UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Herança e Composição – 1/2)

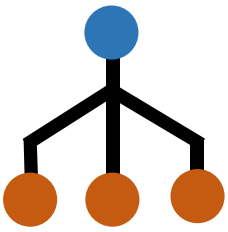
Prof. Luiz Chaimowicz
(slides adaptados do Prof. Douglas Macharet)

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

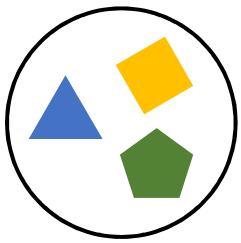
Introdução

Como criar uma relação entre diferentes tipos (classes)?

HERANÇA

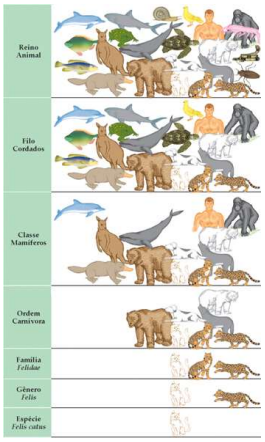


COMPOSIÇÃO



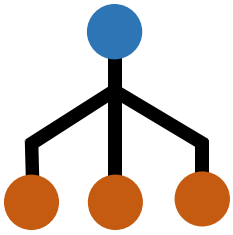
Herança

- Técnica para reutilizar características de uma classe na definição de outra(s) classe(s)
- Relação hierárquica (níveis) entre as classes
- Terminologias relacionadas à Herança
  - Classes mais genéricas: superclasses (pai)
  - Classes especializadas: subclasses (filha)

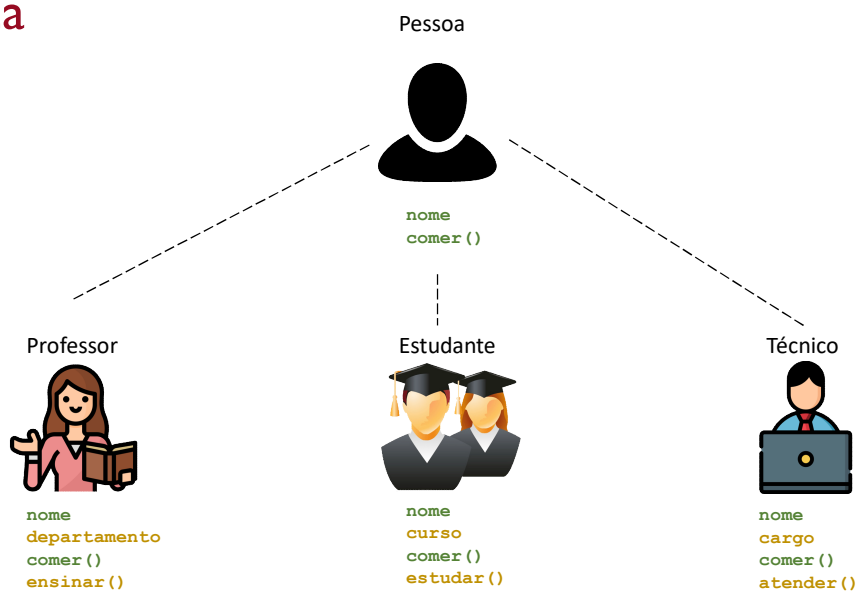


Herança

- Superclasses
  - Devem guardar membros em comum
- Subclasses
  - Acrescentam novos membros (estendem)
  - Redefinem comportamentos (especializam)



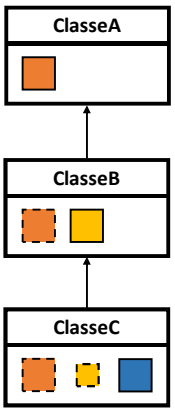
Herança



Herança

Generalização

Especialização



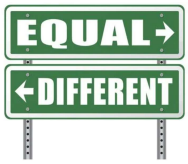
Podemos especializar de qualquer jeito?

NÃO!

Um subtipo deve preservar o significado (propósito) pelo qual o pai é conhecido.

