nada o design das classes já implementados.

Qual é a falha deste design?

```java
package domain;

public class PessoaEstrangeira extends Pessoa {

    private String passaporte;

    @Override
    public boolean valida() {
        super.valida();

        boolean validouPassaporte = true;

        return validouPassaporte;
    }

}
```

1

Toda vez que uma nova classe filha surgir, precisará implementar o super.valida() para que o código-fonte da classe “pai” valide o nome.

Esse tipo de erro é bem passível de acontecer. Esquecer “faz parte” do jogo.

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Qual é a falha deste design?

```
10  public class App {
        Run | Debug
11      public static void main(String[] args) throws Exception {
12          Pessoa pessoa1 = new Pessoa();
13          Pessoa pessoa2 = new PessoaFisica();
14          Pessoa pessoa3 = new PessoaJuridica();
15          Pessoa pessoa4 = new PessoaEstrangeira();
16
17          List<Pessoa> pessoas = new ArrayList<>(Arrays.asList(pessoa1,pessoa2,pessoa3,pessoa4));
18
19          for (Pessoa pessoa: pessoas) {
20              chamaValidacao(pessoa);
21          }
22
23      }
24
25      private static void chamaValidacao(Pessoa pessoa) {
26          pessoa.valida();
27      }
28  }
29
```

2

E se eu quiser que não exista uma “pessoa genérica”, ou seja, uma pessoa só pode ser Física, Jurídica ou Estrangeira?

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Herança – Classe Abstrata

- Não pode ser instanciada
- Projetada para ser uma classe base sobre as outras classes (filhas)
- Contém métodos abstratos e métodos concretos
- Funciona como se fosse um “template” para as classes filhas.

Herança – Classe Abstrata

```java
package domain;

public abstract class Pessoa {

    private Integer codigo;

    private String nome;

    protected abstract boolean validaAbstract();

    public boolean valida() {
        boolean validouNome = true;

        return validouNome && validaAbstract();
    }

}
```

Definição de que a classe é Abstrata

Método abstrato chamado validaAbstract() – toda classe filha de Pessoa será obrigada a implementar este método

O método é protegido justamente porque para o "mundo externo" só importa conhecer o método valida()

Agora o método valida() funciona como um template, onde a classe Pessoa consegue validar o nome, e cada classe filha irá "injetar" a sua respectiva validação com a implementação do método validaAbstract()

Herança – Classe Abstrata

Não é mais necessário executar o super.valida(), porque o método validaAbstract() é chamado pela classe Pessoa

A anotação @Override permanece porque estamos fazendo uma subscrita do método validaAbstract() definido na classe Pessoa.

```java
public class PessoaFisica extends Pessoa {

    private String cpf;

    @Override
    protected boolean validaAbstract() {
        boolean validaCPF = true;

        return validaCPF;
    }

}
```

```java
public class PessoaJuridica extends Pessoa {

    private String cnpj;

    @Override
    protected boolean validaAbstract() {
        boolean validaCnpj = true;

        return validaCnpj;
    }

}
```

```java
package domain;

public class PessoaEstrangeira extends Pessoa

    private String passaporte;

    @Override
    protected boolean validaAbstract() {
        boolean validaPassaporte = true;

        return validaPassaporte;
    }
```

