Programação Orientada a Objetos com C++

Leonardo Medeiros

Maio de 2024

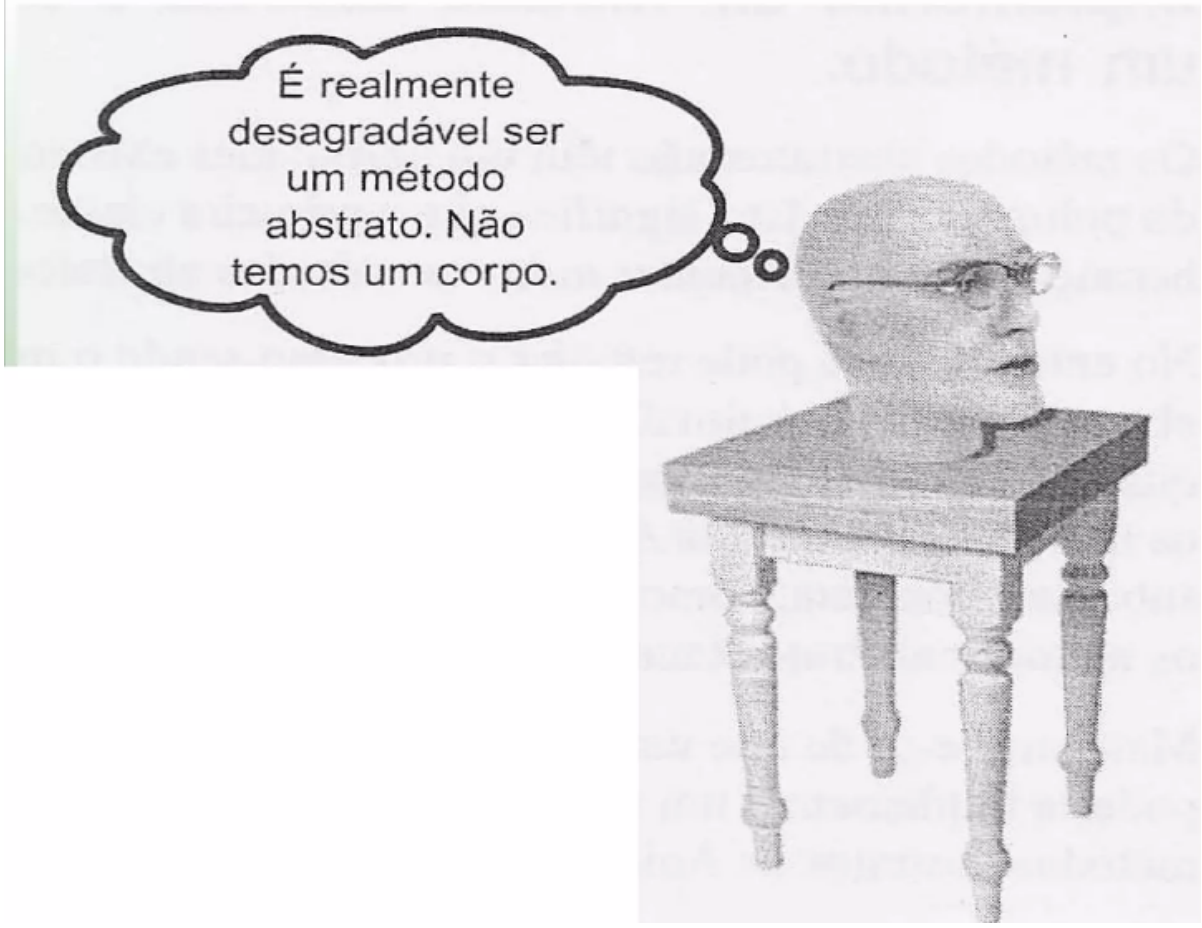
Objetivos de Aprendizado

- Reconhecer o conceito de estruturas abstratas de programação e suas funcionalidades aplicando-as com a linguagem C++

Classes Abstratas

- Usamos a palavra chave virtual para impedir a instanciação de uma classe.
- Classes abstratas normalmente possuem um ou mais métodos abstratos.
- Um método abstrato não possui corpo, apenas define a assinatura (protótipo) do método.
- Classes abstratas são úteis quando uma classe define métodos a serem implementados apenas pelas suas subclasses.

