Projeto de uma Linguagem Orientada a Objetos

Este exemplar corresponde a redação final da tese devidamente corrigida e defendida pelo Sr. José de Oliveira Guimarães e aprovada pela Comissão Julgadora.

Campinas, 16 de Setembro de 1992.

Prof. Dr. Hans Kurt Edmund Liesenberg
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de Mestre em Ciência da Computação.

Projeto de Uma Linguagem Orientada a Objetos[1]

José de Oliveira Guimarães[2]
Departamento de Ciência da Computação
IMECC - UNICAMP

15 de setembro de 1992

[1]Este trabalho contou com o suporte financeiro da CAPES de Março de 1990 a Março de 1992
[2]e-mail: jose@pink.ifqsc.usp.br

José de Oliveira Guimarães

Projeto de Uma Linguagem Orientada a Objetos

Dissertação apresentada ao Curso de Mestrado em Ciência da Computação da Universidade Estadual de Campinas, em cumprimento às exigências para obtenção do grau de Mestre.

Área de Concentração: Linguagens de Programação.

Orientador: Prof. Dr. Hans Kurt Edmund Liesenberg

Projeto de Uma Linguagem Orientada a Objetos

José de Oliveira Guimarães

Examinador: Hans Kurt Edmund Liesenberg (Dr.)

Examinador: Caetano Traina Junior (Dr.)

Examinador: Rogério Drummond (Dr.)

Examinadora: Cláudia Bauzer Medeiros (Dra.)

Esta tese é dedicada a meus pais,
Belchior e Dalila.

Sumário

A orientação a objetos é um mecanismo que permite o reaproveitamento de *software*, tendo por isso despertado grande interesse nos últimos anos. Este paradigma tem sido utilizado em várias áreas da computação, como banco de dados, análise de sistemas e linguagens de programação.

Esta dissertação é dividida em duas partes. A primeira estuda a tecnologia existente sobre orientação a objetos e linguagens de programação. São analisados os mecanismos presentes nas linguagens orientadas a objeto e os objetivos destes mecanismos. Alguns problemas com o paradigma são considerados, apresentando as possíveis soluções, quando existirem.

A segunda parte da dissertação apresenta construções que extendem C++, e são justificadas com base no estudo feito na primeira parte.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Conceitos Básicos de Orientação a Objetos

Entende-se normalmente como sendo linguagens orientadas a objeto aquelas que suportam herança ou delegação. Delegação não será discutida nesta dissertação. Linguagens com herança possuem classes e objetos. Um objeto é um agrupamento de dados mais as operações sobre estes dados. Estas operações afetam o estado dos dados, que são a memória do objeto [Weg87a]. Classes são *templates* (tipos) a partir dos quais os objetos são criados. Objetos da mesma classe suportam as mesmas operações e possuem o mesmo formato para os seus dados internos. As operações suportadas por um objeto são chamadas de métodos. Um objeto é uma instância de uma classe, do mesmo modo que 3 é uma instância do tipo inteiro. Os dados do objeto, via de regra, só são manipulados pelos métodos associados à classe da qual foram declarados. A figura 1.1 mostra, graficamente, a classe `Pessoa`, com os métodos `setnome`, `print` e `M`. A representação de `Pessoa`, composta por `nome` e `RG`, só pode ser manipulada pelos métodos da classe. As variáveis definidas em uma classe são chamadas variáveis de instância, pois cada objeto (instância) conterá um conjunto destas variáveis. Um objeto é estimulado por meio de mensagens, que são compostas por um seletor e eventuais argumentos. Um seletor geralmente é o **nome** de um método. Se variável `P` é da classe `Pessoa`, a instrução ``P.print(5)'' é o envio da mensagem `print(5)` ao objeto associado a `P` durante a execução. O seletor desta mensagem é `print` e o argumento é 5. Ao ser

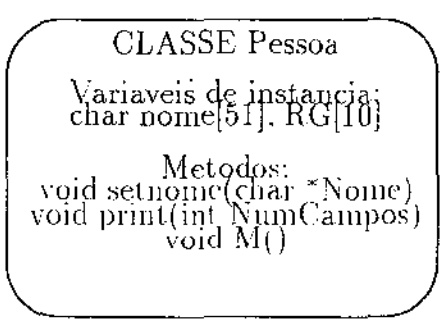


Figura 1.1: Classe Pessoa

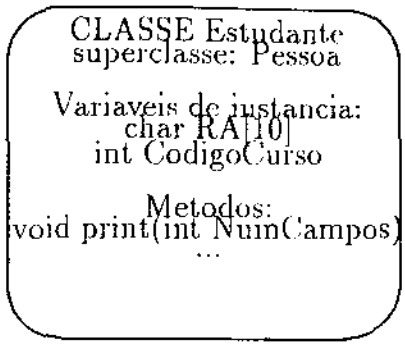


Figura 1.2: Classe Estudante

enviada para um objeto, a mensagem causará a execução de um método. Consideraremos que o nome do método é o mesmo que o do seletor da mensagem - `print`, neste caso.

O mecanismo de herança permite a especialização de uma classe através da criação de uma subclasse. Se classe `B` herda da classe `A`, `B` possuirá variáveis de instância de mesmo nome e mesmos métodos que `A` e possivelmente outras variáveis e métodos definidos especificamente para ela. Considere que a classe `Estudante` herda de `Pessoa`, na figura 1.2. `Estudante` possui variáveis de instância `RA` e `CodigoCurso`, bem como métodos `setnome` e `print`. O método `setnome` é herdado de `Pessoa`. O método `print` é o específico de `Estudante`. O objeto que recebe uma mensagem é referenciado por `this` em C++, `self` em Smalltalk e `Current` em Eiffel [Mey87] [Mey88]. Algumas linguagens possuem uma classe pré-definida, geralmente chamada de `Object`, da qual todas as outras herdam características básicas, mesmo que esta herança não seja explícita.

Considere que objeto da classe `Estudante` receba uma mensagem `M`. O Método `M` é invocado e envia mensagem `print` para `this`. Qual método `print` é utilizado, o de `Pessoa` ou `Estudante`? O método utilizado em resposta a uma mensagem é procurado na classe do objeto. Se ele não for encontrado, procura-se na sua superclasse. Se ainda não é encontrado, a busca continua na superclasse da superclasse e assim por diante. Neste caso, utiliza-se o método `print` de `Estudante`. A busca começa na classe `Estudante` e aí é encontrado método `print`.

A ligação entre a mensagem recebida e o método a ser utilizado é precoce[1] ou tardia[2]. A ligação precoce acontece quando a busca pelo método adequado é feita pelo compilador, que descobre o método a ser utilizado em tempo de compilação. A ligação tardia acontece quando a busca pelo método acontece em tempo de execução. Os motivos pelos quais é empregada um ou outro tipo de ligação serão esclarecidos no capítulo seguinte.

1.2 Observações Sobre a Tese

Após as correções sugeridas pela banca examinadora (e mesmo antes), o título desta dissertação se tornou inadequado. Não é projetada uma linguagem orientada a objetos com-

[1] *early binding*
[2] *late binding*

pleta, apenas é sugerida uma extensão a C++. Infelizmente, questões burocráticas impediram a alteração do título.

A dissertação consiste em um estudo sobre orientação a objetos (capítulos 2, 4) e uma sugestão de extensão a C++ (capítulo 5). O conteúdo preciso de cada capítulo é dado abaixo.

- Capítulo 1. Esta introdução.
- Capítulo 2. Definição de facilidades presentes em orientação a objetos, como herança, abstração de dados, classes abstratas. São discutidos os fatores que fazem herança adequada para reaproveitamento de *software*. Alguns modelos de herança são apresentados.
- Capítulo 4. Traz uma discussão sobre classes parametrizadas: definição, requerimentos, erros de tipos. É exposto uma forma de simular classes parametrizadas com herança.
- Capítulo 3 Exposição de três problemas originais com orientação a objetos.
- Capítulo 5. É apresentada uma sugestão de extensão a C++ baseada nas discussões dos capítulos precedentes.
- Capítulo 6. Conclusão.

Capítulo 2

Classes, Objetos e Herança

O mecanismo de herança múltipla permite que uma classe herde características de mais de uma superclasse. Ela é necessária quando uma classe é uma especialização de duas ou mais classes, devendo herdar características de todas elas. Por exemplo, uma classe `Funcionário` que herda da classe `Pessoa` e da classe `Trabalhador`. Um funcionário é, ao mesmo tempo, uma pessoa e um trabalhador. Da classe `Pessoa` `Funcionário` herda características como idade e nome. Da classe `Trabalhador`, herda tempo de serviço e empregador. Seja `C` uma classe que herda das superclasses S_1, S_2, ..., S_n. Se duas superclasses de C, digamos S_i e S_j ($i \neq j$), definem método com mesmo nome M, qual método M a classe C deve herdar ? M de S_i ou M de S_j ? Este problema é chamado `colisão de nomes`. Linguagens de programação distintas resolvem colisão de nomes de forma diferente. Em C++, as superclasses são ordenadas de acordo com a ordem em que elas são citadas na especificação de uma classe. Se ocorre colisão de nomes entre as superclasses S_i e S_j, o método herdado é o da classe que é citada primeiro na declaração de superclasses. A linguagem Eiffel [Mey88] permite que uma classe mude o nome (renomeie) o nome de um método herdado de uma superclasse. Dentro da classe, este método só é utilizado com o seu novo nome. O nome antigo deixa de existir. Imagine classes `A` e `B` que possuem método `M`. Quando uma classe `C` herda de `A` e `B`, ela pode mudar o nome de `M` herdado de `B` para `Q`. Deste modo, classe `C` herda `M` de `A` e `Q` de `B`, não acontecendo colisão de nomes. Suponha que `M` e `Q` são redefinidos em `C` e que classe `A` possua método `H` que sempre envia mensagem `M` para `self` (o objeto que recebeu a mensagem com seletor `H`). Quando um objeto de `C` recebe mensagem com seletor `H`, o método `H` de `A` é executado e envia mensagem com seletor `M` para o próprio objeto. O método utilizado em resposta a esta mensagem é `M` de `C`. Suponha agora que `B` defina método `G` que sempre invoca método `M`. Em objeto de `C`, execução de `G` causa sempre a execução de método `Q` de `C`, pois este é o correspondente, em `C`, de `M` de `B`. Renomeação permite a especialização de ambos os métodos `M` herdados de modos distintos. Em C++, isto não é possível. Se `M` é redefinido em C, o método `M` de `C` servirá como redefinição de `M` de `A` e `M` de `B`.