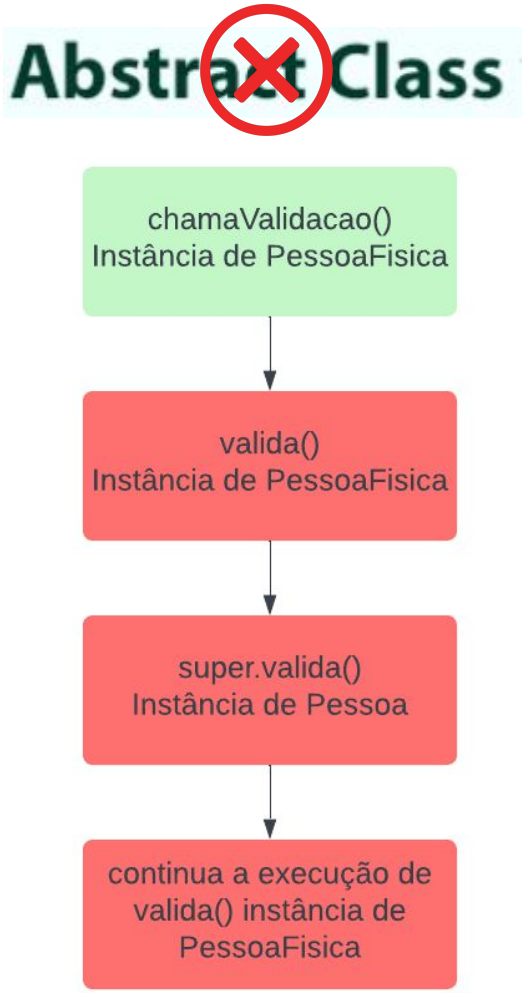
Herança – Classe Abstrata

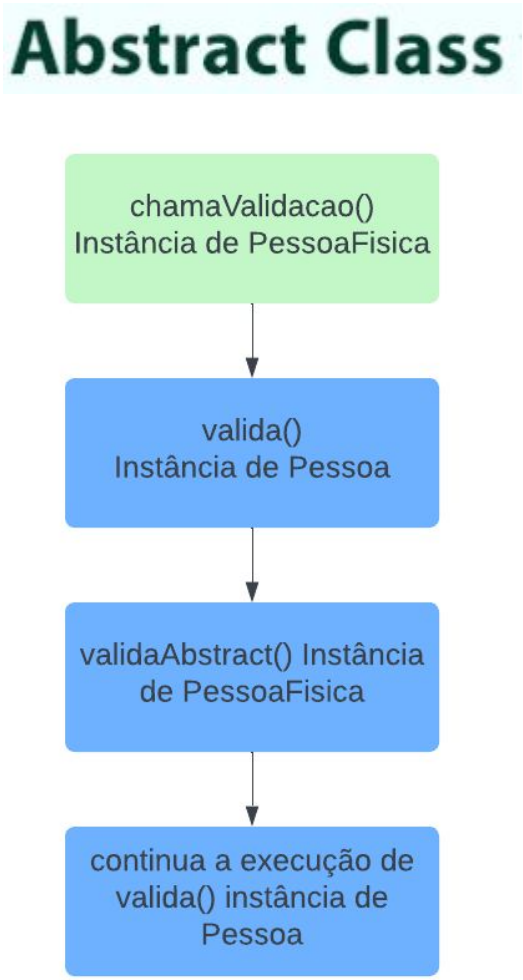
```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        Pessoa carlos = new PessoaFisica();
        Pessoa iftm = new PessoaJuridica();
        Pessoa lewisHamilton = new PessoaEstrangeira();

        List<Pessoa> pessoas = Arrays.asList(carlos,iftm,lewisHamilton);

        for (Pessoa pessoa: pessoas) {
            chamaValidacao(pessoa);
        }

    }

    private static void chamaValidacao(Pessoa pessoa) {
        pessoa.valida();
    }
}
```

Não é mais possível criar uma Pessoa pessoa = new Pessoa(), já que como Pessoa é classe abstrata, não se cria novas instâncias da classe “pai”

Embora a sintaxe dos dois exemplos sejam idênticos (com e sem classe abstrata), semanticamente mudou bastante.
No primeiro exemplo, a classe filha executava o método valida() e chamava super().valida() de Pessoa.
No segundo exemplo, a classe Pessoa criou um template para valida(), onde cada filho irá injetar a implementação do seu respectivo “valida”

Herança – Classe Abstrata





INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Mas ...

