Programação Orientada a Objetos
Composição, Herança e Polimorfismo

Rodrigo Bonifácio

23 de agosto de 2011

Diferentes alternativas de reuso em POO

- Composição
- Herança
- Templates

Diferentes alternativas de reuso em POO

- Composição
- Herança
- Templates (em C++)

Diferentes alternativas de reuso em POO

- Composição
- Herança
- Templates (em C++)
- Reuso de decisões de projeto, bibliotecas, frameworks

Composição

Reuso por composição ocorre quando uma classe C1 possui atributos membros de uma classe C2 qualquer

Composição

Reuso por composição ocorre quando uma classe C1 possui atributos membros de uma classe C2 qualquer (podendo ser a própria classe C1 quando definimos um tipo recursivo).

```
class Circulo : ... {
 private:
  Ponto centro;
  ...
 public:
  ...
}
```

Composição

Reuso por composição ocorre quando uma classe C1 possui atributos membros de uma classe C2 qualquer (podendo ser a própria classe C1 quando definimos um tipo recursivo).

```
class Circulo : ... {
 private:
  Ponto centro;
  ...
 public:
  ...
}
```

- Útil para representar uma relação *faz parte de*, ou seja, um ponto faz parte de um círculo.

Herança

Reuso por herança ocorre quando uma classe C1 (subclasse) herda propriedades e comportamento de uma classe C2 (superclasse).
Em geral, aplicável quando uma nova classe *é uma especialização de tipo* de uma classe existente.

Herança

Reuso por herança ocorre quando uma classe C1 (subclasse) herda propriedades e comportamento de uma classe C2 (superclasse).
Em geral, aplicável quando uma nova classe *é uma especialização de tipo* de uma classe existente.

```
const double pi = 3.1415;

class FiguraGeometrica {
 public:
  double area() { ...} ;
  void desenhe() { ...};
}

class Circulo : public FiguraGeometrica {
 private:
  Ponto centro;
  int raio;
 public:
  double area() { ... } ;
  void desenhe() { ... };
}
```

Navegar nos exemplso de ContaSimples e ContaEspecial

Redefinição de métodos

Subclasses podem redefinir os métodos das superclasses

Redefinição de métodos

Subclasses podem redefinir os métodos das superclasses mas é possível sobrescrever um método incluindo um comportamento adicional e chamar o método da superclasse em algum ponto

```
void ContaEspecial::sacar(double valor) {
  if(valor < saldo) {
    ContaSimples::sacar(valor);
  }
  else if(valor < (saldo + limite)) {
    double juros = (valor - saldo) * (1+taxa);
    saldo = saldo - (valor + juros);
  }
  else {
  ...
  }
}
```

Upcasting

```
class FiguraGeometrica { ... }
class Circulo : public FiguraGeometrica { ... }
...
void exibirImagem(FiguraGeometrica& f) {
  f.desenhe();
}

int main() {
  Circulo circulo(Pont(0,0), 10);
  exibirImagem(circulo);
}
```

Upcasting

```
class FiguraGeometrica { ... }
class Circulo : public FiguraGeometrica { ... }
...
void exibirImagem(FiguraGeometrica& f) {
  f.desenhe();
}

int main() {
  Circulo circulo(Pont(0,0), 10);
  exibirImagem(circulo);
}
```

- Em qualquer ponto da aplicação que se espera uma figura geométrica, um círculo pode ser usado.

Upcasting

```
class FiguraGeometrica { ... }
class Circulo : public FiguraGeometrica { ... }
...
void exibirImagem(FiguraGeometrica& f) {
  f.desenhe();
}

int main() {
  Circulo circulo(Pont(0,0), 10);
  exibirImagem(circulo);
}
```

- Em qualquer ponto da aplicação que se espera uma figura geométrica, um círculo pode ser usado. Um circulo *é uma* figura geométrica!