Como apresentado até agora, o método invocado em resposta a uma mensagem enviada a um objeto é procurado nos métodos da classe e superclasses do objeto. Polimorfismo múltiplo[1] considera não só o objeto que recebe a mensagem, mas também as classes de seus

[1] Polimorfismo será definido na seção 2.4. A compreenção deste parágrafo não exige o conhecimento de outras

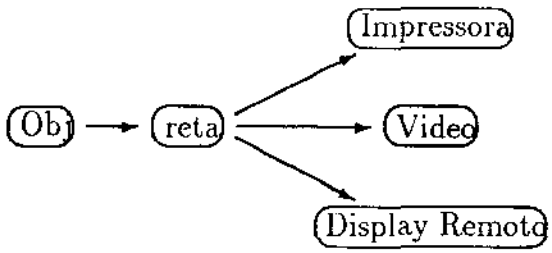


Figura 2.1: Método invocado depende também dos parâmetros

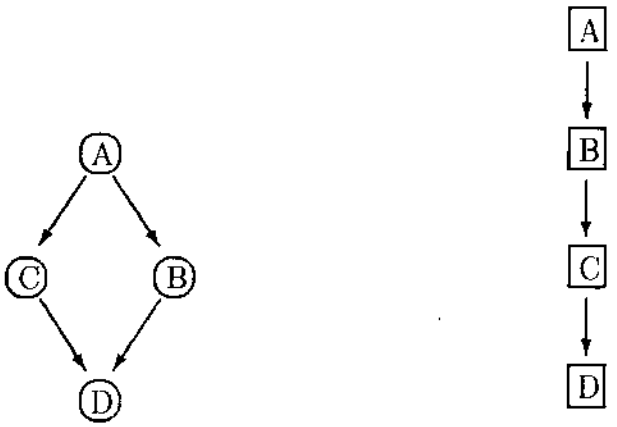
parâmetros. Isto é, a escolha do método é baseado na classe e superclasses do objeto que recebeu a mensagem e também nas classes e superclasses dos parâmetros das mensagens. Uma conseqüências deste fato é que um objeto pode executar dois métodos diferentes para mensagens de mesmo nome, desde que os parâmetros destas mensagens sejam de classes correspondentemente diferentes. Tomemos um exemplo citado em [Ing86]: há objetos gráficos (retas, retângulos, círculos) e dispositivos para saída gráfica (impressora, vídeo, display remoto). Para cada objeto gráfico, deve existir método que visualiza o mesmo em cada um destes dispositivos. Considere ''`imprima`'' como sendo o nome da mensagem para visualizar um destes objetos, com um parâmetro que é um dispositivo. Para uma classe, digamos, `reta`, existem três métodos com nome `imprima`. Cada um deles possue um único parâmetro, que é do tipo `Impressora`, `Video` ou `DisplayRemoto`. No exemplo abaixo, o método invocado pelo envio de mensagem acima é decidido em função da classe de `xx` (no caso, `reta`) e da classe de `dd` (figura 2.1).

```
var  xx : reta;
...
xx.imprima(dd);
```

Em C++, existe este tipo de polimorfismo, mas a classe dos parâmetros deve ser conhecida em tempo de compilação.

As principais dificuldades em programação orientada a objetos é decidir quais são as classes e quando usar herança. As classes representam entidades reais (Pessoa, Estudante, Empregado) ou abstratas (Pilha, Arquivo) de um sistema. Qualquer agrupamento de dados com operações sobre os mesmos pode, em geral, ser encapsulado como uma classe. Uma classe não necessariamente define variáveis de instância. Se o usuário a utiliza apenas por suas operações, é indiferente para ele o número e a natureza de suas variáveis. Um erro comum é colocar componentes (variáveis de instância) de uma classe como superclasses. Como exemplo, colocar `Asa` e `Corpo` como superclasses da classe `Aviao`. A classe `Corpo` modelaria o corpo de um avião. Este erro pode ser evitado perguntando se objeto da classe também é objeto da superclasse: um avião é uma asa ? O correto é modelar `Asa` e `Corpo` como variáveis de instância de `Aviao`. Liskov [Lis87], Halbert e O'Brien [HO88] apresentam algumas diretrizes para o uso de herança e classes.

formas de polimorfismo



(a) (b)

Figura 2.2: (a) Hierarquia de classes (b) Linearização

A representação gráfica de classes e herança é feita considerando classes como círculos com o seu nome no interior. A relação de herança entre A e B é indicada como aresta de A para B (B herda de A). As variáveis de instância de uma classe A são representadas como um retângulo com a letra A em seu interior. Se é colocada uma flexa do retângulo A para retângulo B, então B é subclasse de A.

A figura 2.2 (a) fornece uma hierarquia de classes utilizada na discussão seguinte. Os círculos e as flexas de relação de herança formam um grafo digirido acíclico, chamado grafo de herança.

Existem 3 formas de tratamento da hierarquia de classes:

Linearização O grafo é reduzido a uma lista linear através da sua ordenação topológica. O problema de colisão de nomes deixa de existir, pois a cada classe é associada uma única superclasse (lista linear). Esta forma é ilustrada pela figura 2.2 (b). Se classe `A` define variável de instância `z`. haverá apenas um exemplar de `z` em objetos de `D`. Se classe `B` define uma variável de nome `k` e classe `C` também define variável de nome `k`, haverá apenas uma variável `k` em objetos de `D`. Admite-se que não se especifique tipos na declaração de variáveis, pois os tipos das duas variáveis `k` poderiam ser diferentes.

Grafo As variáveis de uma classe que pode ser atingida por mais de um caminho não são duplicadas. Na figura 2.3 (a), haverá apenas um conjunto de variáveis definidas em A em cada objeto de D.

Árvore As variáveis de instância de uma classe são duplicadas se existe mais de um caminho para ela com início na classe mais especializada (mais embaixo) da hierarquia. No exemplo da figura 2.2, haverá dois conjuntos de variáveis de `A` em objetos de `D`, correspondendo a dois caminhos de `D` para `A` (`D` → `C` → `A` e `D` → `B` → `A`). Figura 2.3 (b) mostra esta estratégia. Na figura, se `A` define variável `x`, então haverá duas variáveis `x` de `A` em objetos de `D`.

(a) (b)

Figura 2.3: (a) Objeto de D (b) Objeto de D

Com linearização, a busca por um método causada pelo envio de uma mensagem segue esta lista. Não existe um algoritmo padrão para a busca por um método em linguagens que adotam a estratégia em árvore ou grafo.

O ensino de orientação a objetos é tratado em [KM87]. Os autores sugerem que o estudo desta matéria não se baseie em linguagens, e sim nos requesitos e princípios de orientação a objetos. As linguagens devem ser analisadas em relação ao nível de suporte ao paradigma. A dificuldade maior em aprender a utilizar um sistema (linguagem + ambiente + bibliotecas) orientado a objetos está na biblioteca de classes (podem chegar a centenas de classes) e no conceito de metaclasse [Tim86], discutido adiante. Os ambientes que acompanham a linguagem desempenham um papel fundamental no aprendizado e produção de software. Eles, geralmente, permitem visualizar uma classe como ela é, já agrupando aos seus próprios métodos, os das superclasses. Sem isto, é difícil até mesmo a saber quais as mensagens um objeto da classe pode responder.

A seguir são apresentadas algumas definições e sinônimos, que serão úteis nas discussões seguintes.

Procedimento Rotina, subrotina, procedimento são considerados sinônimos. Alguns autores chamam métodos de rotinas. A passagem de parâmetros `in` é a passagem por valor. A passagem `in out` implica que o parâmetro real é copiado para o formal antes do início da execução do procedimento (`in`) e o formal copiado para o real após a execução do mesmo (`out`). Observe que o tipo de passagem `in out` de Ada não possue definição rigorosa e, portanto, não necessariamente é igual à aqui apresentada.

Operação Método ou operações aritméticas ou lógicas como +, *, >, <= . Muitas vezes utiliza-se a expressão "invocar uma operação" e não "invocar um método".

Referenciar, Referir-se A declaração de variáveis em algumas linguagens não garantem a alocação de memória para estas variáveis. Elas são semelhantes a ponteiros em linguagens procedimentais. Por este motivo, diz-se que variáveis desta natureza apenas referem-se a objetos.

Figura 2.4: Subclasses diretas e indiretas

:: Este operador indica a origem de um método. O método M da classe A é indicado por A::M.

Superclasse direta Considere a hierarquia da figura 2.4. `B` é a superclasse direta de `C`. `A` é a superclasse direta de `B`. `A` é superclasse de `C`, mas não direta.

Atribuição Associação de um valor ou objeto a uma variável que já existe. Feita com := em Pascal e = em C++.

Inicialização Associação de um valor a uma variável na sua criação. Algumas linguagens permitem especificar valores padrões para as variáveis de instância de uma classe. Quando um objeto desta classe é criado, estes valores são automaticamente copiados neste objeto. Isto é uma inicialização. A linguagem C++ permite definir um método para fazer a inicialização de objetos de uma classe. Este método é o construtor da classe.

Representação, Especificação e Implementação A representação de uma classe são as estruturas de dados (variáveis de instância) utilizadas por ela. A especificação de uma classe são a relação das suas superclasses diretas, os protótipos de seus métodos públicos e variáveis de instância públicas. A especificação contém informações suficientes que definem a visão externa da classe e dos objetos dela derivados. Um protótipo de um método é definido pelo seu identificador, seus parâmetros e o tipo do objeto retornado como resposta da mensagem que disparou o método em questão. Interface de uma classe é o mesmo que sua especificação. A implementação de uma classe consiste do código dos seus métodos.

Ancestrais e Descendentes Ancestrais e descendentes são, respectivamente, as superclasses e subclasses de uma classe.

Clientes de uma classe Os clientes de uma classe são:

- Os procedimentos ou classes que declaram variáveis deste tipo. Exemplo: `C::M` e `Q` da figura 2.5 são clientes da classe `A`.
- Subclasses da mesma. Exemplo: `B` da figura 2.5 é cliente da classe `A`.

```
class A { ... } ;
class B : A { ... } ;
class C { ...
  void M() { A a ; ... }
  };
char Q() { A x ; ... }
```

Figura 2.5: `B`, `Q` e `C` são clientes classe `A`

2.1 Proteção de Informação

Algumas linguagens orientadas a objeto como BETA [KOLM87], Flavors [Moo86] e CLOS [DG88], não oferecem nenhum encapsulamento. Qualquer cliente pode manipular as variáveis de instância de um objeto. C++, Trellis/Owl [S+86] e Simula[2] [Dah73] [Mey88] possuem 3 níveis de visibilidade. Os membros (variáveis de instância e métodos) de uma classe podem ser:

- Públicos: visíveis para qualquer cliente.
- Protegidos: visíveis pelas subclasses, mas não pelos que apenas declaram objetos da classe. Métodos virtuais protegidos devem ser utilizados quando eles são apropriados apenas para subclasses e não para outros clientes.
- Privados: visíveis apenas pelos métodos da classe.

C++ permite que uma função F ou uma classe A manipule membros privados ou protegidos de uma classe B. Para isto, B deve declarar, na sua interface, F ou A como *friend*. Esta facilidade introduz uma violação controlada do encapsulamento. Qualquer alteração na parte privada ou protegida de B pode exigir a modificação dos métodos de B e de todas as funções e classes *friends*. Afinal, estes últimos podem (devem) manipular a parte privada de B. Quando os membros privados de B são alterados, o programador sabe quais os métodos, classes e funções que podem necessitar de modificações: são os métodos de B e os *friends* de B. Por isto, subclasses de A não possuem os mesmos direitos de acesso a B que A possui, apesar de os descendentes de A provavelmente necessitarem de manipular dados privados de B.

A figura 2.6 ilustra a utilidade do nível protegido de proteção de informação. A classe `complex` possui método virtual protegido `printStr`. Este método é utilizado por `print` para imprimir os dados do número complexo já formatado. `printStr` claramente não deve ser público, pois não é uma operação sobre números complexos. A classe `xcomplex`, subclasse de `complex` redefine `printStr` para imprimir *string* em terminal gráfico. Este exemplo mostra a importância da fase de análise em orientação a objetos e porque ela é mais difícil que em programação procedimental. O programador previu que subclasses necessitariam

[2] Versões mais recentes

```
class complex {
  double re, im;
  protected:
    virtual void printStr(char *s) { printf("%s",s); }
  public:
    ...
    virtual void print();
  };
class xcomplex : public complex {
 ...
 protected:
   virtual void printStr(char *s) { ... }
 };
```

Figura 2.6: Classe `complex` com método protegido `printStr`

de redefinir a rotina de impressão e a forneceu como método protegido. Neste exemplo, `print` seria facilmente refeito em `xcomplex`. Mas em outras ocasiões isto seria muito mais complicado.

Em Smalltalk [GR83], qualquer classe pode manipular as variáveis de instância de sua superclasse. Se a superclasse muda o nome de uma variável de instância, ocorrerá erro em uma subclasse que utiliza essa variável.