Herança pode ser o Mal da Orientação a Objetos — Parte 2

May 30, 2016

8 minute read

Cuidado com a Herança! pt

29/08/2013

orientação a objetos | herança | composição | juquinha | programação

A "herança maldita" na Orientação a Objetos

Do Not Use Inheritance

2020.11.06 - 2021.02.14

Inheritance in OOP is the most expensive way of coding. It really makes the code hard to understand and hard to modify. This article does not focus on the issues but the good news:

Why and how to avoid Inheritance in C#?

Siddharth Mittal · Follow

4 min read · Feb 11, 2023

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Qual é o problema?

- E se um usuário pedir um novo requisito para verificar se a Pessoa tem um nome sujo?

- Com CPF podemos olhar o SERASA
- Com CNPJ podemos verificar se está irregular
- **E com o passaporte de um estrangeiro?**

INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Qual é o problema?

```java
package domain;

public abstract class Pessoa {

    private Integer codigo;

    private String nome;

    protected abstract boolean validaAbstract();

    protected abstract boolean isPessoaRegular();

    public boolean valida() {
        boolean validouNome = true;

        return validouNome && validaAbstract() && isPessoaRegular();
    }

}
```

Com este novo método abstrato, agora toda classe filha de Pessoa é obrigada a implementar tal método.

Criou-se uma nova funcionalidade no template para validar se a pessoa é regular

Qual é o problema?

PessoaEstrangeira é obrigado a implementar o método, mesmo "contra a sua vontade"

```java
public class PessoaFisica extends Pessoa {

    private String cpf;

    @Override
    protected boolean validaAbstract() {
        boolean validaCPF = true;

        return validaCPF;
    }

    @Override
    protected boolean isPessoaRegular() {
        boolean chamaOSerasa = true;
        return chamaOSerasa;
    }

}
```

```java
public class PessoaJuridica extends Pessoa {

    private String cnpj;

    @Override
    protected boolean validaAbstract() {
        boolean validaCnpj = true;

        return validaCnpj;
    }

    @Override
    protected boolean isPessoaRegular() {
        boolean chamaAReceitaFederal = true;
        return chamaAReceitaFederal;
    }
}
```

```java
public class PessoaEstrangeira extends Pessoa {

    private String passaporte;

    @Override
    protected boolean validaAbstract() {
        boolean validaPassaporte = true;

        return validaPassaporte;
    }

    @Override
    protected boolean isPessoaRegular() {
        throw new UnsupportedOperationException(message:"Não tenho como saber se o estrangeiro é regular");
    }


}
```

INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Qual é o problema?

```
3   public abstract class Pessoa {
4
5       private Integer codigo;
6
7       private String nome;
8
9       protected abstract boolean validaAbstract();
10
11      protected abstract boolean isPessoaRegular();
12
13      public boolean valida() {
14          boolean validouNome = true;
15
16          if (this instanceof PessoaEstrangeira) {
17              return validouNome && validaAbstract();
18          } else {
19              return validouNome && validaAbstract() && isPessoaRegular();
20          }
21
22
23      }
24
25
26  }
```

Isso nos obriga a alterar o método valida(), onde caso a instância seja de PessoaEstrangeira, que não chame o método isPessoaRegular()

Isso é chamado "quebra de encapsulamento", já que a classe Pessoa se vê obrigada a conhecer o comportamento das classes filhas para evitar chamadas indesejáveis.

INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Herança e o uso do instanceof