Herança

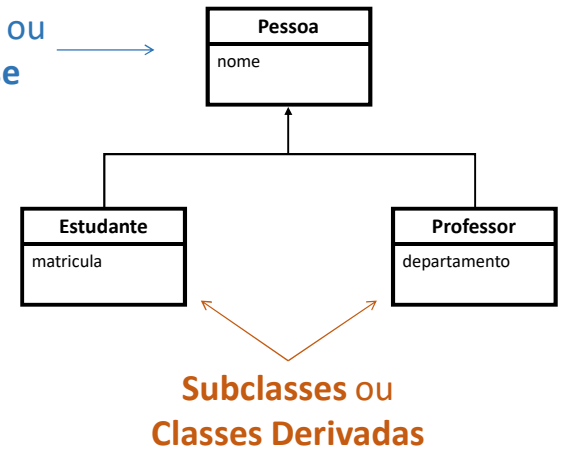
- Reutilização de código
  - Compartilhar similaridades
  - Preservar as diferenças



- Facilita a manutenção/extensão do sistema
  - Maior legibilidade do código existente
  - Quantidade menor de linhas de código
  - Alterações em poucas partes do código

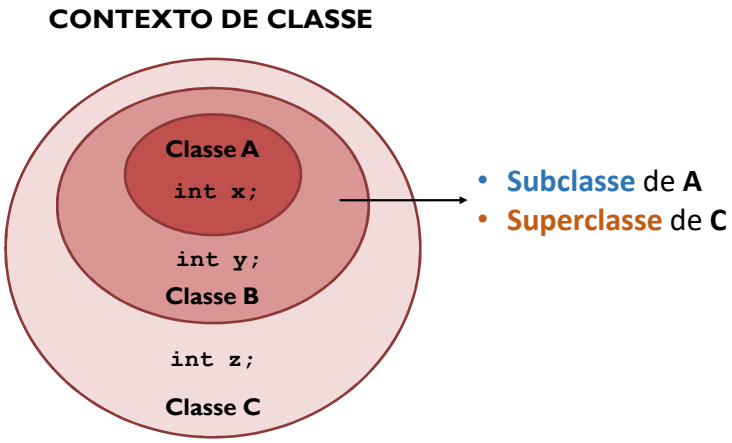
Herança simples
Hierarquia de Classes

Superclasse ou
Classe Base



Subclasses ou
Classes Derivadas

Herança Simples



Herança simples

```
class Pessoa {

  public:
    string nome;

};

class Estudante : public Pessoa {

  public:
    int matricula;

};
```

Também teremos um modificador de acesso aqui, que detalharemos em breve!

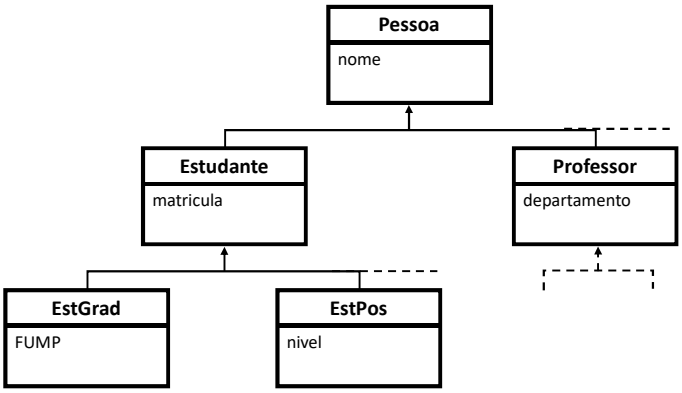
Subclasse

Superclasse

Herança simples

Na definição da herança, só é necessário especificar o primeiro nível acima na hierarquia.

```
class EstGrad : public Estudante
{

  public:
    bool FUMP;

};
```



Atenção: Herança Multinível é diferente de Herança Múltipla!

