Suporte a programação orientada a objetos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Conteúdo

Introdução

Introdução

- Muitas linguagens suportam programação orientada a objeto
  - De Cobol a Lisp
  - C++ e Ada (procedural e orientada a objeto)
  - CLOS (linguagem funcional)
  - Java e C# (orientada a objetos mas usa construções imperativas)
  - Smalltalk (orientada a objeto pura)

Programação orientada a objeto

Programação orientada a objeto

- Uma linguagem que suporta programação OO deve ter três características
  - Tipo abstrato de dados (encapsulamento)
  - Herança
  - Vinculação dinâmica de chamada de métodos (polimorfismo)

Herança

- Limitações dos TAD's
  - Reuso: alguma mudança sempre é necessária
  - Organização: todos TAD's estão no mesmo nível

Herança

- A herança permite criar novos tipos baseados em tipos existentes
  - Reuso
    - Um novo tipo herda os dados e funcionalidades de um tipo já existente
    - É possível adicionar novos dados e funcionalidades
    - É possível alterar as funcionalidades existentes
  - Organização
    - É possível criar hierarquia de classes

Herança

- Conceitos
  - TADs em linguagens OO são chamados de **classes**
  - Uma instância de uma classe é chamada de **objeto**
  - Uma classe que é definida através de herança é chamada de **classe derivada** ou **subclasse**
  - Uma classe da qual uma nova classe é derivada é chamada de **classe pai** ou **super classe**
  - Os subprogramas que definem operações sobre os objetos de uma classe são chamados de **métodos**
  - A chamada de métodos também é conhecida como **mensagem**

Herança

- Uma classe derivada pode diferenciar-se da classe pai de várias maneiras
  - Alguns membros da classe pai podem ser privados, o que implica que eles não são visíveis nas subclasses
  - A subclasse pode adicionar novos membros
  - A subclasse pode modificar o comportamento dos métodos herdados
    - O novo método **sobrescreve** o método herdado
    - O método do pai é **sobrescrito**

Herança

- Classes podem ter dois tipos de variáveis
  - De instância e de classe
- Classes podem ter dois tipos de métodos
  - De instância e de classe

Herança

- Herança simples
  - Subclasse de uma única classe pai
- Herança múltipla
  - Subclasse de múltiplas classes pai

Herança

- Desvantagem da herança
  - Cria dependências entre as classes da hierarquia

Vinculação dinâmica

- Uma **variável polimórfica** de uma classe é capaz de referenciar (ou apontar) para objetos da classe e objetos de qualquer subclasse
- Quando uma classe em uma hierarquia de classes sobrescreve um método, e este método é chamado através de uma variável polimórfica, a vinculação para o método correto será dinâmica
- Permite que softwares sejam mais facilmente modificados

Vinculação dinâmica

- Conceitos
  - Um **método abstrato** não contém a definição, ele define apenas o protocolo
  - Uma **classe abstrata** contém pelo menos um método abstrato
  - Uma classe abstrata não pode ser instanciada

Questões de projeto

Questões de projeto

- Exclusividade de objetos
- As subclasses são subtipos?
- Herança simples e múltipla
- Alocação e desalocação de objetos
- Vinculação dinâmica e estática
- Classes aninhadas
- Inicialização de objetos

Exclusividade de objetos

- Tudo é objeto
  - Vantagem: elegância e uniformidade
  - Desvantagem: baixo desempenho para operações simples
  - Exemplo: Smalltalk

Exclusividade de objetos

- Sistema tradicional de tipo + modelo de objetos
  - Vantagem: operações rápidas em tipos simples
  - Desvantagem: sistema de tipos confuso
  - Exemplo: C++

Exclusividade de objetos

- Tipos primitivos escalares tradicionais e os outros tipos como objetos
  - Vantagem: operações rápidas com os tipos primitivos
  - Desvantagem: sistema de tipos mais complicado
  - Exemplo: Java

As subclasses são subtipos?