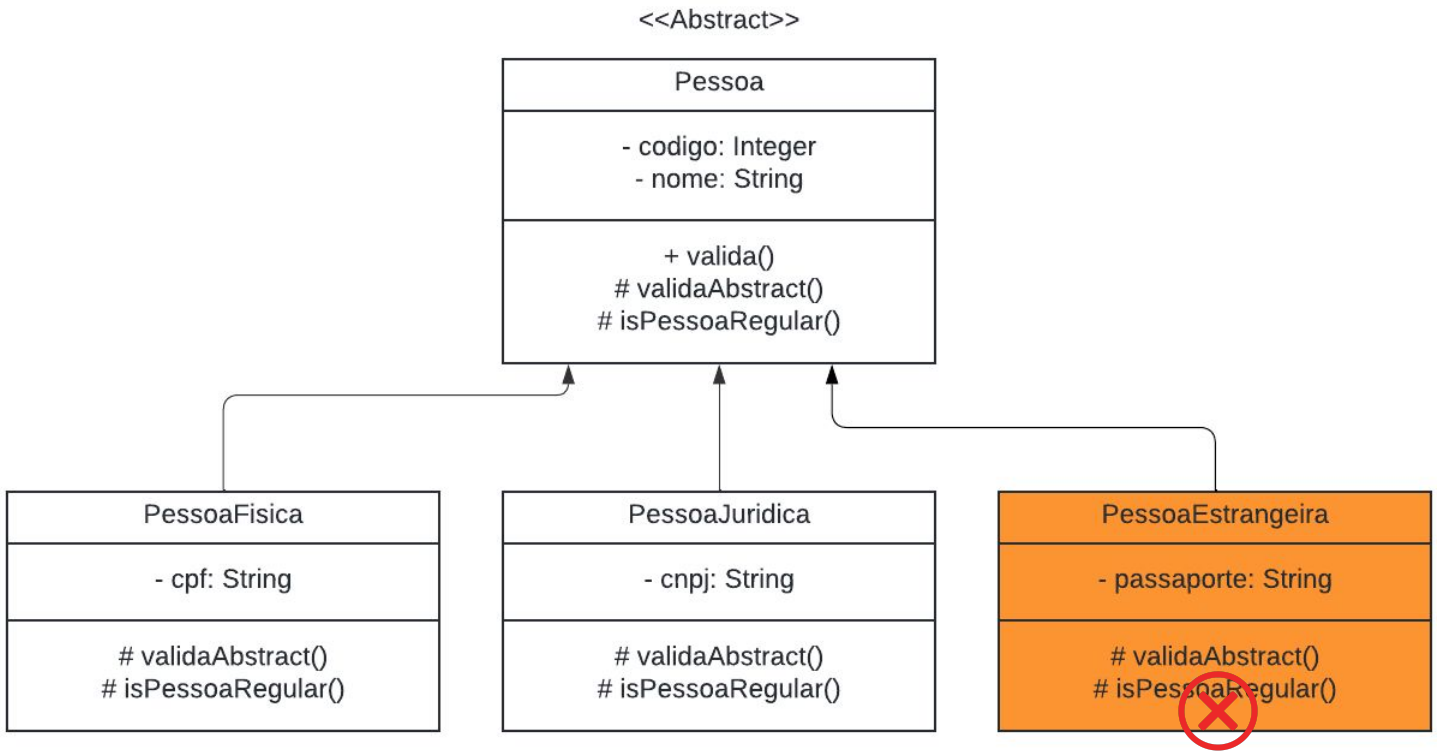
- Quando um sistema faz uso excessivo de instanceof, isso é um mau cheiro (ou code smell), ou seja, de que existe algum problema de design no código.
- O uso de excessivo de instanceof no geral
  - 1) Quebra o encapsulamento
  - 2) Destrói o polimorfismo
  - 3) Favorece o uso de ifs/elses espalhados pelo sistema
  - 4) Dificulta a evolução do software

