PROGRAMAÇÃO ORIENTADA POR OBJETOS

ITA

Marco Túlio de Oliveira Valente
Roberto da Silva Bigonha

UFMG

Dezembro de 1995

CONTENTS

PROGRAMAÇÃO OO EM ITA

MOTIVAÇÃO

- Linguagens Orientadas por Objetos: Smalltalk, Objective C, C++, Oberon-2, Eiffel, Sather, Ada 95 etc.
- LOO de maior sucesso: C++
- Críticas a C++:
  - Número excessivo de recursos;
  - Difícil domínio;
  - Baixa curva de aprendizado;
  - Falta de segurança.
- Orientação por objetos não implica nesses fatores. Exemplos: Oberon-2 e Eiffel.
- O que se deseja: Qualidade de Software
  - Modularidade (reuso);
  - Programação sistemática;
  - Simplicidade;
  - Poder de expressão.
- Objetivo do trabalho:
  - Projetar e implementar uma nova LOO: Ita;
  - Ambiente de experimentação em tecnologia orientada por objetos.

ITA E OUTRAS LINGUAGENS

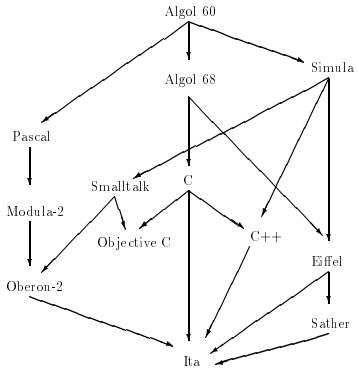

Figura 1: Ita e outras LOO

CARACTERÍSTICAS DE Ita

- Baseada em C:
  - Tipos
  - Comandos
  - Estrutura de Programa
- Programação Orientada por Objetos:
  - Classes (inclusive genéricas e abstratas)
  - Encapsulamento
  - Herança (Simples)
  - Polimorfismo
  - Polissemia (inclusive por estado)
  - Polivalência
- Programação por Contrato:
  - Asserções (invariantes, pré e pós-condições)
  - Tratamento de Exceções (semântica de retomada)

CLASSES

- Construções usadas para suportar a implementação de TAD.

```
class A {          // Membros privados
   int a1;                 // atributo
   int f1 (...) { ..... } // metodo
  .....
  public:  .....   // Membros publicos
   init (...) { ... } // inicializadora
   finish () { ... }  // finalizadora
   int a2;            // atributo
   int f2 (...);      // metodo
} ......
int A::f2 (...) { ..... }
......
A a1= new A (10, 3.56);
a1.f2 (.....);
delete a1; ......
```

- Encapsulamento:
  - Membros públicos ou privados (inclusive para subclasses);
  - Privilegiar programação em larga escala.

SEMÂNTICA DE REFERÊNCIA

- Supondo p1 e p2 dos tipos P1 e P2:
  `p1= p2;  // atribuicao de referencias`

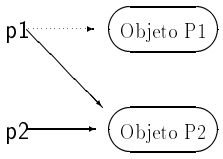

Figura 2: Semântica de Referência

- Sem coleta de lixo.
- Operações com semântica de valor:
  - Atribuição: `copy (p1, p2);`
  - Duplicação: `clone (p1, p2);`
  - Comparação: `equal (p1, p2);`

EXPORTAÇÃO SELETIVA

- Classes Friends: recurso para flexibilizar o encapsulamento de classes.

```
class A friend B {
  int x;  .....
}

class B { .....
  int g (void) {
    A a= new A;
    int y= a.x;  // acesso a atributo
    .....        // privado de A
  }  .....
}
```

- Membros privados de `A` são acessíveis em `B`.