Um objeto `X` recebe uma mensagem que causa a execução de um método. Em Smalltalk ou Emerald [RTL+91], este método pode manipular apenas as variáveis de instância do objeto `X`. Não é permitido que ele manipule as variáveis de instância de um dos parâmetros da mensagem, mesmo que este parâmetro seja um objeto da mesma classe que `X`. Conseqüentemente, em Smalltalk e Emerald todos os parâmetros são manipulados exclusivamente por meio de mensagens. Por este motivo, muitos métodos precisam ser criados para a manipulação de variáveis de instância da classe. Em geral, são criados dois métodos para cada variável de instância : um para retornar (ler) o valor da variável e outro para modificar a variável (escrever). Em Smalltalk, o tipo dos parâmetros do método só são conhecidos em execução. Em Emerald, várias versões de uma mesma classe (com representação – variáveis de instância – diferentes) podem estar presentes simultaneamente na execução de um programa.

A figura 2.7 mostra as classes `Fila` e `ElemLista`. Esta última é a classe dos elementos de `Fila`. Admite-se o uso de uma linguagem hipotética onde as variáveis de instância dos parâmetros não podem ser manipuladas diretamente (como Smalltalk). O método `append` da classe `Fila` concatena a fila que é o seu parâmetro ao final da fila que recebeu a mensagem `append`. A classe `Fila` é implementada como uma lista encadeada de elementos da classe `ElemLista` e contém ponteiros para o início e fim da lista. O modo mais simples e eficiente de implementar `append` é fazer com que o último elemento da fila aponte para o primeiro elemento da fila que é parâmetro e vice-versa. Para tanto, a classe `ElemLista` fornece os métodos `setsucc` e `setpred`, para atribuir valores aos ponteiros para o sucessor e predeces-

```
class ElemLista
  valor : integer;
  v_succ, v_pred : ElemLista ;
  methods
    setsucc( x : ElemLista) is
      begin
      v_succ = x ;
      end
  ...
end -- classe ElemLIsta

class Fila
  first, last : ElemLista ;
  ...
  methods
    insere( e : integer) : boolean is  ...
    append( L : Fila ) is
      begin
      last.setsucc( L.Primeiro );
      L.Primeiro.setpred( last );
      end
    Primeiro : ElemLista is
      begin
      return first;
      end
    ...
end -- classe Fila
```

Figura 2.7: Fila de inteiros em uma linguagem altamente polimórfica. Apenas métodos são exportados

```
TYPE Stack(C)
  PROPERTY  LIFO
  METHOD  get():C
  METHOD  put(C)
END Stack
```

Figura 2.8: Mecanismo de tipo de POOL-I

sor de um objeto de `ElemLista`. O método `append` precisa de uma operação que retorna um apontador para o primeiro elemento de seu parâmetro `L`. O protótipo desta operação, `Primeiro`, deixa transparecer que `Fila` é implementada como lista encadeada e não como um vetor, por exemplo. O único local onde classe `ElemLista` é usada na interface de `Fila` é neste método. Em um lugar qualquer fora da classe `Fila`, pode-se obter o primeiro elemento de uma fila através do método `Primeiro`. Com o uso dos métodos `setsucc` e `setpred` de `ElemLista`, é possível alterar os ponteiros do primeiro elemento da fila, danificando-a. Concluindo, há uma violação do encapsulamento da classe `Fila`. Obviamente, a operação `append` poderia ser implementada tomando-se elemento por elemento de `L` e inserindo-os ao final da fila que recebeu a mensagem `append`. Mas isto seria extremamente ineficiente.

Em Self [CUL89], todas as variáveis de instância são manipuladas por meio de funções, mesmo em métodos do próprio objeto. Normalmente, existe uma função que retorna (*read*) e outra que modifica (*write*) o valor de cada variável. Em Self, apenas os métodos para ler e escrever em uma variável de instância sabem a representação (estrutura) desta variável. Assim, modificações na representação de uma variável afetam apenas dois métodos: o que lê e o que escreve um valor na variável. Esta visão de variáveis de instância aumenta o grau de proteção de informação: a representação de uma variável é escondida de todos os métodos da classe, exceto de dois deles.

2.2 Subtipo e Subclasse

Nas discussões desta seção consideraremos que duas classes são o mesmo tipo se membros (variáveis de instância e métodos) públicos de uma possuem os mesmos nomes que os membros da outra. Classes são utilizadas na criação de objetos. A classe A é subtipo da classe B se um objeto de A pode ser usado onde um objeto de B é esperado. BETA [KOLM87] e Modula-3 [C+88] confundem subtipo e subclasse. Se uma classe B é subtipo de classe A, então B é subclasse de A. E se B é subclasse de A, então B é subtipo de A. A linguagem C++ permite que uma subclasse escolha, na herança, ser ou não ser subtipo de cada uma de suas superclasses. Uma classe em Eiffel é sempre subtipo de seus ancestrais, mesmo que ela não permita que seus usuários utilizem os métodos das superclasses. Snyder [Sny87] [Sny86] sugere que a relação de subtipo seja definida independentemente da relação de subclasse, e pelo programador. POOL-I [AvdL90] permite definir propriedades associadas a tipos. Propriedades são nomes listados na interface de uma classe sem outra finalidade senão para o uso na conferência de tipos. B é subtipo de A se B possui pelo menos os métodos e propri-

```
class A {
  // Sintaxe de C++
  virtual T M( R1 a);
  void N() {
    R1 x;
    T y;
    ...
    y = M(x);
    }
  };
class B : public A {
  ...
  virtual T1 M( R a);
  };
```

Figura 2.9: Herança e especialização de métodos

edades de mesmo nome que os de A. Figura 2.8 mostra tipo parametrizado (classe) `Stack`, com propriedade `LIFO`. O objetivo deste mecanismo é impedir que um tipo seja considerado erroneamente subtipo de outro. Uma outra classe com os mesmos métodos que `Stack`, mas sem as suas propriedades, não é subtipo ou supertipo de `Stack`.

Uma definição formal de subtipos e sua semântica foi dada em [Car84], a qual é apresentada no próximo parágrafo.

A figura 2.9 mostra classes `A` e `B` e métodos `M` e `N` de `A`. A sintaxe é de C++, mas o mecanismo de passagem de parâmetros e atribuição permite que um objeto de um subtipo possa ser utilizado quando um objeto do tipo é esperado. A passagem de parâmetros é por valor (`in`). A classe `B` herda de `A`. redefinindo o método `M`, mas não método `N`. O método `A::M` possui um parâmetro formal `a` do tipo `R1` e retorna um valor do tipo `T`. O método `A::N` declara variáveis `x` e `y`, dos tipos `R1` e `T`, respectivamente. Neste mesmo método, há o envio da mensagem `M` para *self*, com parâmetro real `x`. Suponha que um objeto de `B` invoque o método `N` (recebe mensagem que causa a execução de `N`), que não é redefinido em `B`. A instrução `y = M(x);` deste método não será invalidada pela nova redefinição de `M` em `B`. A variável de instância `y` receberá um objeto que é subtipo do seu tipo (recebe `T1`, espera `T`. `T1` é subtipo e subclasse de `T`). Método `M` de `B`, que é o utilizado por `N`, receberá um parâmetro que é subtipo do esperado (recebe `R1`, espera `R`. `R1` é subtipo e subclasse de `R`). Os parâmetros de M em B devem ser supertipos dos correspondentes em A. O valor de retorno de M de B, se houver, deve ser subtipo do valor de retorno de M de A. Deste modo, clientes de A que invocam método M podem utilizar objetos de B sem problemas. A relação de subtipo foi definida por Cardelli [CW85]:

- Tipo é um conjunto de campos que são valores ou funções. Na terminologia de orientação a objetos, variáveis de instância e métodos. Cada valor e cada função possuem tipos. O tipo de uma função é dado pelo seu nome, número e tipos dos seus parâmetros

e seu valor de retorno.

- B é subtipo de A, escreve-se $B \leq A$, se os campos de mesmo nome de A e B estão na relação $\leq$ e B possui pelo menos os mesmos campos que A.
- Uma função F é subtipo da função G se
  - F retorna resultado que é subtipo do resultado de G
  - F aceita argumentos que são correspondentemente supertipos do argumentos de G

  Formalmente, uma função $F : \sigma \rightarrow \tau$ é subtipo de $G : \sigma' \rightarrow \tau'$ se:
  $\tau \leq \tau' \quad e \quad \sigma' \leq \sigma$
  Onde σ é o domínio de F – número e tipos dos argumentos. τ é o contra domínio – tipo do valor de retorno. Considera-se que os parâmetros são passados por valor. Passagens do tipo `in out` (veja definição página 8) devem considerar que o parâmetro é também um valor de retorno da função. Formalizando:
  $f(x : in\ out\ T) : U \Longrightarrow f(x : T) : (U, T)$
  onde (U, T) indica que a função f retorna um valor do tipo U e outro do tipo T.

A linguagem Eiffel possui um erro de tipos exatamente por que não obedece à relação de subtipos acima. A linguagem Trellis/Owl considera subtipos como definido por Cardelli.

Algumas vezes o mecanismo de herança é utilizado unicamente para o reaproveitamento de código através da criação de subclasses que não guardam uma relação semântica com suas superclasses [Weg87b]. Como exemplo, considere a classe PeixeMarinho e a sua subclasse Golfinho. A relação de subtipo não é válida semanticamente, apesar de ser considerada correta pela linguagem. Algumas relações entre tipos são discutidas em [WZ87].

As linguagens Sina/St [AT88] e Alphard [SWL77] permitem que uma subclasse retire (cancele) métodos da sua superclasse. Um exemplo, citado em [Sny86], é uma Stack (pilha) herdar Deque (fila com inserção e remoção nos dois extremos) e remover os métodos de inserção e remoção em um dos extremos. Esta facilidade não é recomendada por alguns autores [TL90] [Mey88] [Weg87b] porque diminui a clareza do software – subclasses deixam de ser subtipos. Snyder [Sny86] argumenta que herança é um mecanismo através do qual deve-se tentar reaproveitar código ao máximo e, portanto, esta facilidade é desejável em uma linguagem.

Seja X uma variável cujo tipo é a classe A e Y da classe B. A operação = faz com que duas variáveis passem a referenciar a um mesmo objeto (veja definição de referenciar na página 8). Considere a instrução `X = Y` e os possíveis relacionamentos entre A e B.

- Se A e B são a mesma classe ou B é subclasse de A então não é possível erro em execução. Imagine B como sendo a classe Estudante e A como classe Pessoa. Uma variável da classe Pessoa pode referenciar um objeto Estudante. Uma inicialização do tipo "Classe = Subclasse" é sempre correta. Todas as linguagens orientadas a objeto permitem este tipo de atribuição[3].

[3] C++ exige que `X` e `Y` sejam ponteiros

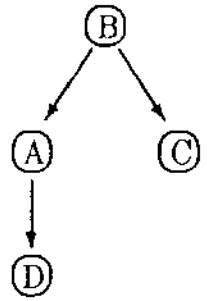


Figura 2.10: Hierarquia é utilizada para testes em execução

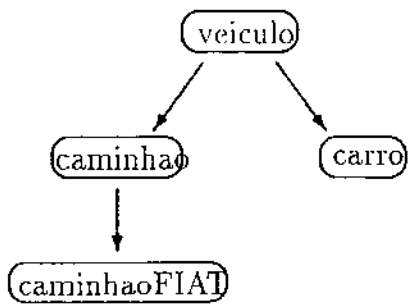


Figura 2.11: Hierarquia de veículos

- Suponha-se que agora A é subclasse de B. Eiffel, Trellis/Owl[4] e C++[5] consideram a instrução um erro. Em Simula, Modula-3 e BETA, o compilador insere código antes da atribuição, que fará um teste em execução para conferir os tipos de Y e X. Acontece um erro em tempo de execução se Y não se refere a objeto da classe B, A ou subclasse de A. Este tipo de atribuição é feito com o comando `assert` em VBASE [AH] e com operador ?= em Eiffel [MM90]. Esta abordagem torna o código mais fácil de ler, diferenciando atribuições normais das que podem causar erros em tempo de execução. Imagine C como sendo a classe carro, A caminhao e B veiculo. É um erro se a variável da classe caminhao passar a referenciar um objeto da classe carro, mas é correto ela referenciar um objeto caminhaoFIAT (subclasse de caminhao). Veja figura 2.11.