INSTITUTO FEDERAL TRIÂNGULO MINEIRO Campus Uberlândia Centro

Interface

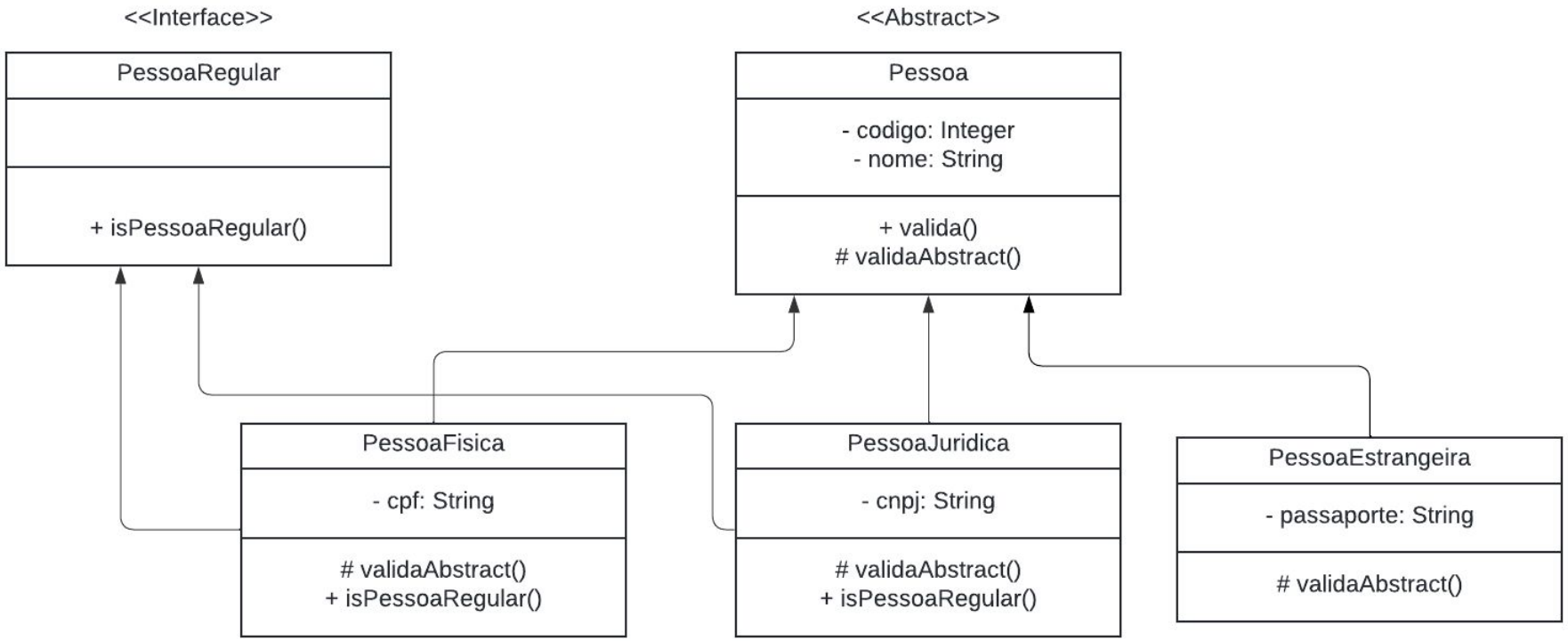
- É como se fosse uma classe, porém só pode conter
  - 1) Métodos de assinatura
  - 2) Métodos padrão
  - 3) Métodos estáticos
  - 4) Constantes