- Uma subclasse é um **subtipo** se ela tem a relação "é uma" com sua classe pai, isto é, objetos da subclasse podem aparecer em qualquer lugar onde a classe pai era válida, sem causar erro de tipo
  - Necessário que todos os membros que eram expostos na classe pai sejam expostos pela classe filho
- Definição mais forte
  - Objetos da subclasse devem se comportar de maneira equivalente aos objetos da classe pai (princípio da substituição de Liskov)

Herança simples e múltipla

- Herança múltipla permite que uma subclasse derive de mais de uma classe
  - Vantagem
    - As vezes a herança múltipla é útil (quando?)
  - Desvantagem
    - Complexidade na implementação (colisão de nomes)
    - Ineficiência, herança múltipla custa mais que herança simples
    - O projeto das classes é mais difícil (aumento na manutenção)

Alocação e desalocação de objetos

- De onde os objetos são alocados?
  - Do heap (uniformidade de acesso, atribuição simples, desreferenciamento implícito)
  - Da pilha (problemas com atribuição de subtipo)
- No caso de objetos alocados do heap, a desalocação é implícita, explícita ou das duas maneiras?
  - Implícita implica que algum método de recuperação de memória é requerido
  - Explícita implica que ponteiros pendentes podem ser criados

Vinculação dinâmica e estática

- Todas as vinculações de mensagens para métodos devem ser dinâmicas?
- Vinculações dinâmicas são lentas, mas necessárias para o polimorfismo de tipo
- Um opção é permitir que o usuário especifique o tipo de vinculação

Classes aninhadas

- Classes aninhadas são interessantes pois aumentam as possibilidades de ocultação de informação
- Classes podem ser aninhadas?
- Quais dos membros da classe aninhada são visíveis pela classe que a encapsula?
- Quais dos membros da classe encapsuladora são visíveis pela classe aninhada?

Inicialização de objetos

- Os objetos são inicializados manualmente ou através de uma mecanismo implícito?
- Como os membros da classe pai são inicializados quando um objeto de uma subclasse é criado?

Suporte a programação orientada a objeto

Suporte a programação orientada a objeto em Smalltalk

- Características gerais
  - Tudo é objeto
  - A computação ocorre através de troca de mensagens
  - Mensagens podem ser parametrizadas
  - Todos os objetos são alocados do heap
  - A desalocação é implícita
  - Os construtores precisam ser chamados explicitamente
  - Não tem a aparência de linguagens imperativas

Suporte a programação orientada a objeto em Smalltalk

- Checagem de tipo e polimorfismo
  - A vinculação das mensagens para métodos é dinâmica
  - Quando uma mensagem é enviada para um objeto, o método corresponde é buscado na classe do objeto, se o método não for encontrado a busca continua na classe pai e assim sucessivamente. O processo continua até a classe `Object`
  - A checagem de tipo é dinâmica e o único erro acontece quando uma mensagem é enviada para um objeto que não tem um método correspondente

Suporte a programação orientada a objeto em Smalltalk

- Herança
  - Uma subclasse herda todas as variáveis de instância, métodos de instância e métodos de classe da classe pai
  - Todas as subclasses são subtipos
  - Uma subclasse pode acessar um método sobrescrito usando a pseudo variável `super`
  - Herança simples

Suporte a programação orientada a objeto em Smalltalk

- Avaliação
  - É uma linguagem pequena, a sintaxe é simples e regular
  - Bom exemplo do poder fornecido por uma linguagem simples
  - Lento quando comparado com linguagens imperativas compiladas
  - Vinculação dinâmica pode adiar a detecção de erros até a execução do programa
  - Introduziu interfaces gráficas e IDE's
  - Avanços na POO