Classes Abstratas



Classes Abstratas

- Regras sobre métodos abstratos e as classes que os contém:
  - Toda classe que possui métodos abstratos é automaticamente abstrata.
  - Uma classe abstrata não pode ser instanciada.
  - Uma subclasse de uma classe abstrata pode ser instanciada se:
    - Sobrescreve cada um dos métodos abstratos de sua superclasse
    - Fornece uma implementação para cada um deles

Classes Abstratas

- Se uma subclasse de uma classe abstrata não implementa todos os métodos abstratos que ela herda, então a subclasse também é abstrata.
- Para declarar um método abstrato, coloque a palavra-chave virtual (*declarator*) no início do protótipo do método e coloque "$= 0$" (*pure-specifier*) no fim.
- Para declarar uma classe abstrata, coloque pelo menos um método abstrato na classe.

Exemplo de Classe Abstrata: Forma

```
#include <iostream>
using namespace std;

class Forma {
   protected:
      int largura;
      int comprimento;
   public:
      // método abstrato
      virtual int getArea() = 0;

      void setLargura(int l) {
         largura = l;
      }

      void setComprimento(int c) {
         comprimento = c;
      }

};
```

Exemplo de Classe Abstrata: Forma (II)

```
// Classes derivadas
class Retangulo: public Forma {
   public:
      int getArea() {
         return (largura * comprimento);
      }
};

class Triangulo: public Forma {
   public:
      int getArea() {
         return (largura * comprimento)/2;
      }
};

int main(void) {
   Retangulo Ret;
   Triangulo Tri;

   Ret.setLargura(5);
   Ret.setComprimento(7);
```

Exemplo de Classe Abstrata: Forma (III)

```
    cout << "A área total do retângulo: " << Ret.getArea() << endl;

    Tri.setLargura(5);
    Tri.setComprimento(7);

    cout << "A área total do triângulo: " << Tri.getArea() << endl;

    return 0;
}
```

Interfaces

- Problema
  - Suponha que você define as classes abstratas Forma e Desenhavel e a classe Circulo (que estende Forma)
  - Como criar a classe CirculoDesenhavel que é ao mesmo tempo Circulo e Desenhavel?

```
class CirculoDesenhavel: public Circulo, Desenhavel {...};
```

- Apesar de C++ suportar herança múltipla, usaremos interfaces.
- No C++, classe abstrata e interface possuem o mesma sintaxe (virtual).
- No Java, são conceitos separados (abstract e interfaces).

Interfaces

- Interface é um contrato onde quem assina se responsabiliza por implementar os métodos definidos na interface (cumprir contrato).
- Ela só expõe o que o objeto deve fazer, e não como ele faz, nem o que ele tem.
- Como ele faz vai ser definido em uma implementação dessa interface.

Interfaces

- Quais as diferenças entre interface e classe abstrata?
  - Declaração de métodos
    - Em uma classe abstrata podem ser definidos métodos abstratos e não-abstratos
    - Todos os métodos de uma interface são abstratos
  - Declaração de variáveis
    - Em uma classe abstrata podem ser definidas variáveis de instância, de classe e constantes
    - Todos os atributos de uma interface são sempre constantes, mesmo que não sejam usadas.

Sintaxe de Interfaces

```
class class_name {
  public:
  // pure virtual function
  virtual return-type func_name() = 0;
};
```

Exemplo de Interfaces: Caixa

```
class Caixa {
   private:
      double largura;
      double altura;
      double comprimento;
   public:
      // método abstrato (pure virtual function)
      virtual double getVolume() = 0;

};
```

Exemplo de Interfaces: Forma

```
#include <iostream>
using namespace std;
class Forma{
    public:
    virtual void desenho()=0;
};
class Retangulo : Forma {
    public:
     void desenho(){
        cout <<"desenhando o retângulo..." <<endl;
    }
};
class Circulo : Forma {
    public:
     void desenho()
    {
        cout <<"desenhando o círculo..." <<endl;
    }
};
```

Exemplo de Interfaces: Forma (II)

```
int main( ) {
   Retangulo ret;
   Circulo cir;
   ret.desenho();
   cir.desenho();
   return 0;
}
```