Interface

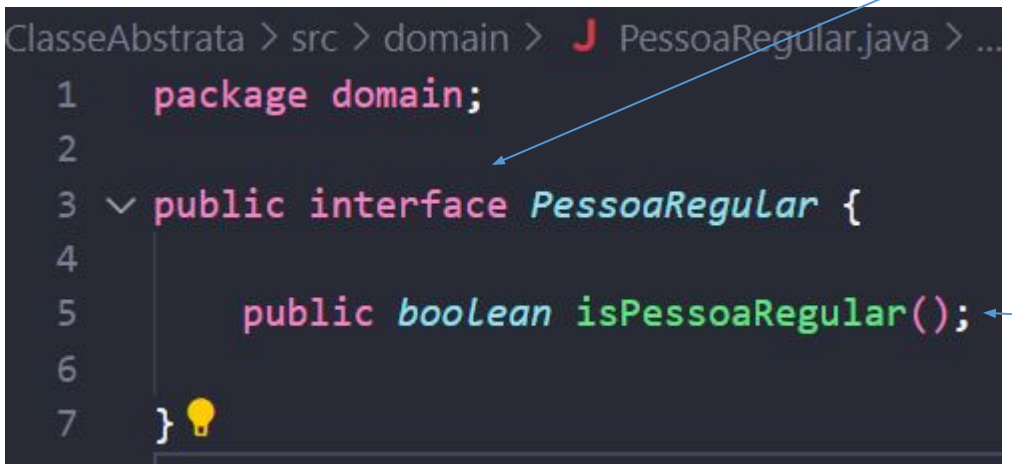

<<Abstract>>
Pessoa
- codigo: Integer
- nome: String
+ valida()
# validaAbstract()
# isPessoaRegular()
PessoaFisica
- cpf: String
validaAbstract()
isPessoaRegular()
PessoaJuridica
- cnpj: String
validaAbstract()
isPessoaRegular()
PessoaEstrangeira
- passaporte: String
validaAbstract()
isPessoaRegular()

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Interface



Interface



Definição de uma Interface

Todo método que implementar esta interface precisa também implementar o método isPessoaRegular()

Interface

PessoaFisica implementa a interface PessoaRegular

```java
package domain;

public class PessoaFisica extends Pessoa implements PessoaRegular {

    private String cpf;

    @Override
    protected boolean validaAbstract() {
        boolean validaCPF = true;

        return validaCPF;
    }

    @Override
    public boolean isPessoaRegular() {
        boolean chamaOSerasa = true;
        return chamaOSerasa;
    }

}
```

Veja que o @override de validaAbstract() diz respeito ao “pai” Pessoa.
Já o @override de isPessoaRegular() diz respeito ao “tio” PessoaRegular

```java
package domain;

public class PessoaJuridica extends Pessoa implements PessoaRegular {

    private String cnpj;

    @Override
    protected boolean validaAbstract() {
        boolean validaCnpj = true;

        return validaCnpj;
    }

    @Override
    public boolean isPessoaRegular() {
        boolean chamaAReceitaFederal = true;
        return chamaAReceitaFederal;
    }

}
```

PessoaEstrangeira não implementa a interface, logo não é “obrigada” a implementar isPessoaRegular()

```java
package domain;

public class PessoaEstrangeira extends Pessoa {

    private String passaporte;

    @Override
    protected boolean validaAbstract() {
        boolean validaPassaporte = true;

        return validaPassaporte;
    }

}
```

Interface

```java
public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        Pessoa carlos = new PessoaFisica();
        Pessoa iftm = new PessoaJuridica();
        Pessoa lewisHamilton = new PessoaEstrangeira();

        List<Pessoa> pessoas = Arrays.asList(carlos,iftm,lewisHamilton);
        List<PessoaRegular> pessoasRegulares = Arrays.asList((PessoaRegular)carlos,(PessoaRegular)iftm);

        for (Pessoa pessoa: pessoas) {
            chamaValidacao(pessoa);
        }

        for (PessoaRegular pessoaRegular: pessoasRegulares) {
            pessoaRegular.isPessoaRegular();
        }

    }

    private static void chamaValidacao(Pessoa pessoa) {
        pessoa.valida();
    }
}
```

Agora a classe App tem duas chamadas, uma para validar o básico de Pessoa, e o PessoaRegular.

O polimorfismo continua intacto com Interfaces, já que qualquer classe que implemente PessoaRegular pode ser chamada

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Mas ....