Suporte a programação orientada a objeto em C++

- Característica gerais
  - Baseado em C e em SIMULA 67
  - Primeira linguagem de programação OO amplamente utilizada
  - Dois sistemas de tipo: imperativo e OO
  - Objetos podem ser estáticos, dinâmicos na pilha ou dinâmicos no heap
  - Desalocação explícita usando o operador `delete`
  - Destrutores
  - Mecanismo de controle de acesso elaborado

Suporte a programação orientada a objeto em C++

- Herança
  - Tipos de controle de acesso aos membros
    - `private`, `protected` e `public`
  - Um classe não precisa ter uma classe pai

Suporte a programação orientada a objeto em C++

- Herança
  - Todos os objetos precisam ser inicializado antes de serem usados, no caso de subclasse, os membros herdados precisam ser inicializados quando uma instância da subclasse é criada
  - Uma subclasse não precisa ser um subtipo
    - Derivação pública: os membros públicos e protegidos são membros públicos e protegidos na subclasse
    - Derivação privada: todos os membros públicos e protegidos são membros privados na subclasse

Suporte a programação orientada a objeto em C++

- Herança
  - Suporta herança múltipla
    - Se dois membros herdados tem o mesmo nome, eles podem ser acessados usando o operador de resolução de escopo (::)
  - Um método da subclasse precisa ter os mesmos parâmetros do método da classe pai para sobrescrevê-lo. O tipo de retorno tem que ser o mesmo ou um tipo derivado (público)

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0;};
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();
    bool empty();
};
```

```
// Como stack é uma derivação pública, todos os
// métodos públicos da classe single_linked_list
// também são públicos em stack, o que deixa a
// classe com métodos públicos indesejados
// (insert_at_head, insert_at_tail e remove_at_head)

class stack: public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
};
```

```
// stack2 é uma derivação privada de
// single_linked_list, portanto os membros
// públicos e protegidos herdados de
// single_linked_list são privados em stack2.
// stack2 tem que redefinir a visibilidade do
// membro empty para torná-lo público.

class stack2: private single_linked_list {
  public:
    stack2() {}
    void push(int value) {
      insert_at_head(value);
    }
    int pop() {
      return remove_at_head();
    }
    single_linked_list::empty;
};
```

```
// Uma alternativa mais interessante é usar composição,
// o que permite que uma pilha possa ser
// definida com qualquer implementação de lista.
class stack3 {
  private:
    list *li;
  public:
    stack3(list *l) : li(l) {}
    void push(int value) {
      li->insert_at_head(value);
    }
    int pop() {
      return li->remove_at_head();
    }
    boolean empty() {
      return li->empty();
    }
};
```

Suporte a programação orientada a objeto em C++

- Vinculação dinâmica
  - Um método pode ser definido como virtual, que significa que ele será vinculado dinamicamente quando chamado em uma variável polimórfica
  - Funções virtuais puras não têm definição
  - Uma classe que tem pelo menos um método virtual puro é uma classe abstrata

```
class Shape {
  public:
    virtual void name() = 0;
};
class Rectangle: public Shape {
  public:
    void name() {
        printf("Rectangle\n");
    }
    void code() {
        printf("R\n");
    }
};
class Square: public Rectangle {
  public:
    void name() {
        printf("Square\n");
    }
    void code() {
        printf("S\n");
    }
};
```

O que será impresso?

```
Shape* s = new Rectangle();
s->name();
...
s = new Square();
s->name();
```

```
Rectangle
Square
```

name é um método virtual e portanto é vinculado dinamicamente ao método da classe **instanciada** referenciada por r.

```
class Shape {
  public:
    virtual void name() = 0;
};
class Rectangle: public Shape {
  public:
    void name() {
        printf("Rectangle\n");
    }
    void code() {
        printf("R\n");
    }
};
class Square: public Rectangle {
  public:
    void name() {
        printf("Square\n");
    }
    void code() {
        printf("S\n");
    }
};
```

O que será impresso?