- Outras combinações entre classes A e B resultam em erro de atribuição. Um exemplo é A ser a classe Estudante e B a classe Veiculo.

Algumas linguagens possuem uma classe pré-definida, geralmente chamada `Object`, que é herdada implicitamente por todas as classes que não herdam de ninguém. `Object` é superclasse de todas as outras. Sempre que um objeto da classe `Object` é exigido, um objeto de qualquer classe pode ser usado. A classe `Object` possui apenas os métodos básicos (cópia, comparação igualdade/desigualdade, ...).

[4] Mais precisamente, esta linguagem considera a instrução um erro se A é **subtipo** de B

[5] Pode-se usar *cast* para fazer esta conversão

```
fun reverse L =
  let fun rev(nil, y) = y
        | rev(hd::tl, y) = rev(tl, hd::y)
  in
      rev(L,nil)
  end;
```

Figura 2.12: Função que inverte uma lista, em SML

```
print(int x);
print(float y);
...
p = NULL ;
```

Figura 2.13: Exemplo de polimorfismo em C++

2.3 Metaclasse

Algumas linguagens orientadas a objeto, como Smalltalk e DSM, consideram classes como objetos. Cada classe é um objeto único de sua metaclasse. Esta contém os métodos que são invocados em resposta a mensagens enviadas à classe correspondente, como *new*, para a criação de objetos da classe:
$a \leftarrow window\ new$
Outro método da metaclasse é o que responde o tamanho dos objetos de uma classe. Estes métodos referem-se a classes, não a instâncias de classes. Metaclasses é o conceito mais difícil de aprender em linguagens orientadas a objetos [WBW88] [Tim86]. Podemos imaginar uma classe para cada metaclasse, classes para estas classes e assim por diante, criando uma cadeia infinita. Em Smalltalk, a classe de todas as metaclasses é uma só, chamada `Metaclass`.

2.4 Polimorfismo

Função polimórfica é aquela em que pelo menos um parâmetro pode ter mais de um tipo [CW85]. Valores e variáveis polimórficos possuem mais de um tipo.

Cardelli e Wegner [CW85] classificam polimorfismo em duas grandes classes:

Universal Divide-se em

- Paramétrico. Uma função que trabalha uniformemente sobre infinitos tipos. O mesmo código é utilizado por todos eles. Um exemplo é uma função `sort` que ordena vetores de qualquer tipo, em Smalltalk. As classes dos elementos dos vetores devem ter algumas características em comum, como as operações $<$, $=$, $<=$, $>$. Os valores `NULL` de C++ e nil de Pascal apresentam polimorfismo paramétrico.

São compartilhados por todos os tipos que são ponteiros, mesmo os ainda não declarados. Ambos estão associados a infinitos tipos. A linguagem SML [Har] é baseada nesta categoria de polimorfismo. O programador raramente especifica tipos. O compilador infere o tipo mais genérico que pode ser utilizado em cada situação. Figura 2.12 mostra uma função que inverte listas e que é utilizada com listas de qualquer tipo. Nenhuma operação específica de um tipo é utilizada. Este tipo de polimorfismo ocorre em linguagens que permitem o uso de estruturas indexadas (*array's*) como parâmetro de subrotinas sem a especificação de limites:

```
void sort( int  vet[], int NumElementos)
{ ... }
```

Esta rotina ordena vetores com número variável de elementos. Algumas linguagens exigem o uso explícito da dimensão dos vetores, aumentando o grau de segurança e diminuindo o polimorfismo.

- Inclusão. Um valor de um subtipo pode ser utilizado onde um valor do tipo é esperado. É utilizado amplamente em linguagens orientadas a objeto, pois (em geral) subclasses são também subtipos.

Funções polimórficas universais podem ser utilizadas por infinitos tipos, desde que estes contenham operações requeridas pelas funções.

ad-hoc Divide-se em

- Sobrecarga[6]. Um mesmo nome é usado por várias funções diferentes, sem nenhuma necessidade da existência de uma estrutura semelhante. Por exemplo, as funções `print` da figura 2.13, em C++. Esta figura 2.13 mostra os cabeçalhos de duas funções `print`. A primeira com parâmetro inteiro e a outra com real. O código de uma é diferente do da outra.
- Coersão. Conversão de um tipo para outro antes de invocar uma operação. Esta facilidade não representa um polimorfismo característico de funções. Suponha que só exista a operação + entre reais. Na expressão 1 + 2.0, 1 (inteiro) é convertido para 1.0 (real) e então é feito uma soma entre dois reais. A distinção entre coersão e sobrecarga pode ser obscura. Suponha que exista soma entre inteiros e entre reais. A soma 1 + 2.0 admite duas possibilidades de conversão antes da operação:
  - 1 é convertido para real ou
  - 2.0 é convertido para inteiro

  O uso indiscriminado de coersões pode tornar o programa difícil de entender, além de haver perda de informação em alguns casos. Cardelli [CW85] e Gries [GG77] recomendam somente conversões explícitas, feitas pelo programador através de funções que tomam um parâmetro de um tipo e retorna este convertido para outro tipo. Mesmo assim, pode haver ambigüidade:

[6] *overloading*

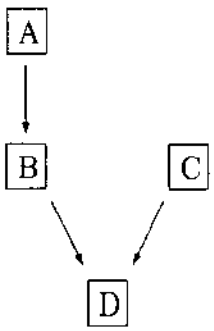


Figura 2.14: Herança causando violação do encapsulamento

```
/* funcao que converte inteiro para complexo */
complex convertfrominteger(int i) { ... }
/* converte inteiro para objeto da classe A */
A convertfrominteger(int i) { ... }
...
...  xx + convertfrominteger(i);
```

Se `xx` referenciar um objeto de uma classe que possui operação + tanto com complexos e como com classe `A`, então há uma ambigüidade. O compilador não pode decidir qual das funções `convertfrominteger` utilizar. Exemplo semelhante a este é dado em [GG77].

2.5 Herança e Encapsulamento de Dados

Em muitas linguagens orientadas a objeto, o uso de herança compromete o encapsulamento de dados [Sny86] [Sny87][7]. Considere a hierarquia de classes da figura 2.2 (a), representando uma classe `D` que herda de classes `B` e `C`. Estas últimas herdam de classe `A`, que define variável de instância `kk` e os métodos denominados `M` e `Q`. A classe `C`, por sua vez, redefine o método `M` herdado de `A`. Nas figuras 2.2 (b), 2.3 (a) e 2.3 (b) estão representados objetos da classe `D`. Cada retângulo envolvendo o nome de uma classe simboliza as variáveis de instância e métodos daquela classe. Assim, um retângulo com `A` em seu interior significa uma variável de instância `kk` com métodos `M` e `Q`. Vejamos o que acontece nas três possíveis interpretações da hierarquia de classes (Cf. pág 7):

- `Linearização`. As classes seriam linearizadas resultando numa lista como a da figura 2.2 (b). A busca por um método, após a recepção de uma mensagem, começa em `D` e continua por `C`, `B` e `A`, nesta ordem.

[7] Esta seção é baseada principalmente nestes dois artigos. Este último é uma versão melhorada do primeiro

A ordem em que D herda[8] B e C é significativa não só para D como também para B ou C. Se classe D herda C antes de B, referências a método M em B referem-se a M de A e não C::M. De acordo com a figura 2.2 (b), qualquer referência a método M em B refere-se ao método definido em C, não ao método de sua superclasse A. A introdução de C nesta hierarquia modificou o comportamento de B. O projetista de D, que herda B e C, deve saber que ambas herdam de uma superclasse comum. Mas as variáveis de instância e os ancestrais de uma classe são informações que podem ser modificadas e devem ser do conhecimento apenas da classe. Assim, linearização viola encapsulamento. Flavour unifica variáveis de instância de mesmo nome - novas variáveis não podem ser introduzidas em uma classe porque subclasses são afetadas.

- Grafo. As variáveis de instância de A não são duplicadas em objeto da classe D - figura 2.3 (a). Cada objeto de D conterá um único conjunto de variáveis de A. Suponha que C não mais herde de A, resultando em objetos iguais aos da figura 2.14. C definirá as operações que herdava de A. Extended Smalltalk e Modular Smalltalk [WBW88] consideram um erro uma classe herdar uma operação por dois caminhos diferentes se a operação não provem de uma mesma superclasse. Na figura 2.3 (a), D herdava Q de B e C, mas ambos provinham de A. Na figura 2.14, D herda de B e C e os métodos Q são diferentes. Nesta última hierarquia, haveria um erro em Modular Smalltalk. Assim, uma alteração na herança de uma classe (como acréscimo ou retirada de superclasses) afeta um descendente, tornando as superclasses de uma classe parte de sua interface.

- Árvore. As variáveis de instância de A são duplicadas em objetos de D - figura 2.3 (b). Esta é a solução adotada por Snyder [Sny87] na linguagem CommomObjects, pois não viola o encapsulamento.

Algumas linguagens, como Smalltalk, permitem que um objeto manipule as variáveis de instância definidas em superclasses da sua classe, quebrando o encapsulamento. Acréscimo de variáveis em uma classe pode causar erros em descendentes, pois a linguagem não permite que uma variável de instância de uma classe tenha nome igual a variável de uma de suas superclasses. Snyder sugere que, se um descendente necessita manipular uma variável de um ancestral, o programador deve negociar com o projetista deste ancestral para que ele forneça as operações necessárias. É interessante comparar Smalltalk com Self. Nesta última linguagem, cada variável de instância é manipulada por dois métodos: uma para recuperar (ler) e outro para modificar (escrever) o valor da variável. Assim, mudanças na representação de uma variável de instância afetam somente dois dos métodos de uma classe.

Na hierarquia da figura 2.2 (a), B é uma subclasse de A. Em Modula-3, subclasse é também subtipo e um subtipo só é possível através da criação de uma subclasse. Um objeto da classe B pode ser utilizado onde objeto de A é esperado. O código abaixo é válido:

```
x : A;
b : B;
...
x:= b;
```

[8] A lista das superclasses de D possui uma ordem. Neste caso, D herda B antes de C.

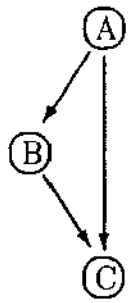


Figura 2.15: Hierarquia não suportada por C++

```
class A { ... };
class B : virtual public A { ... };
class C : virtual public A { ... };
class D : virtual public B,
          virtual public C { ... };
```

Figura 2.16: Classe A é uma classe virtual

Suponha que B seja modificado para não herdar de A, mas mantendo sua interface. O código acima torna-se inválido. Então, a associação de subtipo a subclasse viola o encapsulamento, no caso em que não há outra forma de criação de subtipos. Outras linguagens, como POOL-I[AvdL90], consideram X subtipo de Y se X possui, pelo menos, todos os métodos declarados na interface de Y. Todas as variáveis de instância são privadas à classe em questão. Todas as variáveis de instância são privadas à classe. Snyder sugere que a relação de subtipo seja especificada pelo programador, com *default* que cada classe seja subtipo de suas superclasses. Observe que este problema não acontece em linguagens sem declaração de tipos como Smalltalk ou Flavors, pois estas são altamente polimórficas, com busca de métodos em tempo de execução.