Herança simples

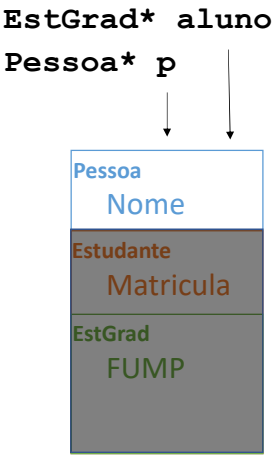
```
int main() {

  EstGrad* aluno = new EstGrad();

  aluno->nome = "Joao";
  aluno->matricula = 2019123456;
  aluno->FUMP = true;

  return 0;
}
```

EstGrad* aluno

Herança simples

```
int main() {

  EstGrad* aluno = new EstGrad();

  aluno->nome = "Joao";
  aluno->matricula = 2019123456;
  aluno->FUMP = true;

  Pessoa* p = aluno;
  p->nome = "Maria";
  p->matricula = 2022367891;
  p->FUMP = true;

   return 0;
}
```

Erro de Compilação!



Princípio da Substituição de Liskov (LSP)

Se S é um subtipo de T, então S deve poder ser usado em qualquer situação em que T seria usado, sem que isso afete a execução correta do programa.

- Contrato de uma classe
  - Encapsulamento → Interface
  - Coleção de atributos e métodos visíveis
- Subcontratação
  - Subclasse altera o contrato da superclasse (adiciona/especializa)
  - Contrato redefinido não pode violar o contrato da superclasse

https://en.wikipedia.org/wiki/Liskov_substitution_principle

Herança simples
Sobrescrita de métodos

- Métodos podem ser **sobrescritos** (override)
  - Diferente de sobrecarga (overload)!
  - Mesma assinatura da classe base
  - Substituir o método da base por implementação própria
  - Polimorfismo!

- Não restringir o acesso (quebra do LSP)
  - Public → Public
  - Protected → Protected, Public (mas isso tb quebra o LSP, não?)

Herança simples
Sobrescrita de métodos – Exemplo

```
class Pessoa {

  public:
    virtual void Imprime() {
      cout << "Sou uma PESSOA." << endl;
    }
};

class Estudante : public Pessoa {

  public:
    void Imprime() override {
      cout << "Sou um ESTUDANTE." << endl;
    }
};
```

É opcional, mas geralmente usado com métodos virtuais.

https://en.cppreference.com/w/cpp/language/override

Herança simples
Sobrescrita de métodos – Exemplo

Funcionaria sem virtual.

Virtual é necessário aqui. É uma Pessoa que se comporta como Estudante.

```
int main() {
  Pessoa p;
  p.Imprime();

  Estudante e;
  e.Imprime();

  Pessoa* p2 = new Estudante();
  p2->Imprime();

  delete p2;

  return 0;
}
```

Sou uma PESSOA.

Sou um ESTUDANTE.

Sou um ESTUDANTE.

Wandbox

Herança simples
Sobrescrita de métodos

- Métodos virtuais
  - Resolvidos dinamicamente (late/dynamic binding)
  - Apenas em tempo de execução que sabemos exatamente qual função será chamada, de acordo com o objeto na memória (portanto só funciona com apontadores)
  - Será mais detalhado ao vermos Polimorfismo!
- Dessa forma, o comportamento base pode ser sobrescrito nas classes derivadas respeitando completamente o LSP

https://en.cppreference.com/w/cpp/language/virtual

Princípio da Substituição de Liskov (LSP)

```
class CoffeeMachine {
  public:
    void virtual makeCoffee() {
        cout << "Making regular coffee." << endl;
    }
};

class PremCoffeeMachine : public CoffeeMachine {
  public:
    void makeCoffee() override {
        cout << "Making capuccino." << endl;
    }
};
```

```
void remoteControl(CoffeeMachine* cm) {
    cm->makeCoffee();
}
```

```
int main() {

  CoffeeMachine *cm = new CoffeeMachine();
  PremCoffeeMachine *pcm = new PremCoffeeMachine();

  remoteControl(cm);
  remoteControl(pcm);

  return 0;
}
```