```
Rectangle *r = new Rectangle();
r->code();
...
r = new Square();
r->code();
```

```
R
R
```

code **não** é um método virtual e portanto é vinculado estaticamente ao método do tipo declarado de `r`.

Suporte a programação orientada a objeto em C++

- Avaliação
  - C++ fornece muitas formas de controle de acesso (diferente de Smalltalk)
  - C++ fornece herança múltipla
  - Em C++, é necessário definir em tempo de projeto quais métodos serão vinculados estaticamente e quais serão vinculados dinamicamente
  - A checagem de tipo em Smalltalk é dinâmico, o que é flexível mas inseguro
  - Smalltalk é 10 vezes lento que C++

Suporte a programação orientada a objeto em Java

- Características gerais
  - Similar a C++
  - Todos os tipos são objetos, exceto os primitivos (booleano, caractere, numéricos)
  - Para os tipos primitivos poderem ser usados com tipos objetos, eles devem ser colocados em objetos que são invólucros (wrappers)
  - Java 5 adicionou autoboxing e autounboxing (criação e remoção do invólucro automaticamente)

Suporte a programação orientada a objeto em Java

- Características gerais
  - Todos as classes são descendentes de `Object`
  - Todos os objetos são dinâmicos no heap, e a desalocação é implícita
  - O método `finalize` é executado quando o objeto é desalocado pelo coletor de lixo
  - Como o momento exato que finalize será executado não pode ser determinado, é necessário outro mecanismo definido pelo usuário para desalocação de recursos

Suporte a programação orientada a objeto em Java

- Herança
  - Um método pode ser declarado `final`, o que significa que ele não pode ser sobrescrito
  - Uma classe pode ser declarada `final`, o que significa que ela não pode ser a classe pai de nenhuma outra classe
  - O construtor da classe pai deve ser chamado antes do construtor da subclasse

Suporte a programação orientada a objeto em Java

- Herança
  - Toda subclasse é subtipo
  - Suporta apenas herança simples
  - Uma interface é semelhante a uma classe abstrata, mas pode ter apenas as declarações dos métodos e constantes
  - Uma classe pode implementar mais de uma interface (mix-in)

Suporte a programação orientada a objeto em Java

- Vinculação dinâmica
  - Todas as mensagens são vinculadas dinamicamente a métodos, exceto se o método for `final`, `private` ou `static`

Suporte a programação orientada a objeto em Java

- Classes aninhadas
  - Várias formas de classes aninhadas
  - A classe que encapsula a classe aninhada pode acessar qualquer membro da classe aninhada
  - Uma classe aninhada não estática, tem uma referência para uma instância da classe que a encapsula e portanto pode acessar os membros desta instância
  - A classe aninhada pode acessar qualquer membro da classe que a encapsula
  - Classes aninhadas podem ser anônimas
  - Uma classe aninhada pode ser declarada em um método

Suporte a programação orientada a objeto em Java

- Avaliação
  - Suporte a POO semelhante ao do C++
  - Não suporta programação procedural
  - Todas as classes tem pai
  - Toda subclasse é subtipo
  - Vinculação dinâmica por padrão
  - Interfaces fornecem uma forma simples de herança múltipla

Suporte a programação orientada a objeto em C#

- Características gerais
  - Incluí classes e estruturas
  - As classes são semelhantes as classes em Java
  - As estruturas são alocadas na pilha e não oferece herança

Suporte a programação orientada a objeto em C#

- Herança
  - Mesma sintaxe utilizada em C++
  - Um método herdado da classe pai pode ser substituído por uma na classe derivada marcando o método com new
  - O método substituído da classe pai pode ser acesso com o prefixo base
  - O suporte a interface é o mesmo que o do Java

Suporte a programação orientada a objeto em C#

- Vinculação dinâmica
  - A classe pai precisa marcar o método com `virtual`
  - A subclasse precisa marcar o método com `override`
  - Um método pode ser marcado como `abstract`
  - Uma classe que contém pelo menos um método abstrato precisa ser marcada como `abstract`
  - Todos as classes são descendentes de `Object`

Suporte a programação orientada a objeto em C#

- Classes aninhadas
  - Uma classe aninhada é como uma classe aninhada estática em Java

Suporte a programação orientada a objeto em C#

- Avaliação
  - Versão recente de uma linguagem de programação baseada em C, manteve o que funcionava e "corrigiu" o que era problemático
  - As diferenças entre C# e Java são pequenas

Suporte a programação orientada a objeto em Ada 95

- Características gerais
  - Suporte a POO foi a principal extensão a Ada 83
  - Tipo etiquetado (tagged type) criados em pacotes
  - Um tipo etiquetado é um tipo cujo os objetos tem uma etiqueta que indica durante a execução qual é o seu tipo
  - Tipos etiquetados podem ser tipos privados ou registros
  - Nem os construtores nem os destrutores são implicitamente chamados

Suporte a programação orientada a objeto em Ada 95

- Herança
  - Subclasses são derivadas de tipos etiquetados
  - Novas entidades são adicionadas as entidades herdadas colocando-as em um definição de registro
  - Todas as subclasses são subtipos
  - Não oferece suporte a herança múltipla (um efeito semelhante pode ser obtido usando classes genéticas)