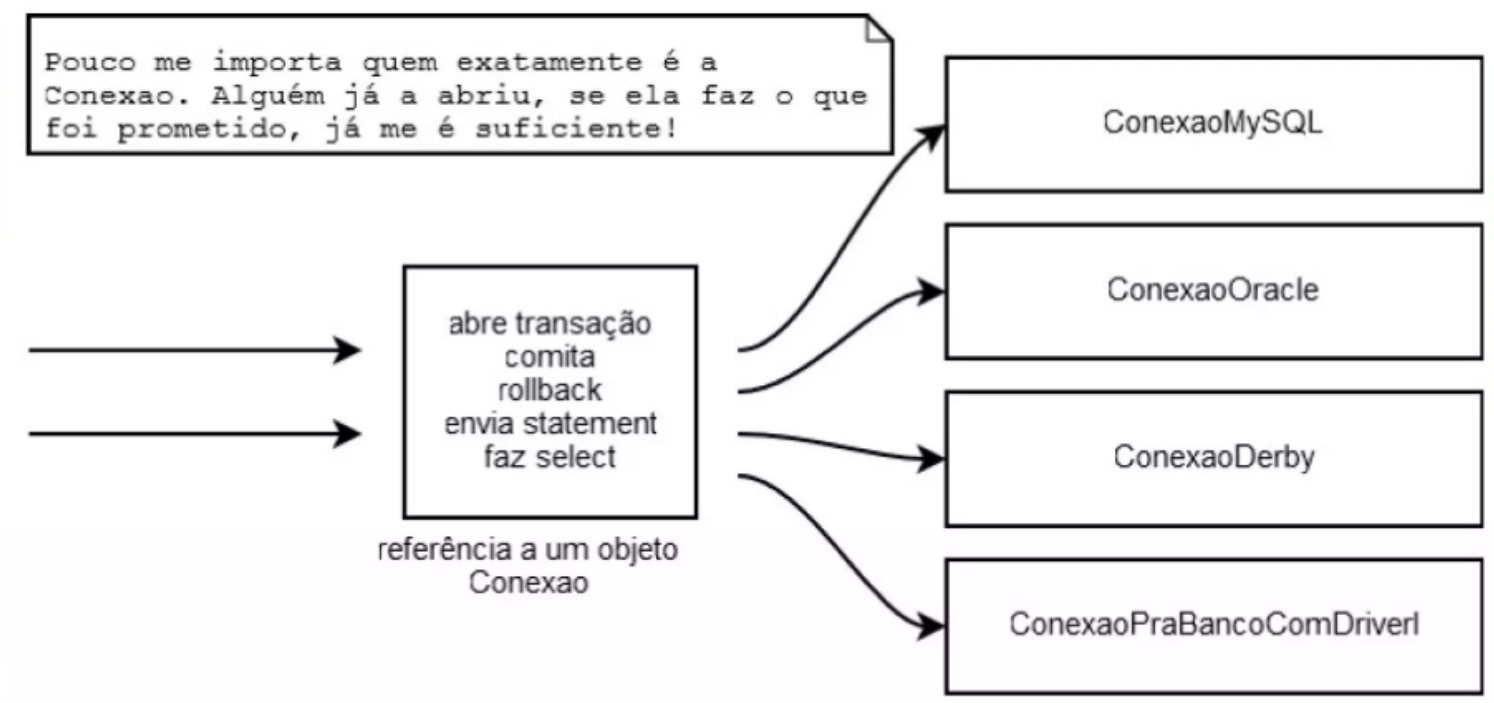
Herança entre Interfaces

- Uma interface pode herdar de mais de uma interface.
- É como um contrato que depende de que outros contratos sejam fechados antes deste valer.
- Você não herda métodos e atributos, e sim responsabilidades.

Pra quê usar Interfaces

- Para deixar o código mais flexível, e possibilitar a mudança de implementação sem maiores traumas.
- Elas também trazem vantagens em não acoplar as classes.
  - Uma vez que herança através de classes traz muito acoplamento, muitos autores clássicos dizem que em muitos casos herança quebra o encapsulamento.

Interfaces


Pouco me importa quem exatamente é a Conexao. Alguém já a abriu, se ela faz o que foi prometido, já me é suficiente!
abre transação
comita
rollback
envia statement
faz select
referência a um objeto Conexao
ConexaoMySQL
ConexaoOracle
ConexaoDerby
ConexaoPraBancoComDriverI

Regras de Ouro

- Do livro Design Patterns:
  - "Evite herança, prefira composição"
  - "Programe voltado a interface e não a implementação"

Exemplo: Mesas

- Queremos modelar mesas em termos de objetos:
  - Uma mesa
  - Uma mesa com rodinhas
  - Uma mesa de tênis de mesa com rodinhas
- A aplicação pode em algum momento usar a mesa pura, em outra com rodinhas, em outra com rede de tênis de mesa,...