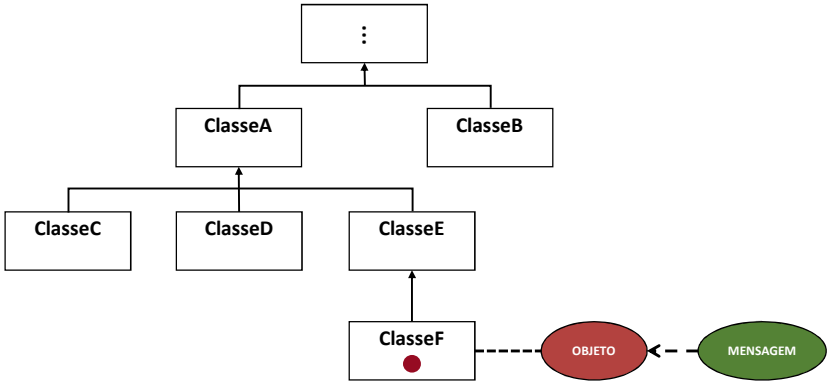
```
Making regular coffee.
Making capuccino.
```

Pode-se entregar um tipo específico de café, mas será que poderia ser entregue um suco?

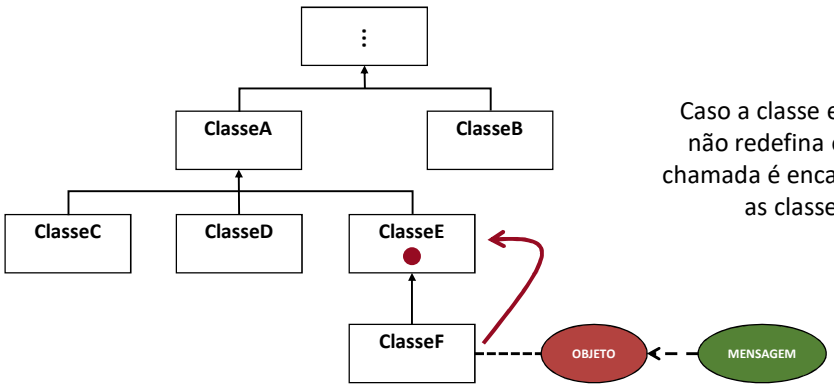
Herança simples
Sobrescrita de métodos

- Não podem ser sobrescritos/redefinidos
  - Atributos
    - Definição na superclasse será ocultada (substituída)
  - Métodos private
    - Não são acessíveis/visíveis fora do escopo (base)
  - Membros estáticos
    - Também serão ocultados, mas como o acesso é feito pelo nome da classe, estar ou não ocultado terá pouco efeito (continua acessível)

Herança simples
Sobrescrita de métodos

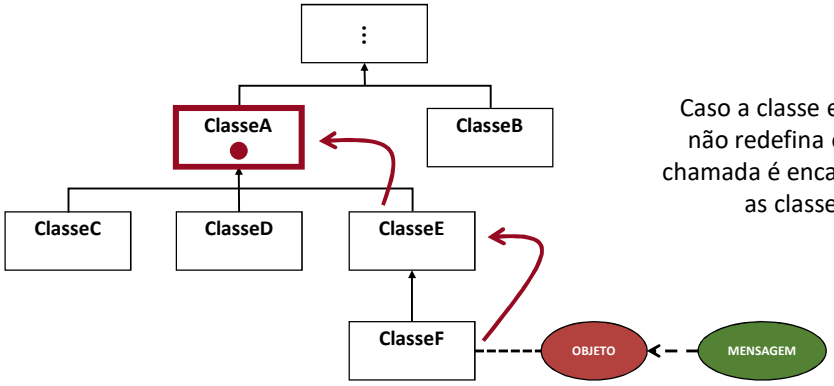


Herança simples
Sobrescrita de métodos



Caso a classe especializada não redefina o método, a chamada é encaminhada para as classes base

Herança simples
Sobrescrita de métodos



Caso a classe especializada não redefina o método, a chamada é encaminhada para as classes base

Herança e Encapsulamento
Modificadores de acesso (C++)