```
Package Person_Pkg is
  type Person is tagged private;
  procedure Display(P : in out Person);
  private
    type Person is tagged
      record
        Name : String(1..30);
        Address : String(1..30);
        Age : Integer;
      end record;
end Person_Pkg;
with Person_Pkg; use Person_Pkg;
 package Student_Pkg is
   type Student is new Person with
     record
       Grade_Point_Average : Float;
       Grade_Level : Integer;
     end record;
   procedure Display (St: in Student);
 end Student_Pkg;
-- Note: Display is being overridden from Person_Pkg
```

Suporte a programação orientada a objeto em Ada 95

- Vinculação dinâmica
  - A vinculação dinâmica é feita com variáveis polimórficas chamadas de tipo classwide
  - As outras vinculações são estáticas
  - Qualquer método pode ser vinculado dinamicamente
  - Classes abstratas podem ser definidas usando a palavra reservada `abstract`

```
with Person_Pkg; use Person_Pkg;
with Student_Pkg; use Student_Pkg;
P : Person;
S : Student;
Pcw : Person'class
...
Pcw := P;
Display(Pcw); -- class the Display in Person
Pcw := S;
Display(Pcw); -- class the Display in Student
```

Suporte a programação orientada a objeto em Ada 95

- Avaliação
  - Oferece suporte completo a POO
  - C++ oferece uma forma de suporte a herança melhor que ADA
  - Ada não incluí construtores
  - Vinculação dinâmica não é restrita a ponteiros e/ou referências (mais ortogonal)

Suporte a programação orientada a objeto em Ruby

- Características gerais
  - Tudo é objeto
  - Declarações de classe são executáveis, permitindo uma segunda definição adicionar membros as definições existentes
  - Definições de método também são executáveis
  - Todos as variáveis são referências sem tipo a objetos
  - Controle de acesso aos membros
    - Todos os dados são privados
    - Getters e setter podem ser definidos por atalhos
    - Os métodos podem ser públicos, privados ou protegidos

Suporte a programação orientada a objeto em Ruby

- Herança
  - O controle de acesso dos métodos herdados podem ser diferente do que aquele da classe pai
  - Subclasses não são necessariamente subtipos
  - Mixis podem ser criados com módulos, fornecendo um tipo de herança múltipla

Suporte a programação orientada a objeto em Ruby

- Vinculação dinâmica
  - Todas as variáveis são sem tipos e polimórficas

Suporte a programação orientada a objeto em Ruby

- Avaliação
  - Não suporta classes abstratas
  - Não suporta completamente herança múltipla
  - Controle de acesso é fraco em ralação a outras linguagens

Implementação de construções orientada a objeto

Implementação de construções orientada a objeto

- Duas questões importantes
  - Estrutura de armazenamento para variáveis de instâncias
  - Vinculação dinâmica de mensagens para métodos

Implementação de construções orientada a objeto

- Registro de instância de classe (RIC) armazena o estado do objeto
- O RIC é estático (construído em tempo de compilação)
- Se uma classe tem pai, as variáveis de instância da subclasse são adicionadas ao RIC da classe pai
- O acesso as variáveis é feito como em registros (usando um deslocamento), acesso eficiente

Implementação de construções orientada a objeto

- Os métodos que são vinculados estaticamente não precisam estar envolvidos com o RIC
- A chamada a métodos vinculados dinamicamente pode ser implementada como um ponteiro no RIC
- Todos os ponteiros para métodos podem ser armazenados em uma tabela de métodos virtuais (vtable), e esta tabela compartilhada por todas as instâncias
- Um método é representado como um deslocamento do início da tabela

Implementação de construções orientada a objeto