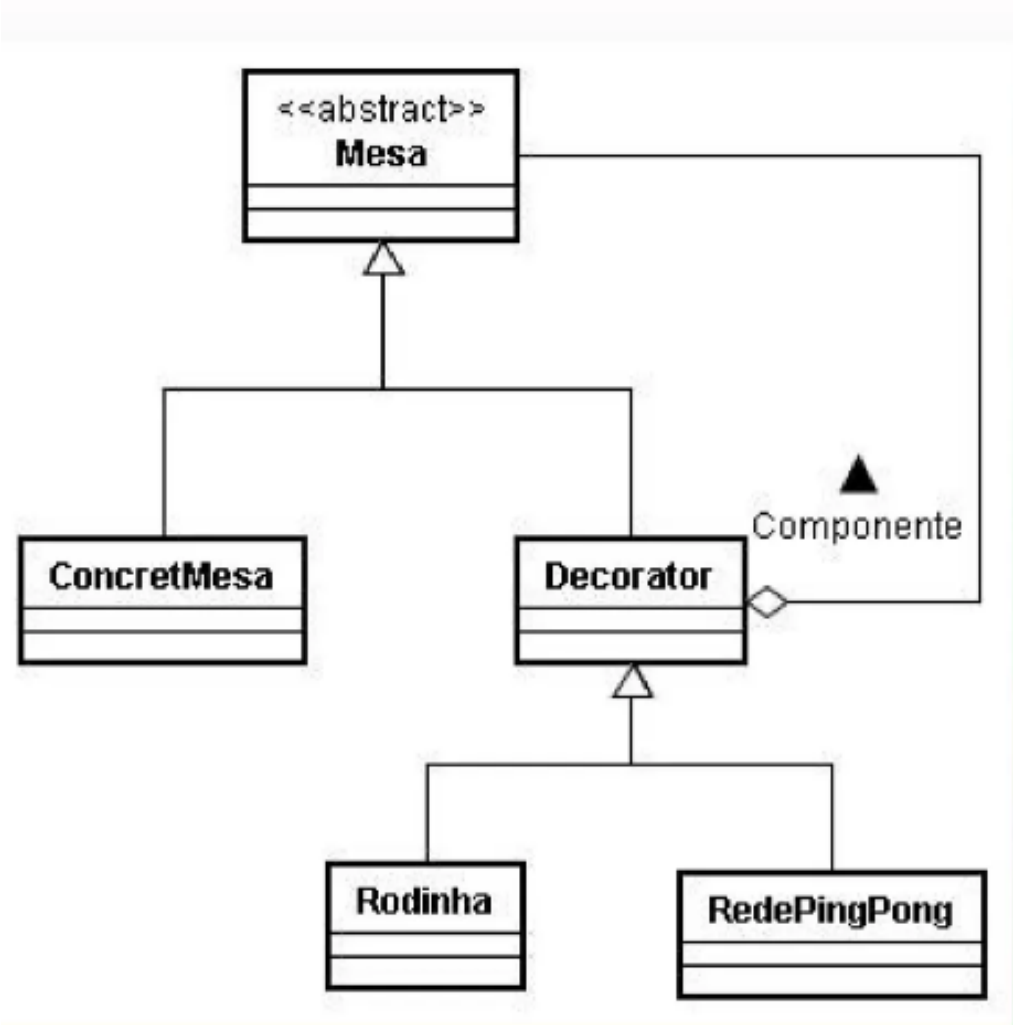
Problema: Como resolver?

- Usando herança?
  - Solução mais "natural".
  - Adição estática de responsabilidades
  - Cliente não tem como controlar como e quando colocar rodinhas, a rede de tênis de mesa ou outras coisas
  - Responsabilidade é atribuída a classe e não a cada objeto.
  - Não é facilmente extensível
  - E se quisermos usar a mesa para jantar?

