

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Programação Orientada a Objetos (Herança e Composição)

Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

Introdução

- Técnica para reutilizar características de uma classe na definição de outra classe
- Determinação de uma hierarquia de classes
- Terminologias relacionadas à Herança
 - Classes mais genéricas: superclasses (pai)
 - Classes especializadas: subclasses (filha)

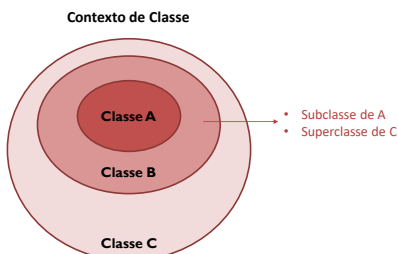
Introdução

- Superclasses
 - Devem guardar membros em comum
- Subclasses
 - Acrescentam novos membros (especializam)
- Componentes facilmente reutilizáveis
- Facilita a extensibilidade do sistema

Herança

- Os atributos e métodos são herdados por todos os objetos dos níveis mais baixos
 - Considerando o modificador de acesso
 - Membros private "herdados", mas não acessíveis pois pertencem apenas ao escopo da classe
- Diferentes subclasses podem herdar as características de uma (ou mais) superclasse

Herança



Herança

Benefícios

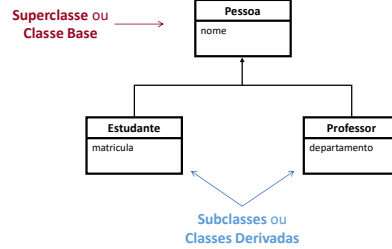
- Reutilização de código
 - Compartilhar similaridades
 - Preservar as diferenças
- Facilita a manutenção do sistema
 - Maior legibilidade do código existente
 - Quantidade menor de linhas de código
 - Alterações em poucas partes do código

Herança

Árvore de herança

- O conjunto de classes que herdam entre si recebe o nome de árvore de herança
 - Hierarquia de classes relacionadas
- Cada subclasse pode possuir
 - Uma única superclasse (Java)
 - Herança simples
 - Várias superclasses (C++)
 - Herança múltipla

Herança simples



Herança simples

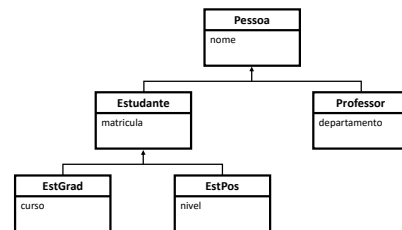
```

class Pessoa {
public:
    string nome;
};

class Estudante : public Pessoa {
public:
    int matricula;
};
  
```

→ Superclasse

Herança simples



Herança simples

Classe: Estudante	
Superclasses: Pessoa	
Subclasses: EstGrad, EstPos	
Responsabilidades	Colaborações

Herança simples

```

class EstGrad : public Estudante {
public:
    string curso;
};
  
```

Herança simples

```
int main() {
    EstGrad* aluno = new EstGrad();

    aluno->nome = "Joao";
    aluno->matricula = 2019123456;
    aluno->curso = "Computacao";

    return 0;
}
```

EstGrad	
Pessoa	
Nome	
Estudante	
Matricula	
EstGrad	
Curso	

Herança simples

Construtores e Destrutores

- A classe derivada executa o construtor da classe base ANTES de executar o próprio
 - Chamado mesmo que implicitamente (padrão)
 - Pode estar explícito na lista de inicialização
- A classe derivada executa o destrutor da classe base DEPOIS de executar o próprio

Herança simples

Exemplo 1

```
class A {
public:
    A() { cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
};

class B : public A {
public:
    B() { cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
};

class C : public B {
public:
    C() { cout << "C()" << endl; }
    ~C() { cout << "~C()" << endl; }
};
```

Herança simples

Exemplo 1

```
int main() {
    cout << "Alocando B:" << endl;
    B b1;

    cout << "Alocando C:" << endl;
    C* c1 = new C();

    cout << "Deleting C:" << endl;
    delete c1;

    cout << "Quitting..." << endl;
    return 0;
}
```

<https://wandbox.org/pemlink/G8ty8q9MePTKJf>

Saída:

```
Alocando B:
A()
B()
Alocando C:
A()
B()
C()
Deleting C:
~C()
~B()
~A()
Quitting...
~B()
~A()
```

Herança simples

Exemplo 2

```
class A {
    int _a;
public:
    A(int a) : _a(a) {}
    void getAtributo() { cout << _a << endl; }
};

class B : public A {
    int _b;
public:
    B(int a, int b) : A(a), _b(b) {}
};
```

```
int main() {
    A objA(55);
    B objB(77, 99);
    objB.getAtributo();

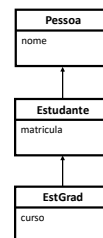
    return 0;
}
```