```
public class A {
    public int a, b;
    public void draw() { . . . }
    public void area() { . . . }
}

public class B extends A {
    public int c, d;
    public void draw() { . . . }
    public void sift() { . . . }
}
```

Implementação de construções orientada a objeto

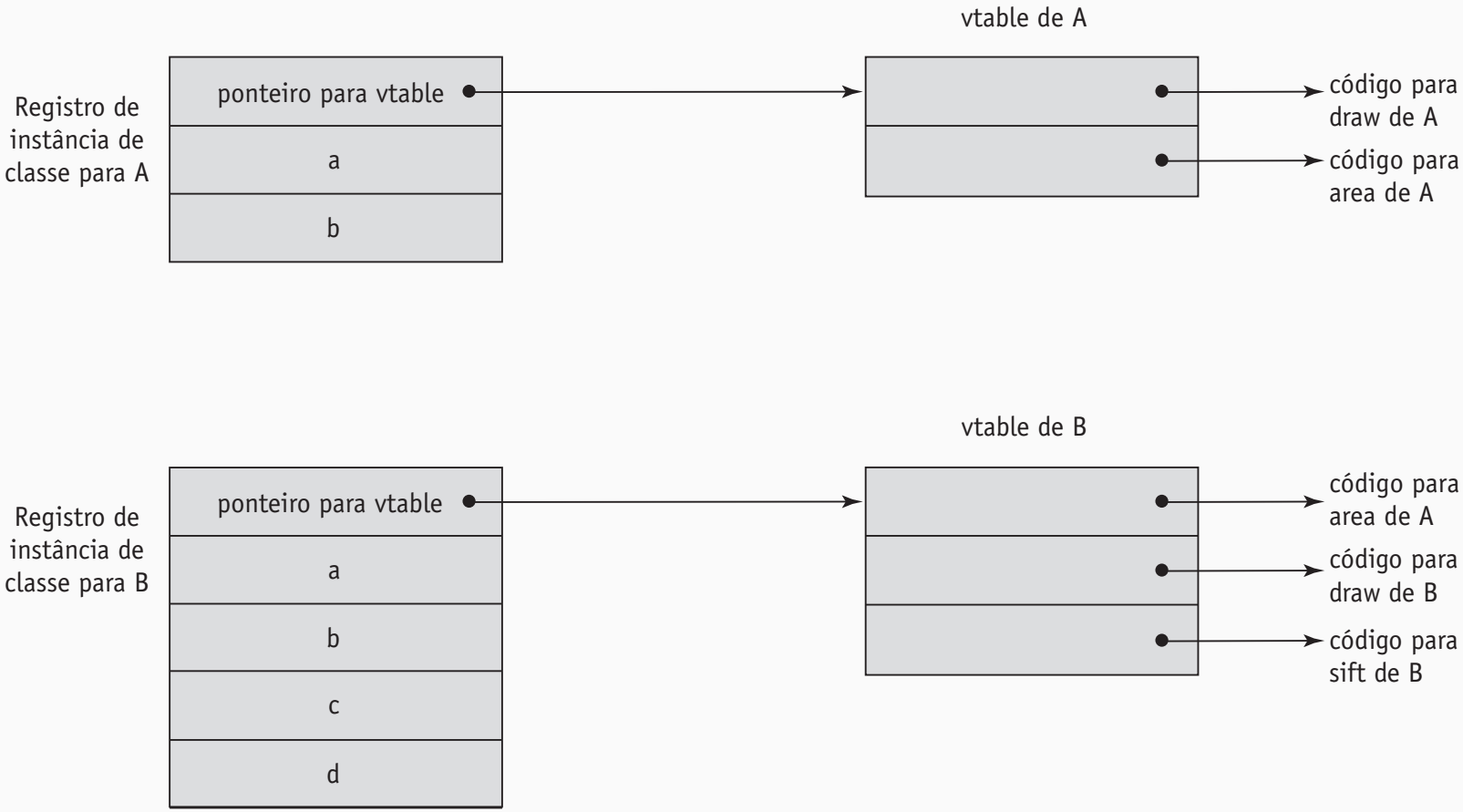

Figura 12.2 Um exemplo das RICs com herança simples.

Implementação de construções orientada a objeto