A hierarquia da figura 2.15 é invalida em C++, se todas as heranças são públicas. Acontece erro de compilação. A classe C não pode herdar A por dois caminhos distintos, sendo uma delas herança direta. Neste caso, C herda publicamente B e A. E B herda publicamente A. O fato de que B herda de A é, então, tornado público.

2.6 Herança e Variáveis de Instância

Como foi visto na seção anterior, um objeto de D da figura 2.2 (a) pode conter um ou dois exemplares de cada variável de instância de A. A semântica da classe pode exigir uma opção ou outra.

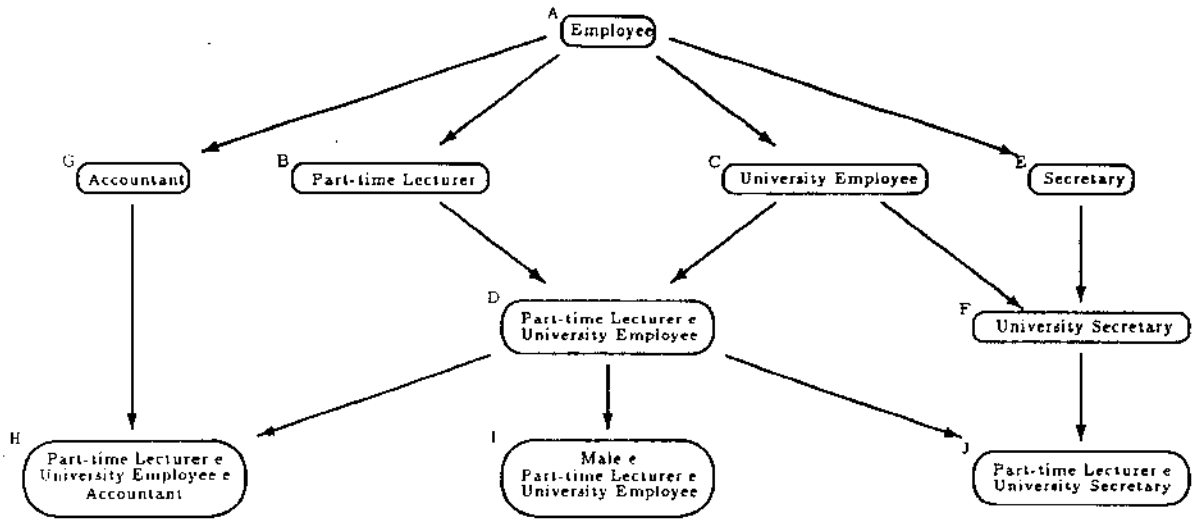


Figura 2.17: Hierarquia de empregos

Uma solução razoável e simples para este problema seria que o projetista da classe `A` especifique quais as variáveis que devem ser duplicadas. Assim, a sua semântica estaria definida com mais precisão. Knudsen [Knu88] apresenta a hierarquia da figura 2.17, onde a classe `Employee` possui as variáveis de instância `Name`, `Adress` e `Seniority` (tempo de serviço). As subclasses de `Employee` são funções, como `Secretary` e `Part-time Lecturer`. Algumas subclasses herdam `Employee` por mais de um caminho, como uma classe que chamaremos de `X`. A classe `X` deve possuir vários ou apenas um atributo `Seniority` ? `Name` e `Adress` certamente são únicos, pois se trata de uma só pessoa. Se `X` herda de duas subclasses de `Employee` que denotam duas funções diferentes, atributo `Seniority` deve ser duplicado – um tempo de serviço para cada função. As classes que `X` herda também podem estar representando uma única função. Neste caso, há um só tempo de serviço e um só atributo `Seniority` – classe `University Secretary` da figura 2.17. Knudsen sugere, então, dois diferentes modos de especialização:

Unificação O número de vezes que o atributo aparece na classe é a soma das vezes que ele aparece nas suas superclasses. Na figura 2.17, há 3 atributos `Seniority` em H: um herdado de G e dois de D. H representa três funções diferentes.

Intersecção O número de vezes que o atributo aparece na classe é menor do que a soma das vezes em que ele aparece nas superclasses. Duas das superclasses, pelo menos, admitem o mesmo significado para o atributo. No exemplo, duas superclasses estariam representando uma mesma função, como `University Secretary`.

Se a variável de instância `Seniority` é privada, não se pode decidir pela sua duplicação sem violação do encapsulamento.

Em Eiffel, há duas possibilidades quando uma variável de instância (digamos, xx) pública de uma classe (neste caso, A) é herdada por mais de um caminho:

- Se variável xx não for renomeada (página 5) em B e/ou C, um objeto de D conterá uma única variável xx. Logo, variáveis privadas de A são sempre unificadas.
- Se for renomeada, um objeto da classe D conterá duas variáveis xx da classe A.

Em C++, o programador pode escolher entre duplicar ou não **todas** as variáveis de A. Se a herança for virtual (fig. 2.16), um objeto de D conterá apenas um exemplar das variáveis de A.

Em Eiffel, se duas variáveis como nomes diferentes são herdadas de duas classes, elas não podem ser unificadas. Isto é, não pode haver nenhum compartilhamento – sempre ocuparão posições de memória diferentes.

2.7 Classificação e Variações de Linguagens Orientadas a Objeto

Linguagens com objetos são classificadas da seguinte forma, em [Weg87a] [CW85]:

Baseadas em Objetos. Suportam objetos, mas não classes ou herança. Ex: Ada [Ada81] e Modula-2 [Wir83]. Um *package* de Ada é um objeto, pois possui dados e operações próprios. Um *package* não é uma classe porque não define um tipo (não podem ser criados objetos a partir dele). Cada *package* define um único objeto. Linguagens Actors [Agh86] sem delegação enquadram-se nesta categoria. Elas possuem objetos e abstração de dados, mas não classes.

Baseadas em Classes. Suportam objetos e classes, mas não herança. Ex: CLU, Emerald [RTL+91].

Linguagens Orientadas a Objeto. Suportam objetos, classes e herança ou mecanismos similares. Ex: Smalltalk, Self [US87].

Pode-se considerar um módulo de Modula-2 (ou Ada) com um único tipo opaco[9] como uma classe. Uma variável declarada com tipo que é opaco armazena um objeto em execução.

Objetos são entidades de primeira classe quando eles podem ser valores de variáveis, passados como parâmetro e ser membros de estruturas (registros). Em resumo, quando objetos possuem os mesmos privilégios que valores dos tipos básicos (integer, boolean, ...). Objetos de Ada e Modula-2 (*packages* e módulos) não são de primeira classe, ao contrário dos objetos de linguagens Actors.

A tabela 2.18 mostra a presença de algumas linguagens e suas características. O significado das abreviações da primeira linha é dado abaixo:

[9] Tipo exportado por um módulo mas que só pode ser manipulado dentro do módulo

[10] Consideramos que existe uma falha no sistema de tipos desta linguagem.

[11] Um ponteiro para objeto da subclasse B de A pode ser usado onde objeto de A é esperado.

[12] Permite aninhamento, mas a classe B encaixada dentro de A tem o mesmo escopo que A.

[13] Ponteiros para void e as facilidades de conversão entre tipos impedem que C++ seja fortemente tipada.

[14] BETA possui uma construção, a classe virtual, que simula classes parametrizadas.

[15] Possui exceções descritas na versão corrente e não na descrita no manual de referência [DdS87].

[16] Não é fortemente tipada porque possui ponteiros para void e facilidades de conversão de tipos

Linguagem	HS	HM	SS	E	AC	M	FT	DT	CP	ME	RE
Smalltalk	S										
Eiffel	S	S		S			N[10]	S	S		S
C++	S	S	S[11]	S	N[12]		N[13]	S	S	S	
Trellis	S	S		S			S	S	S		
Flavor	S	S									
POOL-I	S	S					S	S	S		S
Modula-3	S		S	S		S	S	S			
CLU				S			S	S	S	S	
Simula	S		S		S			S		S	
BETA	S		S		S			S	S[14]	S	
CM	S	S	S	S[15]		S	N[16]	S	S		S

Figura 2.18: Comparação entre diversas linguagens. Os campos em branco significam que a linguagem não suporta a característica.

- HS. Herança simples.
- HM. Herança múltipla.
- SS. Todo subtipo é também subclasse.
- E. A linguagem possui tratamento de exceções.
- AC. Aninhamento de classes. Uma classe pode ser declarada dentro de outra.
- M. Módulos.
- FT. Fortemente tipadas. Uma linguagem é fortemente tipada se erros de tipo são impossíveis de acontecer em execução.
- DT. Com especificação de tipos na declaração de variáveis. Algumas linguagens admitem a declaração de variáveis sem especificar de que tipo elas são.
- CP. Classes parametrizadas (Cf. capítulo 4).
- ME. Possui opção de fazer ligação estática de métodos – possui métodos não virtuais, além dos virtuais.
- RE. Pode-se renomear métodos herdados. Veja definição de renomear na página 2.

Nenhuma das linguagens da tabela é estaticamente tipada[18], nem mesmo CLU. Nesta linguagem, a maior parte do código de uma classe parametrizada é compartilhado por diversas instanciações. Neste caso, alguns tipos não são conhecidos em tempo de compilação.

Algumas características não presentes na tabela estão relacionadas abaixo:

[18] Uma linguagem é estaticamente tipada se os tipos de todas as variáveis são conhecidas em compilação

- Exceções que uma rotina (método) pode sinalizar fazem parte do seu tipo. Para que duas rotinas sejam do mesmo tipo é necessário que as exceções que elas podem sinalizar sejam as mesmas.
- Classes abstratas (seção 2.8).
- Classes mixin (seção 2.10.2).
- Linearização do grafo da hierarquia de herança.
- Número de níveis de proteção de informação (público, protegido, privado - página 10).
- Exigência de verificação de tipo em execução. O compilador gera código que confere, em tempo de execução, se as atribuições de variáveis são consistentes. Uma atribuição é inconsistente é associar a uma variável da classe B um objeto da classe C que não é relacionada com B.
- Permissão para atribuição de variável de subtipo com variável de tipo. É possível a atribuição `b = a` se `b` é da classe `B`, subclasse da classe `A` de `a` ? Veja seção 2.2.
- Exigência de recompilação dos clientes em caso de alteração de variáveis de instância privadas de uma classe.
- Orientação a objetos pura - a única forma de abstração são classes, não sendo permitidos procedimentos ou funções.
- Tipos básicos, como inteiros, são objetos.
- Possui alguma forma de escolha quanto à duplicação/compartilhamento das variáveis de instância de uma classe herdada por mais de um caminho - seção 2.6.

2.8 Classes Abstratas

Uma classe abstrata é uma classe que especifica, mas não implementa, alguns de seus métodos. Estes métodos são, obrigatoriamente, implementados nas subclasses. Como este tipo de classe é incompleta, não se pode criar objetos a partir delas. Sua única utilidade é ser herdada por outras classes.

Smalltalk não possui nenhuma notação especial para criação de classes abstratas. Os nomes de métodos não definidos não são especificados na interface. Não há erro em compilação, pois a busca pelo método só acontece em execução.

2.9 Eficiência e Geração de Código

O principal motivo da ineficiência de linguagens orientadas a objeto com ligação tardia é o acoplamento mensagem/método. Segundo Takahashi e Liesenberg [TL90], 40% do tempo total de execução dos programas em Smalltalk é gasto na busca do método adequado e sua

invocação. Nesta linguagem, a busca por um método se faz por nome. Quando um objeto recebe uma mensagem, digamos M, procura-se M entre os nomes dos métodos da classe do objeto. Se não for encontrado, procura-se na superclasse e assim por diante. Utiliza-se comparação entre as *strings* do nome da mensagem e do nome dos métodos.