- O modificador de acesso usado para <u>declarar a superclasse</u> determina o nível de acesso aos seus membros na subclasse
- **Public** ⬅
  - Mantém os níveis da classe base
- Protected
  - Public e Protected ⇒ Protected
- Private
  - Public e Protected ⇒ Private

	TIPO DE HERANÇA (DERIVADA)		
ESCOPO (BASE)	`public`	`protected`	`private`
`public`	Public	Protected	Private (acessível)
`protected`	Protected	Protected	Private (acessível)
`private`	Private (não acessível)	Private (não acessível)	Private (não acessível)

Herança
Críticas

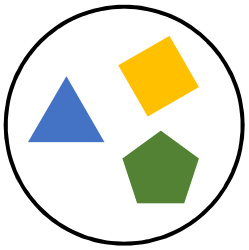
- "Fere" o princípio do encapsulamento
  - Membros fazem parte de várias classes
- Dependência estrutural grande entre classes
  - Mudanças em superclasses podem ser difíceis
- Como resolver isso?

> Composition is often more appropriate than inheritance.
> When using inheritance, make it public.
>
> – Google C++ Style Guide

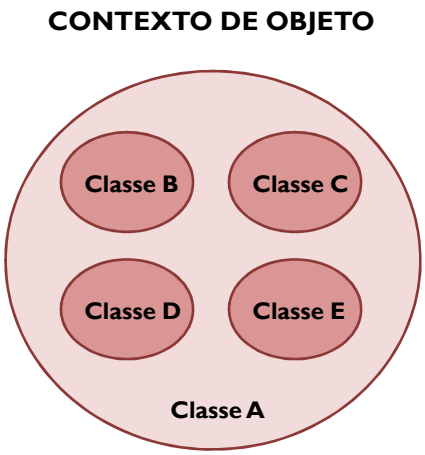
Composição

Composição

- Criar um novo tipo não pela derivação, mas pela junção de outras classes de menor complexidade
- Conceito lógico de agrupamento
  - Modo particular de implementação
  - Não existe palavra-chave ou recurso
- Princípio do encapsulamento
  - Acesso apenas à parte pública das classes
- Menor interdependência entre classes

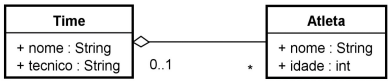


Composição

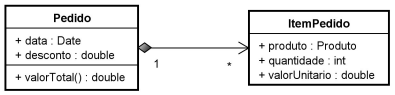


Composição
Modelagem

- Agregação
  - Relação do todo ser formado por partes
  - "Parte" pode estar em diferentes "Todos"
  - "Parte" existe mesmo sem o "Todo"



- Composição
  - "Parte" pertence a um único objeto "Todo"
  - "Parte" **não** faz sentido sem o "Todo"
  - Remoção do todo → Remoção das partes



https://en.wikipedia.org/wiki/Object_composition

Herança vs. Composição

- Herança
  - Relação do tipo "é um" (is-a)
  - Subclasse tratada como a superclasse (LSP)
  - Estudante **é uma** Pessoa
- Composição
  - Relação do tipo "tem um" (has-a)
  - Objeto incorpora objetos (≥ 1) de outras classes
  - Estudante **tem um** Curso

Herança vs. Composição

```
class Curso {

  public:
    string nome;
    int creditos;

};

class Estudante : public Pessoa {

  public:
    Curso* curso;

};
```

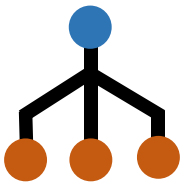
Herança

Composição

Herança vs. Composição

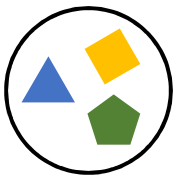
- Como escolher/saber qual utilizar?

HERANÇA



O TipoB vai manter todo o contrato do TipoA, e poderá ser usado onde o TipoA é esperado?

COMPOSIÇÃO



O TipoB deseja apenas utilizar parte do comportamento exposto pelo TipoA?