CLASSES GENÉRICAS

- Generalidade: capacidade de parametrizar classes.
- Parâmetros Genéricos:
  - Irrestritos: instanciados com qualquer tipo;
  - Restritos: instanciados apenas com subtipos do tipo restringente.
-
```
class A <T1, T2:B> {   // T1: Irrestrito
       T1 x;           // T2: Restrito
     public:
       T1 f (T1 t1, T2 t2) { ..... }
}
......
A <int, B> a1= new A <int, B>;
A <X, C> a2= new A <X, C>;
B b= new B;
......
int z= a1.f (10, b);
```

- Implementadas com semântica de código único.
- Vantagens:
  - Facilidade de uso e distribuição;
  - Segurança (verificação de tipos).
- Desvantagens: Tipos básicos tratados como objetos.

HERANÇA

- Mecanismo para criar subtipos.

```
class A {
    int a1;
  public:
    int f (int x)   { ..... }
}

class B: A {   // subclasse de A
    int b1;
  public:
    int f (int x)   { ..... }
    int g (float y) { ..... }
}
```

- Por que não suportar herança múltipla ?
  - Custo de implementação;
  - Problemas de difícil solução:
    - Colisão de nomes;
    - Herança repetida.
  - Na maior parte dos casos, alternativas de modelagem via herança simples e composição.

POLIMORFISMO

- **Segundo o Aurélio:** 1. Que se apresenta sob numerosas formas; multiforme. 2. Sujeito a variar de forma.

- **Segundo Cardelli & Wegner:** Designa a capacidade de variáveis de uma linguagem possuírem mais de um tipo.

$$\text{Polimorfismo} \begin{cases} \text{Universal} \begin{cases} \text{Paramétrico} \\ \text{Inclusão} \end{cases} \\ \text{Ad-hoc} \begin{cases} \text{Sobrecarga} \\ \text{Coerção} \end{cases} \end{cases}$$

- **Definição de** Ita: Propriedade de referências denotarem em tempo de execução objetos de uma classe ou de suas subclasses.
- Referências é que são polimórficas !
- Supondo `A` uma classe e `B` uma subclasse:

```
A a;
B b= new B;
a= b;
a.f (10);      // chama B::f
```

POLISSEMIA

- **Segundo o Aurélio:** refere-se a palavras com muitas significações.
- **Definição de** Ita**:** capacidade de uma função possuir várias implementações, todas elas denotadas pelo mesmo nome.
- Escolha da implementação:
  1. Número e tipo dos parâmetros (sobrecarga);

```
int f (int x, float *y) { ..... }

int f (int x) { ..... }
```

  2. Forma do objeto (funções virtuais);

```
class A { .....
  int f (int x) { ..... }
}

class B: A { .....
  int f (int x) { ..... }
}
```

  3. Estado do objeto.

POLISSEMIA POR ESTADO

- Métodos de mesma assinatura, mas com implementações diferentes em função do estado do objeto.
- Estado: conjunto de valores dos atributos do objeto.
- Variável de Estado: interpretação.
- Importar para orientação por objetos parte da inteligência das máquinas de inferência da programação lógica.

EXEMPLO DE POLISSEMIA POR ESTADO

```
class Hotel {
   char mes;
   state Temporada=((mes <= 3) !!
                    (mes >= 11));
   state NaoTemporada= !Temporada;
 public:
   float CalculaDiaria() at Temporada
   { ..... }

   float CalculaDiaria() at NaoTemporada
   { ..... }
 }
```

- Estados podem ser redefinidos em subclasses.

```
 class HotelSerrano: Hotel {
     state Temporada=((mes >= 6) &&
                      (mes <= 8));
     ......
  }
```


POLIVALÊNCIA

- **Segundo o Aurélio:** que oferece diversas possibilidades de emprego.
- **Definição de** Ita**:** capacidade de uma função, com uma única implementação, trabalhar com um conjunto infinito de tipos (respeitada a estrutura hierárquica).

```
void f (A a) { .... }
  // A classe

void g (T t) { .... }
  // T parametro generico
```