Multiple inheritance

You can inherit from one class, so it would seem to make sense to inherit from more than one class at a time. Indeed you can, but whether it makes sense as part of a design is a subject of continuing debate. One thing is generally agreed upon: You shouldn't try this until you've been programming quite a while and understand the language thoroughly. By that time, you'll probably realize that no matter how much you think you absolutely must use multiple inheritance, you can almost always get away with single inheritance.

Initially, multiple inheritance seems simple enough: You add more classes in the base-class list during inheritance, separated by commas. However, multiple inheritance introduces a number of possibilities for ambiguity, which is why a chapter in Volume 2 is devoted to the subject.

Thinking C++, Bruce Eckel

Métodos virtuais e polimorfismo

Métodos virtuais

Permite que uma classe C1 estabeleça um contrato indicando o que as subclasses de C1 possuem de diferente.

Conceitos relacionados

- Reúso de Tipos x Reúso de Implementação
- Ligação tardia
- Overridden, polimorfismo, ...

Hierarquia de instrumentos:

```
class Instrument {
 public:
  void playNote();
};

class Guitar : public Instrument {
 public:
  void playNote();
};
```

...E uma implementação correspondente

```
void Instrument::playNote() {
    cout << "[Instrument::play] zzzzzzzzzzz" << endl;
}

void Guitar::playNote() {
  cout << "[Guitar::play] ..., load up on guns, bring your ..." << endl;
}

void tune(Instrument& i) {
  i.playNote();
}

int main() {
  Guitar kurt;

  kurt.playNote();

  tune(kurt);
}
```

O que é exibido na tela?

O que é exibido na tela?

```
bash-3.2$ ./main
================================
Playing instruments:
================================
[Guitar::play] ..., load up on guns, bring your ...
[Instrument::play] zzzzzzzzzz
```

Limitações dessa implementação

- O método *playNote()* não é polimórfico
- Ou seja, a ligação das chamadas ao método *playNote()* com a implementação correspondente é feita em tempo de compilação (early binding).

Métodos polimórficos

A ligação de uma chamada de um método polimórfico com uma implementação específica ocorre em tempo de execução (late binding).

Métodos polimórficos

A ligação de uma chamada de um método polimórfico com uma implementação específica ocorre em tempo de execução (late binding).

- Em C++, apenas os métodos virtuais são polimórficos
- Em Java, por *default* os métodos são polimórficos

Como isso funciona?