<https://wandbox.org/pemlink/huzz7CnrtWQiehd1>

Herança simples


Generalização

Especialização



Herança e Encapsulamento

C++

- Modificadores de acesso na herança
- Public 
 - Mantém os níveis de acesso da classe base
- Protected
 - Public e Protected \Rightarrow Protected
- Private
 - Public e Protected \Rightarrow Private

DCC 

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

19

Herança simples

Sobrescrita de métodos

- Métodos podem ser sobrescritos (overriding)
 - Diferente de sobrecarga!
 - Mesma assinatura e tipo de retorno (!)
 - Métodos private não são sobrescritos
 - Devem ser definidos como 'virtuais'
- Não restringir o acesso (quebra do LSP)
 - Public \rightarrow Public
 - Protected \rightarrow Protected, Public

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

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

20

Herança simples

Sobrescrita de métodos

- Atributos não são redefiníveis
 - Se atributo de mesmo nome for definido na subclasse, a definição na superclasse é ocultada
- Membros estáticos
 - Não são redefinidos, mas ocultados
 - Como o acesso é feito pelo nome da classe, estar ou não ocultado terá pouco efeito

DCC 

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

21

Herança simples

Sobrescrita de métodos

- Métodos virtuais
 - Resolvidos dinamicamente
 - Apenas em tempo de execução que sabemos exatamente qual função deverá ser chamada
 - Será mais detalhado ao vermos Polimorfismo
- Dessa forma, o comportamento base pode ser sobrescrito em classes derivadas

<https://en.cppreference.com/w/cpp/language/virtual>DCC 

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

22

Herança simples

Sobrescrita de métodos – Exemplo

```
class Pessoa {
public:
    virtual void meuNome() {
        cout << "Meu nome é PESSOA." << endl;
    }
};

class Estudante : public Pessoa {
public:
    void meuNome() override {
        cout << "Meu nome é ESTUDANTE." << endl;
    }
};
```

<https://en.cppreference.com/w/cpp/language/override>DCC 

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

23

Herança simples

Sobrescrita de métodos – Exemplo

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

    Estudante e;
    e.meuNome();

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

    delete p2;

    return 0;
}
```

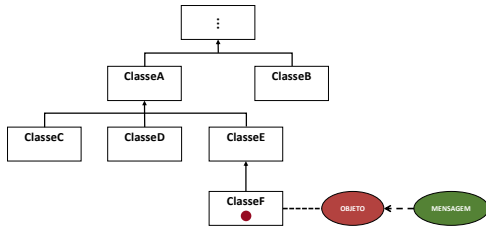
<https://wandbox.org/pemfmlk/4D7N6/p72ZRGb4IX>DCC 

PD5 2 - Programação Orientada a Objetos (Herança e Composição)

24

Herança simples

Sobrescrita de métodos



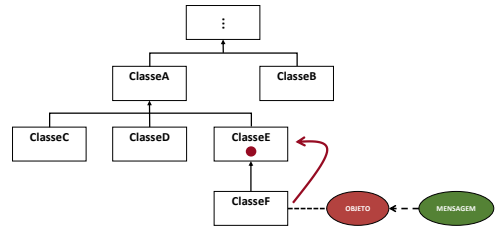
DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

25

Herança simples

Sobrescrita de métodos



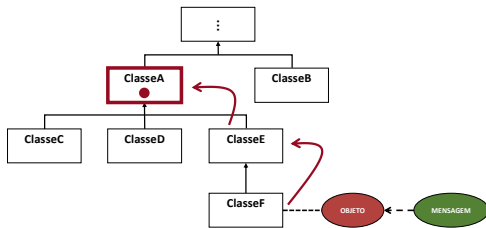
DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

26

Herança simples

Sobrescrita de métodos



DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

27

Herança múltipla

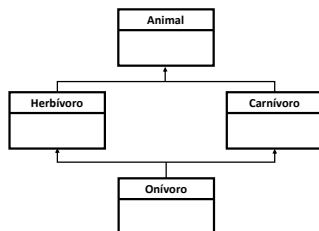
- Subclasse herda de mais de uma superclasse
 - Algumas linguagens permitem isso!
- Problemas
 - Dificulta a manutenção do sistema
 - Também dificulta o entendimento
 - Reduz a modularização (super objetos)
 - Classes que herdam de todo mundo
 - “Saída do preguiçoso”

DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

28

Herança múltipla


https://en.wikipedia.org/wiki/Multiple_inheritance#The_diamond_problem

DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

29

Herança múltipla