REDEFINIÇÃO DE MÉTODOS

- Suponha uma classe `A` com um método `m`:

$$m : a_1 \times a_2 \times a_3 \cdots \times a_n \rightarrow r$$

- Suponha que `B`, subclasse de `A`, redefina `m`:

$$m : b_1 \times b_2 \times b_3 \cdots \times b_n \rightarrow s$$

- Relacionamento entre a_i e b_i ?
- Três possibilidades:
  - Covariância: $b_i \preceq a_i$ e $s \preceq r$ (Eiffel)
  - Contravariância: $b_i \succeq a_i$ e $s \preceq r$ (Sather)
  - Invariância: $a_i = b_i$ e $s = r$ (C++)
- Regra mais flexível: covariância.

```
class Xi
   .....
   IsEqual (arg: Xi) is BOOLEAN
     -- Covariancia
   do .....
   end -- IsEqual
   .....
end -- class Xi
```


COVARIÂNCIA X SEGURANÇA

```
class A
  feature
    f (arg: A) is ....
 end -- A
class B inherited A redefine f
  feature
    f (arg: B) is ....  -- covariancia
end -- B
class C
  .....
  a1, a2: A;  b1: B;
  !!a2;  !!b1;
  if ... then
     !!a1;
  else
    a1:= b1;
  end;
  a1.f (a2);    -- ERRO ! (se a1 = b1)
  .....
end -- C
```

- Eiffel (versão 2) tem uma falha no sistema de tipos devido ao uso de covariância.

CONTRAVARIÂNCIA GUARDADA

- Eiffel 3:
  - Conjunto de Classes Dinâmicas;
  - Validade a Nível de Sistema.
- Ita: Regra da Contravariância Guardada
  - Uso de contravariância;
  - Verificação dinâmica de tipos ao estilo de Oberon-2.

```
class A { .....
  public:
    char IsEqual (A a) { .... } ....
}

class B: A { .....
  public:
    char IsEqual (A a) {
      B b;
      if (a is B) {     // type test
         b= (B) a;      // type guard
         .....
      } else return FALSE;
    } .....
}
```


PROGRAMAÇÃO POR CONTRATO

- Asserções: pré-condições, pós-condições e invariantes.

```
class A {
  int a;
  .....
  int f (int x)
  pre (x > 0)       // pre-condicao
  pos (result > 0)  // pos-condicao
  { .....
    check NEGATIVE (...);
    .....
  }
  rescue {
    if (exception == NEGATIVE)
       .....
    else .....
    .....
    retry;
    .....
  }
  invariant: (a >= 0)   // invariante
}
```

TRATAMENTO DE EXCEÇÃO

- Violação de asserção produz exceção.
- Exceção é tratada por um bloco `rescue`.
- Semântica de retomada: método executa com sucesso ou falha.
  - Com sucesso, quando cumpre o seu contrato;
  - Com falha, quando o descumpre.
- Identificação da exceção: `exception`

CLASSES ABSTRATAS

- Classes que não possuem instâncias;
- Podem possuir métodos postergados (`deferred`).

```
abstract class A {   // classe abstrata
  .....
  deferred void f (int x)
  pre (x > 0) pos (...);
  .....
}
```

- Ferramenta de projeto orientado por objetos.

ESTRUTURA DE PROGRAMA

- Especificadores de acesso: `public` e `private`.

```
// arq1.ita
  .....
public:
  .....
private:
  .....
```

- Geração automática de arquivos .ih.

ESTUDO DE CASO

- Implementação de alguns TAD.

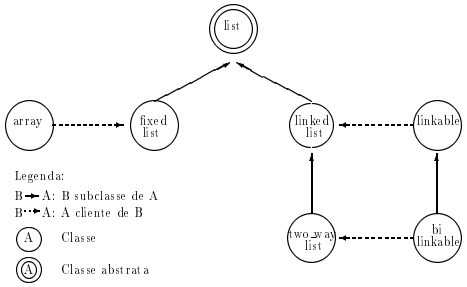

Figura 3: Hierarquia de classes implementada

- Vantagens:
  - Classes Abstratas: projeto de classes;
  - Classes Genéricas: flexibilidade;
  - Herança: alto grau de reuso;
  - Encapsulamento: coerência com TAD;
  - Asserções: modelagem de propriedades semânticas.