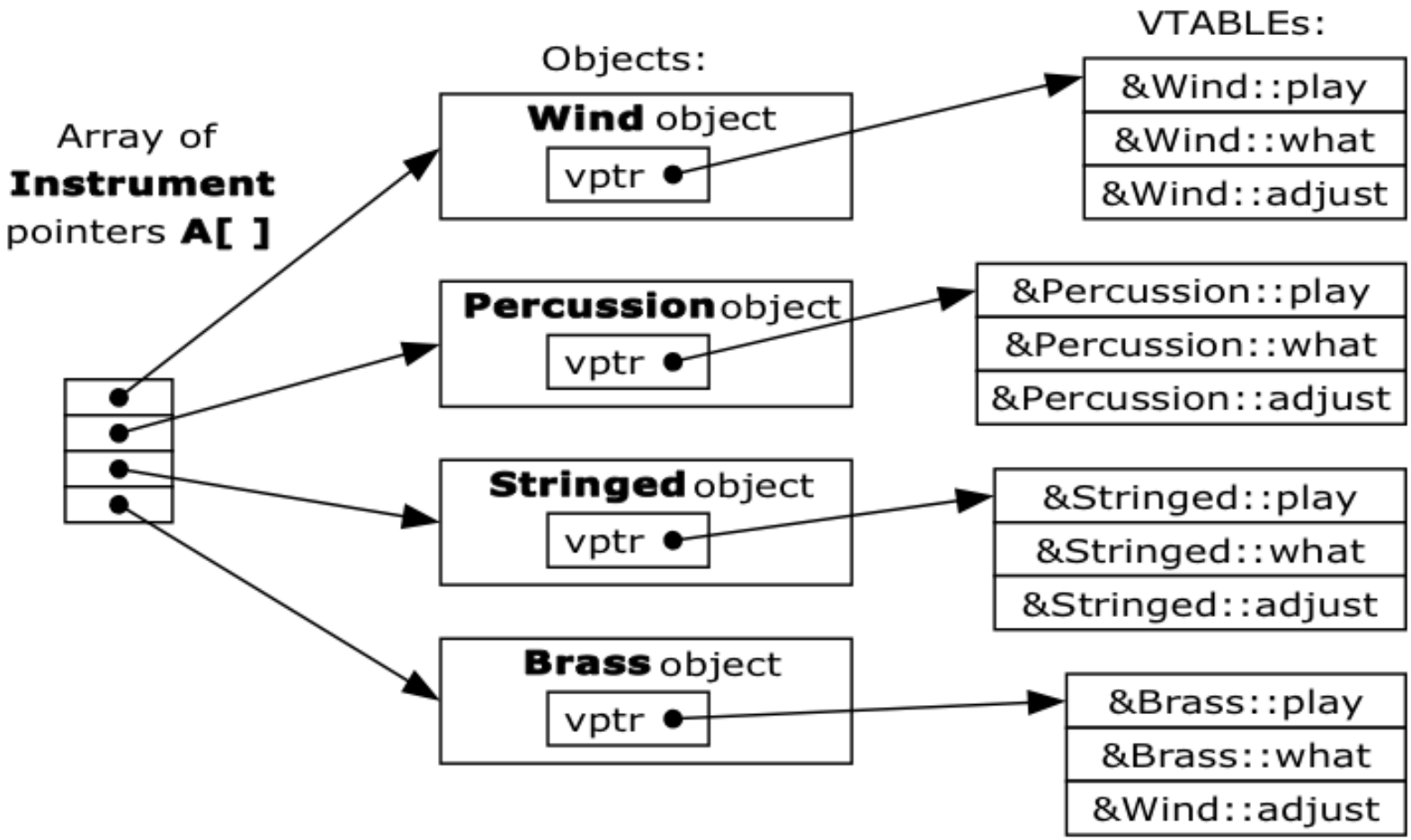
VTABLE

Para cada classe com pelo menos um método virtual, os compiladores C++

VTABLE

Para cada classe com pelo menos um método virtual, os compiladores C++

1. criam um array (chamado de *virtual table*, VTABLE) para endereçar o código dos métodos virtuais.
2. adicionam os endereços das implementações dos métodos virtuais de uma classe na VTABLE correspondente.
3. adiciona, em cada instância de uma classe com método virtual, uma referência para a VTABLE correspondente.



Thinking C++, Bruce Eckel

Mais um “Toy Example”

```
#include <iostream>
#include <string>

using namespace std;

class Pet {
 public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
 public:
  string speak() const { return "Bark!"; }
};
```

```
int main() {
 Dog ralph;
 Pet* p1 = &ralph;
 Pet& p2 = ralph;
 Pet p3;
 cout << "p1.speak() = " << p1->speak() << endl;
 cout << "p2.speak() = " << p2.speak() << endl;
 cout << "p3.speak() = " << p3.speak() << endl;
}
```

Se é tão interessante, por que todos os métodos não são virtuais C++?

Se é tão interessante, por que todos os métodos não são virtuais C++?

- Basicamente, porque C++ foi projetada tendo eficiência como uma das principais preocupações.

Métodos virtuais puros

Frequentemente, métodos na classe base não têm uma implementação real

Métodos virtuais puros

Frequentemente, métodos na classe base não têm uma implementação real (qual a representação de uma Figura Geométrica?)

Métodos virtuais puros

Frequentemente, métodos na classe base não têm uma implementação real (qual a representação de uma Figura Geométrica?) servindo apenas como uma interface para as suas subclasses.