- Queríamos que validaAbstract() e isPessoaRegular() continuassem validando juntos, sem necessidade de ter duas listas. É possível?

```
3   public abstract class Pessoa {
4
5       private Integer codigo;
6
7       private String nome;
8
9       protected abstract boolean validaAbstract();
10
11      protected abstract boolean isPessoaRegular();
12
13      public boolean valida() {
14          boolean validouNome = true;
15
16          if (this instanceof PessoaEstrangeira) {
17              return validouNome && validaAbstract();
18          } else {
19              return validouNome && validaAbstract() && isPessoaRegular();
20          }
21
22
23      }
24
25
26  }
```

Vamos refatorar o código e excluir a herança para ver o resultado

```java
package domain;

public interface Validavel {

    public boolean valida();


    default boolean validaDefault(String nome) {
        boolean validouNome = true;

        return validouNome;
    }

}
```

Método que todas as classes deverão implementar na interface

Método default para validar o que é comum entre todas as classes

Vamos refatorar o código e excluir a herança para ver o resultado

Campos código e nome repetidos, e apenas implementando a interface Validavel.

Todos sendo obrigados a chamar o validaDefault() .... A vibe do super.valida() está de volta ☺

```java
public class PessoaFisica implements Validavel {

    private Integer codigo;

    private String nome;

    private String cpf;

    @Override
    public boolean valida() {
        boolean validaCPF = true;

        return validaCPF && validaDefault(nome) && isPessoaRegular();
    }

    public boolean isPessoaRegular() {
        boolean chamaOSerasa = true;
        return chamaOSerasa;
    }

}
```

```java
public class PessoaJuridica implements Validavel {

    private Integer codigo;

    private String nome;

    private String cnpj;

    @Override
    public boolean valida() {
        boolean validaCnpj = true;

        return validaCnpj && validaDefault(nome) && isPessoaRegular();
    }

    public boolean isPessoaRegular() {
        boolean chamaAReceitaFederal = true;
        return chamaAReceitaFederal;
    }

}
```

```java
public class PessoaEstrangeira implements Validavel {

    private Integer codigo;

    private String nome;

    private String passaporte;

    @Override
    public boolean valida() {
        boolean validaPassaporte = true;

        return validaPassaporte && validaDefault(nome);
    }

}
```

Vamos refatorar o código e excluir a herança para ver o resultado

```java
import java.util.Arrays;
import java.util.List;

import domain.PessoaEstrangeira;
import domain.PessoaFisica;
import domain.PessoaJuridica;
import domain.Validavel;

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        Validavel carlos = new PessoaFisica();
        Validavel iftm = new PessoaJuridica();
        Validavel lewisHamilton = new PessoaEstrangeira();

        List<Validavel> pessoas = Arrays.asList(carlos,iftm,lewisHamilton);

        for (Validavel pessoa: pessoas) {
            chamaValidacao(pessoa);
        }

    }

    private static void chamaValidacao(Validavel pessoa) {
        pessoa.valida();
    }
}
```

Não existe mais uma classe Pessoa, o que mantém os objetos em comum é a interface Validavel.

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Herança ou Interface?

- O ponto não é sobre qual é melhor, mas sim qual solução que tem menores chances de quebrar o encapsulamento do seu código.
- Se houver certeza de que a herança não quebrará o encapsulamento, pode usá-la.
- Entretanto, o futuro costuma ser incerto, e o acoplamento da herança é muito forte. A herança de Pessoa com PessoaFisica, PessoaJuridica e PessoaEstrangeira parece óbvia, mas em um design orientado a objetos, nem sempre o design é “tão óbvio assim” para se decidir por herança.

INSTITUTO FEDERAL
TRIÂNGULO MINEIRO
Campus Uberlândia Centro

Referências

- FOWLER, Martin – UML essencial: um breve guia para a linguagem padrão de modelagem de objetos / 3a ed: Bookman, 2005;
- - PRESSMAN, Roger S. Engenharia de Software: uma abordagem profissional. Porto Alegre, 2011;
- - PILONE, D., MILES, R. Use a Cabeça Desenvolvimento de Software. Altabooks, 2008;
- - FILHO, W. P. P. Engenharia de Software: Fundamentos, Métodos e Padrões. 3ª ed. Rio de Janeiro: LTC, 2009.