```
class Animal {
};

class Herbivoro : public Animal {
};

class Carnivoro : public Animal {
};

class Onivoro : public Herbivoro, public Carnivoro {
};
```

DCC

PDS 2 - Programação Orientada a Objetos (Herança e Composição)

30

Herança

Críticas

- “Fere” o princípio do encapsulamento
 - Membros fazem parte de várias classes
- Cria interdependência 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

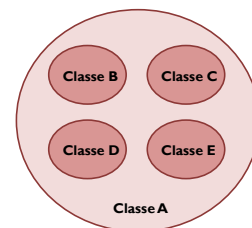
- Técnica para criar um novo tipo não pela derivação, mas pela junção de outras classes de menor complexidade
- Conceito lógico de agrupamento
 - Modo particular de implementação
 - Não existe palavra-chave ou recurso

Herança vs. Composição

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

Composição

Contexto de Objeto



Composição

```
class Curso {
public:
    string nome;
    int credits;
};

class EstGrad : public Estudante {
public:
    Curso* curso;
};
```

Exemplo

```
Alpha.hpp
#ifndef ALPHA_H
#define ALPHA_H
#include <iostream>

class Alpha {
private:
    int _atributoPrivadoAlpha;
protected:
    int _atributoProtectedAlpha;
public:
    int atributoPublicoAlpha;
    Alpha(int atPrivado, int atProtected, int atPublico);
    void metodoA();
    virtual void metodoB();
};
#endif
```

Exemplo

```
Alpha.cpp
#include "Alpha.hpp"

Alpha::Alpha(int atPrivado, int atProtected, int atPublico) :
    _atributoPrivadoAlpha(atPrivado), _atributoProtectedAlpha(atProtected),
    atributoPublicoAlpha(atPublico) {}

void Alpha::metodoA() {
    std::cout << "Alpha:metodoA()" << std::endl;
}

void Alpha::metodoB() {
    std::cout << "Alpha:metodoB()" << std::endl;
}
```

Exemplo

```
AlphaSub.hpp
#ifndef ALPHASUB_H
#define ALPHASUB_H

#include "Alpha.hpp"
#include "Gamma.hpp"

class AlphaSub : public Alpha {
private:
    int _atributoPrivadoAlphaSub;
protected:
    int _atributoProtectedAlphaSub;
public:
    int atributoPublicoAlphaSub;
    Gamma* atributoGamma;

    AlphaSub(int atPrivadoAlpha, int atProtectedAlpha, int atPublicoAlpha,
            int atPrivadoAlphaSub, int atProtectedAlphaSub, int atPublicoAlphaSub,
            Gamma* atributoGamma);

    void metodoB() override;
    void metodoC();
};

#endif
```

Exemplo

```
AlphaSub.cpp
#include "AlphaSub.hpp"

AlphaSub::AlphaSub(int atPrivadoAlpha, int atProtectedAlpha, int atPublicoAlpha,
    int atPrivadoAlphaSub, int atProtectedAlphaSub, int atPublicoAlphaSub,
    Gamma* atributoGamma) : Alpha(atPrivadoAlpha, atProtectedAlpha, atPublicoAlpha),
    _atributoPrivadoAlphaSub(atPrivadoAlphaSub), _atributoProtectedAlphaSub(atProtectedAlphaSub),
    atributoPublicoAlphaSub(atPublicoAlphaSub), atributoGamma(atributoGamma) {}

void AlphaSub::metodoB() {
    std::cout << "AlphaSub:metodoB()" << std::endl;
}

void AlphaSub::metodoC() {
    std::cout << "AlphaSub:metodoC()" << std::endl;
}
```

Precisa construir a
parte de Alpha antes!

Exemplo

```
Gamma.hpp
#ifndef GAMMA_H
#define GAMMA_H

#include <iostream>

class Gamma {
private:
    int _atributoPrivadoGamma;
protected:
    int _atributoProtectedGamma;
public:
    int atributoPublicoGamma;

    Gamma(int atPrivado, int atProtected, int atPublico);

    void metodoD();
};

#endif
```

Exemplo

```
Gamma.cpp
#include "Gamma.hpp"

Gamma::Gamma(int atPrivado, int atProtected, int atPublico) :
    _atributoPrivadoGamma(atPrivado), _atributoProtectedGamma(atProtected),
    atributoPublicoGamma(atPublico) {}

void Gamma::metodoD() {
    std::cout << "Gamma:metodoD()" << std::endl;
}
```

Exemplo

```
main.cpp
#include "Alpha.hpp"
#include "AlphaSub.hpp"
#include "Gamma.hpp"

int main() {
    Alpha alpha(10, 10, 10);
    alpha.metodoA();
    alpha.metodoB();

    Gamma gamma(10, 30, 30);
    AlphaSub alphaSub(15, 15, 15, 20, 20, 20, &gamma);

    alphaSub.metodoA();
    alphaSub.metodoB();
    alphaSub.metodoC();

    alphaSub.atributoGamma->metodoD();

    return 0;
}
```

<https://www.w3schools.com/permissions/permissions.asp>

Considerações finais

- Reuso
 - Escreva código em comum uma vez apenas
- Extensão
 - Adicione novas responsabilidades (membros)
- Especialização
 - Redefina responsabilidades existentes
 - Classe Base → Classe Derivada

Comentários finais