Smalltalk considera tipos básicos (inteiros, lógicos, ...) como objetos. Há perda de eficiência porque aumenta o número de mensagens e não são utilizadas diretamente instruções do *hardware* disponíveis para estes tipos. Por exemplo, considere a instrução de incremento de uma variável inteira `i`. Em Smalltalk, ela seria executada através do envio da mensagem + a i seguido da atribuição ($\leftarrow$) a esta variável. A instrução é $i \leftarrow i + 1$. Considerando a linguagem Pascal (ou C) e o microprocessador 8088, seria utilizado uma única instrução de máquina[19] para este incremento (`i:= i + 1` ou `i++`).

Segundo Wirfs-Brock e Wilkerson [WBW88], o motivo da ineficiência de Smalltalk não é a busca pelo método, mas a impossibilidade de fazer qualquer otimização porque a linguagem é reflexiva. Isto significa que o programa pode modificar-se a si mesmo, inclusive alterando classes e métodos.

O código de um programa construído com orientação a objetos é maior do que se ele fosse feito de maneira convencional. O motivo é que toda a hierarquia é incluída, mesmo quando não é utilizada. Este é um dos motivos porque herança não foi incluída na linguagem Emerald [RTL+91].

Programação orientada a objetos utiliza um número maior de chamadas de procedimentos (envio de mensagens) que programação procedimental. Os motivos são os seguintes:

- As rotinas são menores.
- Incapacidade de manipular variáveis diretamente, por causa da proteção de informação. Em linguagens como Smalltalk e Trellis/Owl, os parâmetros de um método só podem ser manipulados via mensagens, mesmo que um parâmetro seja da mesma classe que o objeto que recebeu a mensagem. O número de mensagens nestas linguagens é muito maior que em outras, como C++. Alguns acessos a variáveis nesta última convertem-se em envio de mensagens nas primeiras.
- Segundo Liskov [LSRS77], o uso de abstração de dados tende a introduzir níveis de invocação de procedimentos que fazem pouco ou nada de computação.

C++ é eficiente porque possui métodos não virtuais, com ligação estática em tempo de compilação. O conhecimento do programador de que uma certa invocação de método pode ser feita em tempo de ligação não é desperdiçada. Trellis/Owl é uma linguagem fortemente tipada que não vincula subtipo a subclasse, conseguindo um alto grau de polimorfismo. Como no caso de funções virtuais, alguns métodos não necessitam ser polimórficos. Poderiam ter seus parâmetros fixados a classes em compilação. A otimização do código seria facilitada. Segundo Andraws e Harris [AH], até 90% das ligações de métodos pode ser feita estaticamente, em sua linguagem VBASE. Outro fator importante na eficiência é a possibilidade de um método invocar outro da mesma classe com um qualificador. Considere uma classe A com métodos M e N e o seguinte código em N:

[19] Em assembler, "inc i"

A::M();
A ligação será estática, em C++, mesmo se M for método virtual.

A linguagem DSM [SHBR89] possui três tipos de ligação mensagem/método

- Estática - o método a ser utilizado é definido em tempo de compilação.
- Dinâmica, por acesso a ponteiro para função. O método adequado é aquele para o qual o ponteiro para função se refere. O acesso ao ponteiro para função substitui a busca pelo método correspondente à mensagem.
- Dinâmica, por busca por nome. Procura-se, na hierarquia de classes, por método com mesmo nome que a mensagem. A *string* com o nome da mensagem é comparada com *strings* com os nomes dos métodos. Semelhante à busca em Smalltalk.

Esta linguagem permite a criação de classes e acréscimo de métodos a classes em execução. Este é o motivo da existência do terceiro método de busca. Um método pode ser declarado "folha", garantindo que ele não será sobreposto numa subclasse. Assim, ligação estática é usada para este método. Em C++, pode-se fazer um método virtual a partir de um determinado ponto na hierarquia, mesmo que acima deste ponto ele não o seja - o oposto de DSM.

Lee, Ungar e Chambers [CUL89] sugerem uma métrica para medir a performance de linguagens orientadas a objeto e seus compiladores:
MiMS - milhões de mensagens por segundo.
As mensagens contabilizadas devem ser as semanticamente enviadas, mesmo que isto não ocorra no durante a execução por causa de ligações estáticas ou otimizações do compilador.

Imagine um processo com 100 objetos de 10 bytes cada um e cada objeto em uma página física da memória diferente. Se cada página armazena 4 Kbytes e não há outros dados no sistema, há um desperdício de $4096*100-10*100$ bytes. Este problema motivou a criação de sistemas de memória virtual por software [Kae81]. Objetos são carregados e descarregados do disco da mesma forma que são páginas pelo sistema operacional. Fazer *swap* de objetos reduz a memória principal utilizada, mas é mais complexo por dois motivos [Kae81]:

- Não há suporte de *hardware*.
- Eles são de tamanhos diferentes, ao contrário de páginas. A otimização possível é agrupar objetos de classes diferentes em páginas diferentes do disco. Isto aumenta a localidade das referências, pois é provável que, após um objeto de uma classe ser manipulado, outros da mesma classe também o sejam. É importante notar que, se um programa utiliza algumas partes de um objeto, ele provavelmente utilizará outras partes também. Informações foram agrupadas em um objeto por terem mais em comum entre si do que com os outros dados.

```
class A { ...
  void print();
  M() {
    printf("1");
    inner;
    printf("3");
    };
  };
class B : A {
  void print();
  M() {
    printf("2");
    }
  };
  ...
  B  b;  // declara b como da classe B
  b.M();
```

Figura 2.19: Classes com semântica de Beta

2.10 Outros Modelos de Herança

2.10.1 Herança em BETA

BETA [KOLM87] possui apenas herança simples e permite apenas a extensão de métodos de uma classe em suas subclasses. Suponha que classe `B` herde da classe `A` e ambas definam método `M`. Figura 2.19 mostra classes `A` e `B` com sintaxe de C++ e semântica de BETA. Quando objeto de `B` recebe mensagem `M`, método `M` de `A` é invocado. Este método invoca `M::B` com o comando `inner`. A pseudo-variável `super` em Smalltalk é utilizada para se referir à superclasse. e `inner` em BETA, ao método de mesmo nome da subclasse. A instrução `b.M()` da figura 2.19 imprimirá `123`.

O método original da superclasse é sempre invocado, para preservar a semântica do método. Este mecanismo é uma tentativa de fazer com que cada subclasse seja um subtipo. Suponha-se que o método `print` de `A` da figura 2.19 possua o significado "imprima os valores associados à classe". Se esta classe é herdada por `B`, é exigência semântica que `print` de `A` seja executado quando objeto de `B` recebe mensagem `print`. Observe que BETA admite que todos os métodos possuem um significado que é distribuído ao longo da hierarquia de uma classe.

A semântica da herança de BETA é simulada imperfeitamente em outras linguagens quando um método `M` invoca o método de mesmo nome da superclasse. Mas uma superclasse não pode requerer que um método virtual seu seja invocado em redefinições deste em sub-

```
// classe normal D
class D {  ... void print(); }
// Esta 'e uma  classe mixin
class mixin B_m {
  ...
  void print() {
    super.print();
    }
  };
// Esta 'e uma  classe mixin
class mixin C_m {
  ...
  void print() {
    super.print();
    }
  };
// Esta 'e uma classe normal
class A : B_m, C_m, D {
  ...
  void print() {
    super.print();
    }
  };
```

Figura 2.20: Herança Mixin de C_m e B_m

classes. Poder-se-ia utilizar restrições [RID90]. Deve então existir uma linguagem apenas para especificar este tipo de relacionamentos entre classes. Na interface de uma classe `A` estaria especificado que redefinições de um método `M` em subclasses devem invocar método `M` da classe `A`. BETA limita o uso da herança, direcionando-a para extensões de classes.

2.10.2 Herança Mixin

Um mixin [BC90] [Weg87b] é uma classe que não herda de nenhuma outra mas cujos métodos podem enviar mensagens para a pseudo-variável `super`[20]. Uma classe mixin só pode ser herdada, nunca utilizada diretamente. Se é criado um objeto a partir de um mixin, ocorre um erro de execução quando uma mensagem é enviada para `super` em algum método do mixin (este mixin não possui superclasses).

Quando uma classe mixin é herdada, a ela é associada uma única superclasse. Referência a `super` em métodos do mixin referem-se a esta superclasse. Como exemplo considere as classes `A`, `B_m`, `C_m` e `D`. As classes `B_m` e `C_m` são mixins e as classes `A` e `D` são normais. A classe `D` não herda de ninguém, enquanto que a classe `A` herda de `B_m`, `C_m` e `D`, nesta ordem.

[20] Exemplo de envio de mensagem: `super.print()`

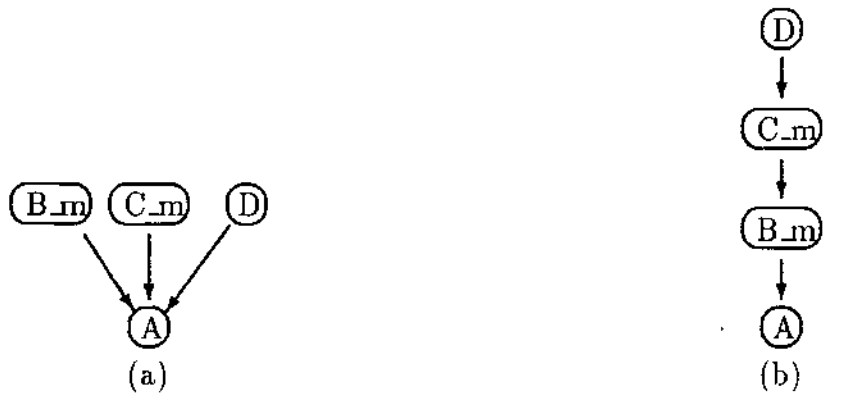


Figura 2.21: (a) Herança mixin de `B_m` e `C_m` (b) Linearização da hierarquia

```
class mixin B_min
  uses void super.print(),
       void super.imprime(char *s)
  { ... };
```

Figura 2.22: Mixin com tipos

A classe `D` é considerada superclasse de `C_m` em objetos de `A`, assim como `C_m` o é de `B_m`. Classes mixins são o oposto de classes abstratas, onde a classe faz referências a métodos implementados na subclasse.

Herança mixin exige linearização da hierarquia de classes. A superclasse, não citada na definição do mixin, é considerada como sendo a classe que se segue a ela na hierarquia linearizada. Figuras 2.20 e 2.21 (a) mostram mixins `B_m` e `C_m`, utilizadas na declaração da classe `A`. A sintaxe é a de C++. Considera-se que uma classe (classe `A`) possa herdar de vários mixins (`B_m`, `C_m`), mas apenas de uma classe não mixin (classe `D`). A hierarquia linearizada (figura 2.21 (b)) é a seguinte:

```
A  B_m  C_m  D
```

`super` em `B_m` refere-se a `C_m`, `super` em `C_m` é associado a `D`. A seguinte seqüência de ações acontece quando um objeto da classe `A` invoca o método `print`:

- Instrução `super.print()` em `A::print` invoca `B_m::print`.
- Instrução `super.print()` em `B_m::print` invoca `C_m::print`.
- Instrução `super.print()` em `C_m::print` invoca `D::print`.

Observe que método `print` da superclasse é utilizado em `B_m` e `C_m` sem o conhecimento da quantidade e tipos dos parâmetros deste método. Uma linguagem fortemente tipada exigiria a declaração de todos os métodos utilizados pelo mixin - veja figura 2.22.

```
(defmethod print ( (self B_m))
  (call-next-method)
  (...)
  )
```

Figura 2.23: Mixin em CLOS

```
class mixin B_m[ X, Y]
  uses  X::print(),
        Y::print()
  { ...
    void print() {
      X::print();
      Y::print();
      };
    };
class A : B_m[E, F],
          C_m, E, F {
{  ...
  void print() { B_m::print();}
};
```

Figura 2.24: Mixins com herança múltipla

CLOS suporta mixins, mas não há uma construção na linguagem especialmente para esta finalidade. O método da superclasse com nome igual ao do método corrente é invocado com a instrução `call-next-method`. Figura 2.23 mostra método `print` da classe `B_m` da figura 2.20 implementado em CLOS. A instrução `call-next-method` desempenha a mesma função que `super.print()` de `B_m::print`. A instrução `call-next-method` não causa erro em tempo de compilação, pois CLOS não é fortemente tipada. Esta instrução só é avaliada em execução, quando a classe deve ter uma superclasse.