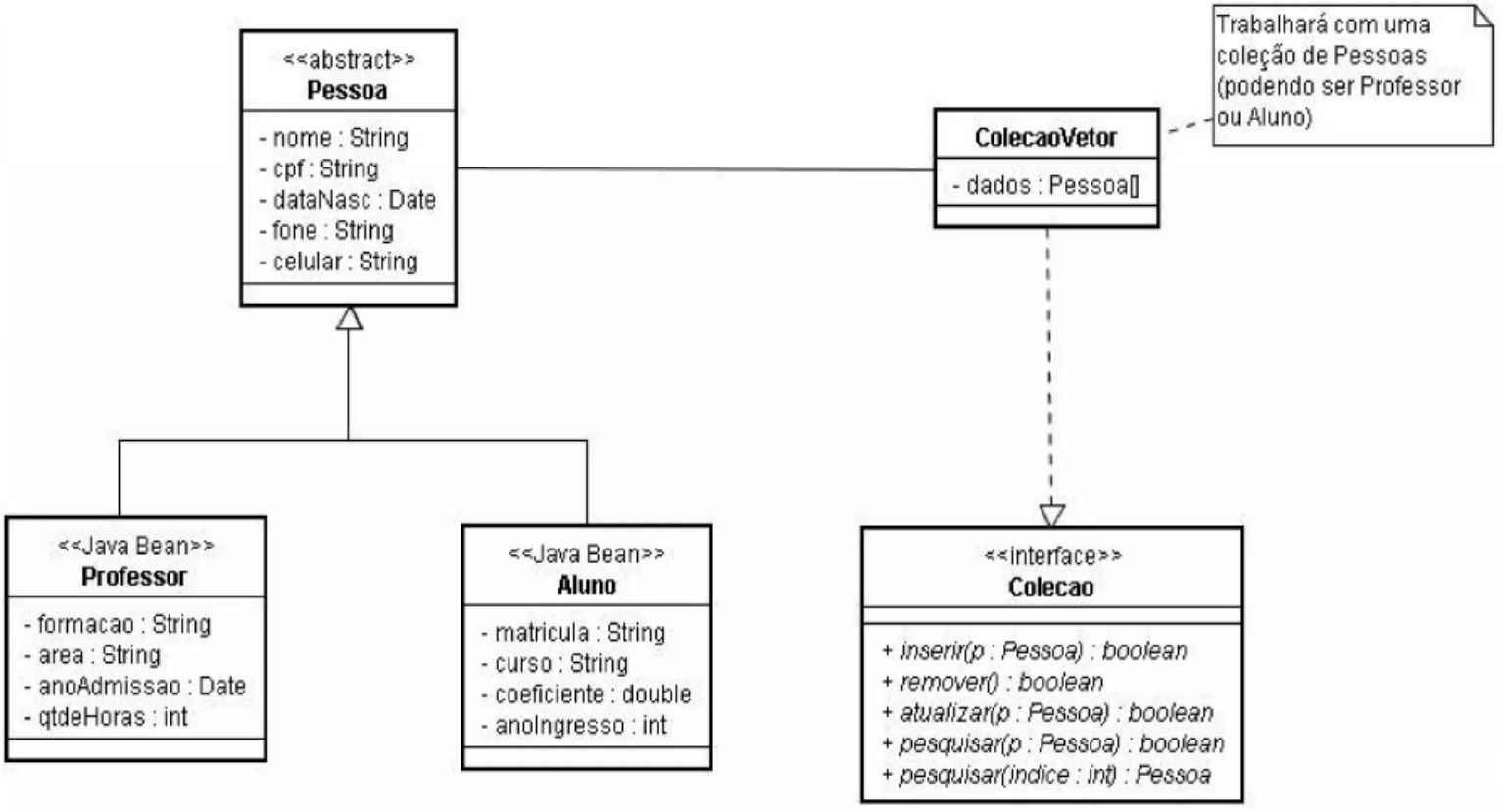
Solução Indicada

- Por que não decorar?
  - Embelezar a mesa com objeto que adiciona rodinhas, outro objeto, que adiciona rede de tênis de mesa, ...
  - cada objeto que "embeleza"a mesa é um *decorator* (Design Pattern)
  - Isto é composição de objetos

Diagrama: Mesas



Exercício


<<abstract>>
Pessoa
- nome : String
- cpf : String
- dataNasc : Date
- fone : String
- celular : String
ColecaoVetor
- dados : Pessoa[]
Trabalhará com uma coleção de Pessoas (podendo ser Professor ou Aluno)
<<Java Bean>>
Professor
- formacao : String
- area : String
- anoAdmissao : Date
- qtdeHoras : int
<<Java Bean>>
Aluno
- matricula : String
- curso : String
- coeficiente : double
- anoIngresso : int
<<interface>>
Colecao
+ inserir(p : Pessoa) : boolean
+ remover() : boolean
+ atualizar(p : Pessoa) : boolean
+ pesquisar(p : Pessoa) : boolean
+ pesquisar(indice : int) : Pessoa