Herança vs. Composição
Boas práticas e recomendações – Herança

- Se possível, mova interfaces, dados e comportamentos comuns (gerais) para os níveis mais altos da hierarquia
- Suspeite de classes base com apenas uma classe derivada
- Evite hierarquias muito profundas (rigidez)
- Atenção ao encapsulamento (protected)

Herança vs. Composição
Boas práticas e recomendações – Composição

- Seja crítico em relação número de membros
- "7 ± 2" [Miller, 1956]
  - Estimativa para o número de itens que uma pessoa consegue lembrar enquanto está executando outras tarefas
- Evitar classes "super-agregadoras"
- Decompor a classe em outras menores

Exercício

- Modelar um automóvel
  - Quais atributos devem existir?
  - Quais métodos devem existir?
- Onde usar Herança/Composição?

Exercício

```
class Motor {
  public:
    void injetar_gasolina() {
        cout << "Injetando gasolina." << endl;
    };
};
```

```
class Carro {
  private:
    int _num_portas;
    Motor _motor;

  public:
    Carro(int num_portas) : _num_portas(num_portas) {}

    void acelerar() {
      this->acionar_motor();
    }

    void frear() {
      this->acionar_discos_freio();
    }

    int get_num_portas() {
        return this->_num_portas;
    }

  private:
    void acionar_motor() {
        this->_motor.injetar_gasolina();
    }

    void acionar_discos_freio() {
        cout << "Acionando discos de freio." << endl;
    };
};
```

Carro **tem** um motor.

Comportamentos específicos de Carro.

Invocando um comportamento externo pelo contrato (interface pública).

Exercício

```
int main() {

  Carro ferrari(2);
  cout << ferrari.get_num_portas() << endl;
  ferrari.acelerar();
  ferrari.frear();

  Carro bmw(4);
  cout << bmw.get_num_portas() << endl;
  bmw.acelerar();
  bmw.frear();

  return 0;
}
```

```
2
Injetando gasolina.
Acionando discos de freio.

4
Injetando gasolina.
Acionando discos de freio.
```


Exercício

```
class Turbina {
  public:
    void ligar_turbo() {
        cout << "Ligando turbo." << endl;
    };
};
```

BatMovel **é** um Carro.

```
class BatMovel : public Carro {
  private:
    Turbina _turbina;

  public:
    BatMovel() : Carro(1) {}

    void acelerar_turbo() {
        this->acelerar();
        this->acionar_turbina();
    }

    void frear() {
        cout << "Acionando freios especiais." << endl;
    }

  private:
    void acionar_turbina() {
        this->_turbina.ligar_turbo();
    }
};
```

A parte relativa a CARRO também precisa ser inicializada!

Extensão/ Especialização

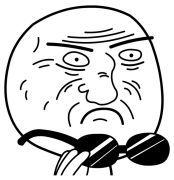
Exercício

```
int main() {

  BatMovel batmovel;
  cout << batmovel.get_num_portas() << endl;
  batmovel.acelerar();
  batmovel.acelerar_turbo();
  batmovel.frear();

  return 0;
}
```

```
1
Injetando gasolina.
Injetando gasolina.
Ligando turbo.
Acionando freios especiais.
```


Exercício

Esse código não compila!



```
int main() {

  Carro *batmovel2 = new BatMovel();
  cout << batmovel2->get_num_portas() << endl;
  batmovel2->acelerar();
  batmovel2->acelerar_turbo();
  batmovel2->frear();

  return 0;
}
```

Erro!

E qual será esse comportamento?

Wandbox

Exercício

- Tarefas
  - Faça a modularização de todas as classes
  - Adicione membros com diferentes níveis de acesso
  - Corrija os problemas existentes (virtual, desalocação)
  - Crie uma nova subclasse com composição/herança

Considerações finais

- **Reuso**
  - Escrever código em comum uma vez apenas

- **Extensão**
  - Adicionar novas responsabilidades (membros)

- **Especialização**
  - Redefinir responsabilidades já existentes

Considerações finais

http://vidadeprogramador.com.br/2013/11/06/heranca/