Outra linguagem que suporta mixins é DSM [SHBR89]. Cada classe pode ter uma superclasse principal não mixin e várias superclasses mixins. DSM é fortemente tipada. Brack e Cook propõem em [BC90] uma extensão a Modula-3 para suportar mixins. Modula-3 suporta herança simples e é, essencialmente, fortemente tipada. Um mixin especifica os métodos da superclasse utilizados por ele, semelhante à classe `B_m` da figura 2.22.

Mixins só foram implementados em linguagens sem herança múltipla ou com linearização da hierarquia de classes. A parametrização de classes permite-nos sugerir um modelo com mixins para linguagens com herança múltipla e sem linearização. Neste caso, um mixin poderia possuir parâmetros que representam as suas superclasses. A figura 2.24 apresenta uma classe `A` que herda de `B_m`, `C_m`, `E` e `F`. `B_m` é mixin, com parâmetros `X` e `Y`. O método

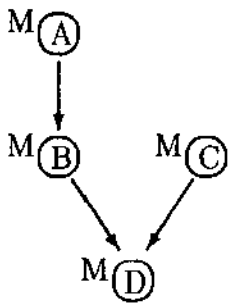


Figura 2.25: Todos os métodos de `M` executam concorrentemente em MELD

`print` de `B_m` invoca os métodos `print` de `X` e `Y`. Na cláusula `uses` está especificado que as classes `X` e `Y` possuem método `print`. `X` e `Y` são associados a `E` e `F`, na classe `A`. Invocação de `A::print` causa a execução de `B_m::print`, `E::print` e `F::print`.

2.10.3 Herança e Data-Flow

Kaiser e Garlan [KG87] apresentam uma linguagem, MELD, que combina herança e o modelo Data-Flow [DK82] [Ack82]. Suponha que métodos de nome `M` sejam definidos em uma classe e todos os seus ancestrais. Quando um objeto desta classe receber a mensagem `M`, todos os métodos da hierarquia com este nome começam a executar concorrentemente – figura 2.25. A única restrição de execução é a disponibilidade de dados, como em computadores Data-Flow. MELD permite que classes manipulem variáveis privadas das suas superclasses. Portanto, há duas formas de dependência de dados entre os métodos `M` em uma hierarquia:

- Manipulação de variáveis de uma superclasse comum.
- Invocação de métodos que manipulam variáveis de uma superclasse comum.

Segundo Kaiser e Garlan, o objetivo do entrelaçamento de métodos é aumentar a reutilização de software. Não há dados ou experiências comprovando que orientação a objetos se torne mais flexível com este modelo. A interdependência entre métodos sem relacionamento entre si provavelmente é uma causa de erros sutis, pois existe a disputa por dados por classes feitas independentemente.

2.10.4 Herança Baseada em Regras

Nenhum esquema de herança é adequado para todas as situações. Diferentes hierarquias implicam em diferentes significados e, conseqüentemente, necessitam diferentes formas de herança. A linguagem Sina/St [ATS8] e o modelo baseado em regras apresentado em [MR87] têm a pretenção de permitir ao usuário a manipulação do significado da herança. Sina/St não possui herança como construção da linguagem, mas ela é facilmente simulada. Uma forma muito limitada de delegação, se podemos considerar assim, também existe.

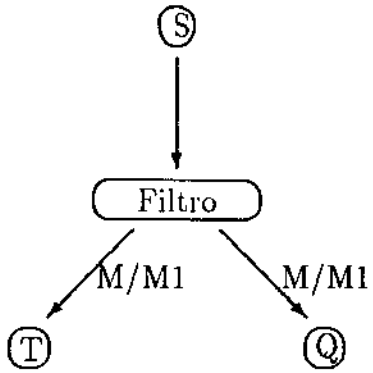


Figura 2.26: Opções de resolução de uma mensagem. Filtro são as cláusulas em Prolog.

O modelo de orientação a objetos baseado em regras admite que existe um filtro entre cada mensagem e o objeto para o qual ela é enviada. Considere que a mensagem `M` é enviada pelo objeto `S` (*source*) para objeto `T` (*target*). Escreve-se <S,M,T>, segundo a notação de Minsky e Rozenshtein [MR87]. Cada vez que uma mensagem é enviada, um interpretador Prolog deve resolver o *goal*

```
send(S,M,T)
```

de acordo com regras definidas nesta linguagem. Estas regras podem:

- enviar mensagem M para T, como se não houvessem regras.
- enviar uma outra mensagem M1 para T ou M para outro objeto Q ou ambas as coisas – veja figura 2.26
- Não enviar mensagem alguma.

Estas regras governam todas as mensagens de um programa. Sem alterar o código, pode-se modificar a semântica de um sistema via alteração das regras em Prolog. Estas regras podem mesmo modificar a si mesmas. O modelo, como apresentado em [MR87], não admite abstração de dados ou herança – ambas são implementadas por meio de regras. Diversos esquemas de herança podem coexistir em um mesmo programa, moldando melhor a sua semântica.

Este modelo é bastante poderoso, mas diminui consideravelmente a legibilidade de um programa. O significado de cada herança e de cada troca de mensagens é definido em código Prolog, que pode até modificar-se a si mesmo. Cada hierarquia de classes deve ser compreendida separadamente das outras, através do estudo das cláusulas que a regem.

2.10.5 Ponto de Vista

Figura 2.27 mostra uma hierarquia de classes (extraída de [CG90]), definida pelo código da figura 2.28. `BoundedPoint` define pontos com coordenadas limitadas. `HistoryPoint`

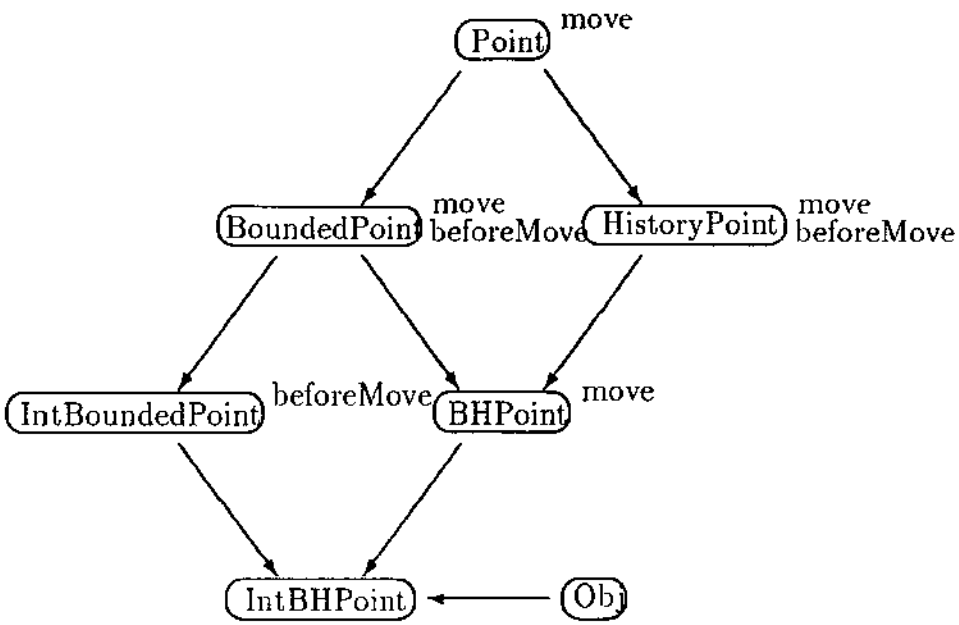


Figura 2.27: Representação gráfica de uma hierarquia de pontos

armazena as últimas localizações do ponto. Em `IntBoundedPoint`, as coordenadas são números inteiros e limitados. O gerenciamento da história e limites é feito nos métodos `beforeMove`, que geralmente são invocados pelos métodos `move`. Assim, método `move` de `BoundedPoint` envia mensagem `beforeMove` para `self` (invoca `beforeMove`) antes de mover realmente o ponto. `BoundedPoint::beforeMove` confere se o ponto estará dentro dos limites permitidos. Em `BHPoint` (que herda de `BoundedPoint` e `HistoryPoint`), `move` invoca os métodos `beforeMove` de suas superclasses para atualizar a história e conferir os limites. `IntBHPoint` herda de `IntBoundedPoint` (que redefine métodos `beforeMove`) e de `BHPoint` (que redefine método `move`). Logo, `IntBHPoint` herda `beforeMove` de `IntBoundedPoint` e `move` de `BHPoint`. Mas `BHPoint::move` não invoca `IntBoundedPoint::beforeMove`. Assim, o método `move` de `IntBHPoint` não confere se as novas coordenadas do ponto são inteiras ou não. Uma solução para este problema é redefinir `IntBHPoint::move` para que ele invoque o método `IntBoundedPoint::beforeMove`. Se houver esta redefinição, estaremos considerando que `move` invoca `beforeMove`. Isto é verdade nesta implementação específica, mas pode ser modificado pelos projetistas de `BHPoint`. `BoundedPoint` e `HistoryPoint`[21]. Deste modo, redefinição de `move` em `IntBHPoint` exige conhecimento de fatores internos ao método `BHPoint::move`. Se não há redefinição , a semântica estará errada, pois `beforeMove` de `IntBoundedPoint` não será invocado quando `IntBHPoint` receber mensagem `move`. Carré e Geib [CG90] sugerem a noção de "Ponto de Vista" para resolver este problema.

Ponto de vista de uma classe `A` em relação a um objeto `Obj` é a interseção entre `R(A)` e `S(Obj)`. `R(A)` é o conjunto de todas as superclasses e subclasses de `A`. `S(Obj)` é o conjunto das superclasses mais a classe de `Obj`. A busca por um método, quando uma mensagem é

[21] Este exemplo apresenta o sugestivo nome de `beforeMove`, que indica uma ação a ser executada antes de `move`. Em outras situações, a relação de dependência entre dois métodos pode ser menos explícita

```
class  Point
  methods
    move
end -- Point
class BoundedPoint
  methods
    move
    beforeMove
end -- BoundedPoint
class HistoryPoint
  methods
    move
    beforeMove
end -- HistoryPoint
class IntBoundedPoint
  methods
    beforeMove
end -- IntBoundedPoint
class BHPoint
  methods
    move
end -- BHPoint
class IntBHPoint
end -- IntBHPoint
```

Figura 2.28: Hierarquia de Pontos

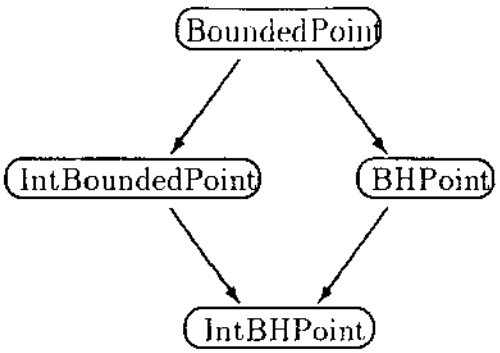


Figura 2.29: Ponto de vista de `BoundedPoint` em rela,c ao a `Obj`

enviada para `self`, inicia-se na classe mais específica da hierarquia definida por seu ponto de vista (PV) e envolve apenas as classes de seu PV.