O COMPILADOR DE Ita

- Objetivos:
  - Prático: utilizar a linguagem;
  - Validar a linguagem.
- Versão atual para MS-DOS.
- Processo Tradicional de Compilação:

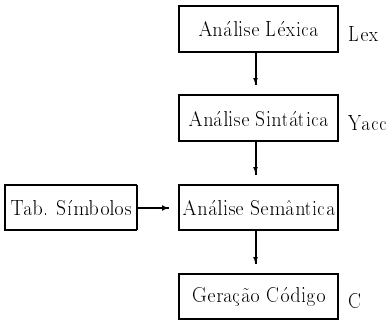

Figura 4: Processo de Compilação

GERAÇÃO DE CÓDIGO

newslideGeração de Código para Classes

- Classes ⇒ estruturas
  Referências ⇒ ponteiros
- Descritor de Classe (DC): tabela inserida no código gerado para dar suporte à chamada dinâmica de métodos.
- *Layout* de uma classe com atributos $< a_1, a_2, \ldots, a_m >$, métodos $< m_1, m_2, \ldots, m_n >$ e cujas superclasses possuem descritores $< s_1, s_2, \ldots, s_p >$:

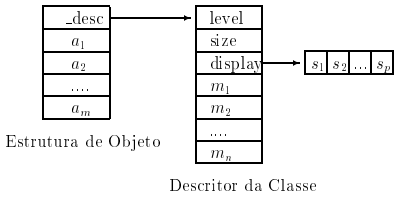

Figura 5: *Layout* de classes

newslideChamada Dinâmica de Funções

- Em Ita: `a.f(10);`
- Em C: `(*(a->_desc->f)) (a, 10);`

GERAÇÃO DE CÓDIGO PARA CLASSES GENÉRICAS

- Parâmetros genéricos são traduzidos para:
  - `void *` (se irrestritos);
  - tipo restrigente (se restritos).
- Em Ita:

```
class A <T1, T2: B> {
  T1 x;
  T2 y;
  T1 f1 (T1 p1, T2 p2) { ..... }
}
```

- Em C:

```
struct _A_class {
  struct _A_descriptor *_desc;
  void *x;
  B y;
}
.....
void *_A_f1 (A self, void *p1, B p2)
{ ..... }
```


PRINCIPAIS CONTRIBUIÇÕES

- Em relação a C++, Eiffel e Oberon-2:
  - Definições claras para conceitos como polimorfismo, polissemia, polivalência, funções inicializadoras e finalizadoras;
  - Solução para o impasse entre covariância e contravariância através de verificação dinâmica de tipos;
  - Conceito de encapsulamento privilegiando programação em larga escala;
  - Classes "inteligentes" através da definição de funções polissêmicas por estado.
- Especificamente em relação a C++:
  - Semântica de referência;
  - Implementação de classes genéricas através de uma semântica de código único e com verificação total de tipos;
  - Programação por contrato;
- Especificamente em relação a Eiffel:
  - Abordagem pragmática para a política de redefinição de asserções em subclasses;

Bibliografia

- Valente, Marco Túlio O. & Bigonha, Roberto S., *A Linguagem de Programação ITA*, RT 005/96, Departamento de Ciência da Computação, UFMG, 1996.
- Stroustrup, Bjarne, *The C++ Programming Language*, Addison-Wesley Publishing Company, Second Edition, 1991.
- Ellis, M. e Stroustrup, B. *The annotated C++ reference manual*, Addison-Wesley, 1990.
- Dantas, João E. R. *Comunicação Pessoal*, DCC/UFMG, Junho 1994.
- Meyer, Bertrand, *Object-oriented Software Construction*, Prentice-Hall International Series in Computer Science, C.A.R. Hoare Series Editor, 1988.
- Meyer, B. *Eiffel the language.* Prentice-Hall, 1992.
- Omohundro, S. *The Sather 1.0 specification.* International Computer Science Institute, Berkeley, 1994.