```
class A {
    public: int a;
    virtual void fun() { . . . }
    virtual void init() { . . . }
}

class B {
    public: int b;
    virtual void sun() { . . . }
}

class C: public A, public B {
    public: int c;
    void fun() { . . . }
    virtual void dud() { . . . }
}
```

Implementação de construções orientada a objeto

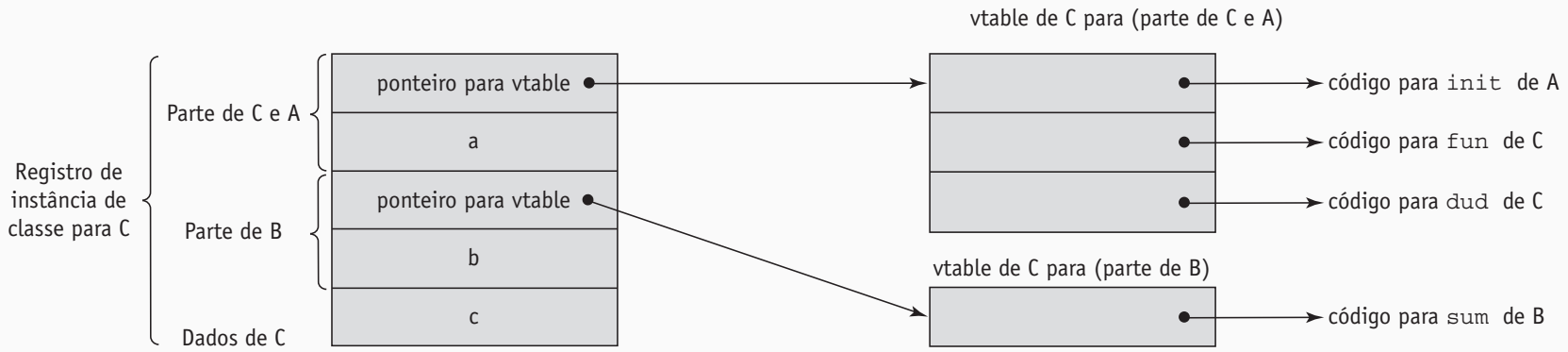

Figura 12.3 Um exemplo de um RIC de uma subclasse com múltiplos pais.

Exemplo em C++

- Arquivo `vtable.cpp`

Referências

- Robert Sebesta, Concepts of programming languages, 9ª edição. Capítulo 12.