Métodos virtuais puros

Frequentemente, métodos na classe base não têm uma implementação real (qual a representação de uma Figura Geométrica?) servindo apenas como uma interface para as suas subclasses.

- Não fazendo sentido, inclusive, instanciar a classe base
- Essas classes são ditas "abstratas", e possuem pelo menos um método virtual puro.

Métodos virtuais puros

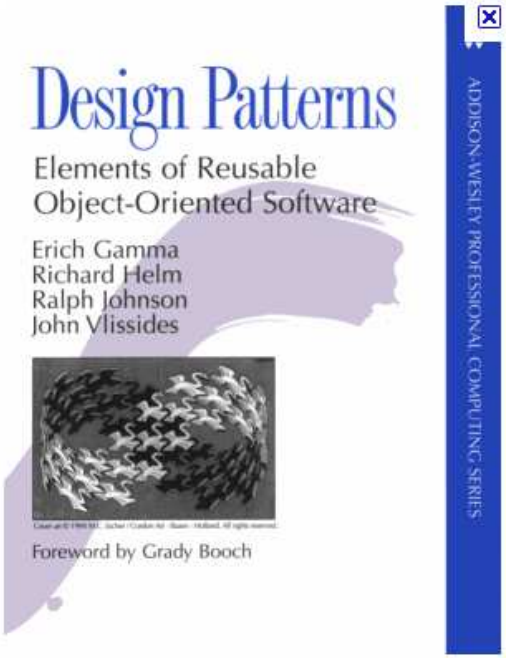
Frequentemente, métodos na classe base não têm uma implementação real (qual a representação de uma Figura Geométrica?) servindo apenas como uma interface para as suas subclasses.

- Não fazendo sentido, inclusive, instanciar a classe base
- Essas classes são ditas "abstratas", e possuem pelo menos um método virtual puro.

```
class FiguraGeometrica {
 public:
  virtual double area() = 0;
  virtual void desenhe() = 0;
}
```

Métodos virtuais puros são formas de contrato, permitindo um design em termos de interfaces, não em termos de uma implementação específica!

Métodos virtuais puros são formas de contrato, permitindo um design em termos de interfaces, não em termos de uma implementação específica!



Em linhas gerais . . .

Em linhas gerais . . .

- Polimorfismo ocorre quando, em tempo de compilação, não sabemos qual o comportamento da chamada de um método.

Em linhas gerais . . .

- Polimorfismo ocorre quando, em tempo de compilação, não sabemos qual o comportamento da chamada de um método.
- Conceito chave, suportado por ligação tardia (late binding), e que, caso não aplicado no design de software, perdemos oportunidades de tornar o software mais flexível.

Em linhas gerais . . .

- Polimorfismo ocorre quando, em tempo de compilação, não sabemos qual o comportamento da chamada de um método.
- Conceito chave, suportado por ligação tardia (late binding), e que, caso não aplicado no design de software, perdemos oportunidades de tornar o software mais flexível.
- A prática programar para uma interface, não para uma implementação direciona a construção de soluções polimórficas.

Em linhas gerais . . .

- Polimorfismo ocorre quando, em tempo de compilação, não sabemos qual o comportamento da chamada de um método.
- Conceito chave, suportado por ligação tardia (late binding), e que, caso não aplicado no design de software, perdemos oportunidades de tornar o software mais flexível.
- A prática programar para uma interface, não para uma implementação direciona a construção de soluções polimórficas. Em C++, isso significa que possíveis pontos de extensão do software devem ser abstraídas em termos de métodos virtuais (possivelmente puros)

Polimorfismo, de acordo com Cardelli e Wegner

Polimorfismo, de acordo com Cardelli e Wegner

Universal

- **Polimorfismo por subtipos** (coberto aqui)
- Polimorfismo parametrizado (C++ templates)

Ad hoc

- Sobrecarga de funções e operadores
- Conversões de tipos implícitas ou explícitas