Um exemplo: a mensagem `move` é enviada para objeto `obj` de `IntBHPoint` (veja figura 2.27). O método executado é `BHPoint::move`, que envia mensagem `beforeMove` para `self`. A busca pelo método `beforeMove` a ser executado é feita na hierarquia definida pelo ponto de vista, que é calculado da seguinte forma:

```
R(BoundedPoint) = { Point, BHPoint, IntBoundedPoint, IntBHPoint}
S(Obj) 'e toda a hierarquia
PV(BoundedPoint) = { Point, BHPoint, IntBoundedPoint, IntBHPoint}
```

Figura 2.29 mostra o ponto de vista de `BoundedPoint` em relação à hierarquia da figura 2.27. Logo, a busca inicia-se em `IntBHPoint` e encontra o método `beforeMove` a ser invocado em resposta à mensagem em `IntBoundedPoint`.

Capítulo 3

Alguns Problemas com Orientação a Objetos

A seguir são apresentados alguns problemas com orientação a objetos.

3.0.6 Problema 1

Considere a rotina P e a classe ArvBin, em C++:

(a) P é definido como

```
int P(int x);
```

A documentação de P descreve a relação entre a sua entrada a sua saída.

(b) A classe ArvBin (árvore binária) com métodos

- Insere - insere elementos na árvore.
- Busca - procura por um elemento na árvore.

Subclasse xArvBin de ArvBin redefine o método Busca. Esta classe será usada para armazenar árvores binárias especiais, em que a árvore assume formas mais restritas (um exemplo é uma com todos os ponteiros da esquerda nulos - uma lista linear). O método Insere deve também ser redefinido? Se ele usa Busca, não é necessário. Se ele não usa, deve ser redefinido.

Surge então um problema: a semântica com que Insere usa outros métodos de ArvBin (em mensagens para *self*) é parte de sua interface. Este fato deve ser do conhecimento do programador. Se o código do método Insere é reprojetado (construído novamente em ArvBin), o programador deve considerar a relação entre a sua entrada e a sua saída (como na rotina P) e a semântica de utilização de outros métodos da mesma classe (em mensagens para *self*). De outra forma, há uma violação do encapsulamento.

Imagine Insere de ArvBin utilize inicialmente método Busca. A subclasse xArvBin é construída, redefinindo apenas Busca - o programador sabe, pela documentação, que Insere

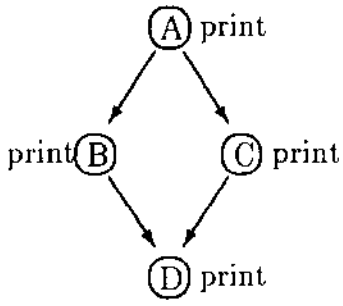


Figura 3.1: Hierarquia com método `print`

de ArvBin utiliza este método[1]. ArvBin é alterada pela substituição do código de Insere por outro que não utiliza Busca. A classe xArvBin é afetada.

3.0.7 Problema 2

Todas as classes da figura 3.1 definem método `print`. A sua semântica é "imprima os dados do objeto". `print` é definido em cada subclasse de `A` obedecendo a esta semântica Cada `print` imprime os dados particulares de cada classe e invoca cada um dos métodos `print` de suas superclasses. A definição de `D::print` pode ser:

```
// C++
void D::print()
{
  // Aqui esta a impressao dos dados de D
  B::print();
  C::print();
  }
```

A semântica das entidades representadas nas classes exige que objetos de `D` contenham um único conjunto de variáveis de `A`.

Quando um objeto de `D` recebe uma mensagem `print`, ele invoca `print` de `B` e `C`. Cada um destes invoca `print` de `A`. Logo, a saída conterá os dados de `A` duas vezes, com valores idênticos. Este fato não é evitável se `B` e `C` forem desenvolvidas independentemente.

O problema é que a semântica de `print` não está definida completamente nos métodos das classes. A semântica envolve também a forma do grafo da hierarquia de classes. Observe que o fato de todos os métodos se chamarem `print` não é a causa do problema.

3.0.8 Problema 3

Considere a figura 2.27 da seção 2.10.5. Faremos alguma modificações:

- Nenhum método `move` ou `beforeMove` invoca métodos de ancestrais.

[1] O que implica em que ele sabe algo sobre as estruturas de dados de ArvBin e sobre o algoritmo utilizado !!

```
void IntBHPoint::rmove()
{
  // visibilidade nao 'e restrita aos ancestrais
  IntBoundedPoint::beforeMove();
  BoundedPoint::beforeMove();
  HistoryPoint::beforeMove();
  BHPoint::move();
  }
```

Figura 3.2: Método rmove completamente definido

```
void Point::rmove()
{
  // Pseudo-Codigo
  Para cada superclasse Sc do objeto corrente faca:
    Sc::beforeMove();  // invoque todos os beforeMove
  move();   // invoque move
  }
```

Figura 3.3: Método rmove genérico, em pseudo-código

- Nenhum método `move` invoca métodos `beforeMove`.

São criados métodos `rmove` (`move` real) em todas as classes da hierarquia, incluindo `IntBHPoint`, que originalmente não possui um método `move`. Se alguém quer mover um ponto, deve chamar `rmove` e não `move`. Cada `rmove` deve chamar todos os métodos `beforeMove` de seus ancestrais. Depois, deve chamar o método `move` de sua classe (herdado ou definido). Figura 3.2 mostra uma definição do método `rmove` de `IntBHPoint`, com sintaxe de C++. Consideramos que a visibilidade não é restrita aos ancestrais.

Com a presença de ferramentas apropriadas, `rmove` pode ser definido apenas uma vez, como combinação de métodos de Flavors ou CLOS. Uma sugestão para `rmove` é dada na figura 3.3. O resultado deste método aplicado a objetos de `IntBHPoint` é idêntico à aplicação do método `rmove` da figura 3.2 sobre os mesmos. Este exemplo ilustra a influência da hierarquia na semântica do relacionamento entre métodos. O significado de `beforeMove` e `move` não é totalmente expresso por eles mesmos.

3.0.9 Conclusão

Estes problemas levam-nos às seguintes conclusões:

- A semântica de um método envolve o seu relacionamento com outros métodos da mesma classe em mensagens para *self*.

- A semântica de um método ou conjunto de métodos depende da forma do grafo da hierarquia das classes.

Estes problemas não possuem soluções simples, talvez nem mesmo soluções.

Capítulo 4

Classes Parametrizadas

4.1 Introdução

Considere que um programa utilize uma classe `Heap`[1] de inteiros, uma de carateres e outra de números reais. O código para a implementação de uma `Heap` seria utilizado 3 vezes no programa, sendo que a única diferença entre eles é no tipo de objetos da `Heap`. Para evitar a repetição de um mesmo código em vários pontos de um programa, existem classes com parâmetros. Os parâmetros são tipos ou constantes. Como exemplo, é apresentado na figura 4.1 a interface (sem a implementação) de uma `Heap` genérica com parâmetro `TipoElemento`. Dois de seus métodos são `Vazia` e `Insere`.

`TipoElemento` é instanciado para um tipo na declaração de um objeto `Heap`. Neste exemplo, `TipoElemento` deve possuir as operações de comparação (<, <=, >, ...), pois estas são utilizadas pelos métodos que `Heap` possui. Algumas linguagens, como Eiffel, permitem que os métodos de uma classe parametrizada utilize apenas as operações triviais sobre os parâmetros, como teste de igualdade, desigualdade e cópia. Em outras, como CLU, os métodos podem invocar qualquer operação sobre os parâmetros. As operações que um parâmetro deve suportar são dadas explicitamente na interface da classe, através da cláusula `where`.

De modo análogo a classes parametrizadas, existem procedimentos, tipos e módulos parametrizados. Os parâmetros de uma classe parametrizada podem ser classes, outros tipos, constantes. Em geral, pode-se declarar variáveis de uma classe parametrizada diretamente, especificando-se os parâmetros na declaração, como foi feito com HpInt, HpChar e HpReal na figura 4.1. Código de `Heap` associado aos tipos `integer`, `char` e `real` serão criados automaticamente. Em Ada, que possui *packages* (módulos) parametrizados, existe uma construção (cláusula) especial para a instanciação (criação) de novos *packages* a partir de um *package* com parâmetros. Um exemplo é dado abaixo, com o *package* genérico (parametrizado) `BinTree`, que possui os parâmetros `T` e `Menor`.

```
-- BinTree 'e um package generico
generic
  type T is private;
```

[1] estrutura em forma de árvore onde cada nó possui um valor que é maior que os valores dos nós filhos

```
class Heap[TipoElemento]
  instance variables:
          { arvore da heap 'e representada no vetor }
     vet : array[1..100] of TipoElemento ;
     ...
  methods :    { Cabecalho dos metodos }
     Vazia : boolean;
     Insere( x : TipoElemento ) : boolean;
     ...
end  class
   ...
   { declaracao de variaveis }
   var  HpInt  : Heap[ integer ];
        HpChar : Heap[ char ];
        HpReal : Heap[ real ];
```

Figura 4.1: Uma Heap parametrizada

```
     with
       function Menor( a, b : T )  return BOOLEAN is <> ;
     ...
package BinTree is
   ...
end BinTree;
...

package Int_BinTree is new BinTree( int, CompareIntMenor) ;
package A_BinTree is new BinTree(A, MenorA);
...
 UmaBinTree : Int_BinTree;
```

A abordagem de Ada permite que os atributos exigidos na interface do *package* (T e Menor) possuam nomes diferentes dos atributos usados na criação do *package* (int e CompareIntMenor para Int_BinTree). No exemplo acima, a função exigida por BinTree chama-se Menor e a dada na criação de A_BinTree é MenorA.

Uma forma de implementação de classes parametrizadas é a substituição textual de todas as ocorrências do parâmetro na classe (com seus métodos) pelo tipo real (integer na declaração de HpInt da figura 4.1). A declaração de HpInt produziria uma nova classe não acessível diretamente ao usuário:

```
   class Heap_Integer
      instance variables:
```

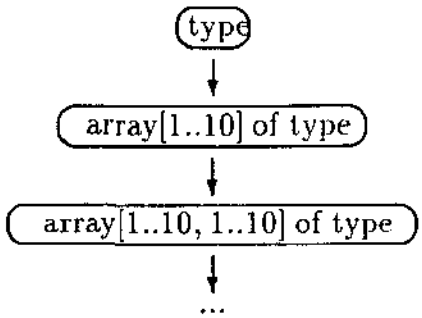


Figura 4.2: Criação de infinitos tipos

```
        vet : array[1..100] of integer;
        ...
    methods:
        Vazia : boolean;
        Insere( x : integer ) : boolean;
        ...
end class
```

Esta forma de implementação pode levar a uma recursão infinita, se o tipo ou procedimento inclui a si mesmo em sua definição [GG77]:

```
{ p 'e procedimento parametrizado. <type> 'e o parametro }
procedure p(a : <type> ; n : integer )
var b : array[1..10] of <type>;
begin
  if n >= 1 and  n <=  10
  then
    p(b, n + 1) ; { aumenta <type> de uma dimensao }
  ...
end { p }
```

Utilizando substituição textual, o número de procedimentos criados é infinito – figura 4.2. Embora em execução seja necessário no máximo 10 procedimentos p.

Segundo Gries [GG77], o problema de determinar se um programa gera um número infinito de tipos é indecidível. Afinal, gerar ou não um tipo pode depender do fluxo de execução, como no exemplo acima. Recursão infinita acontece também se um tipo A utiliza tipo B e este utiliza A:

```
class A[T] {
  typedef VetT T[10];
  B[ VetT ] vetb;
  ...
```

```
class stack( type T = int; int N = 100)
T [N]vet; /* Declara vetor do tipo T */
...
export T pop()  /* m'etodo exportado */
{ ...
  }
...
```

Figura 4.3: Uma classe parametrizada em Cm

```
class A {
  ...
  void M() {
    int aa, ii ;
    ...
    }
  };
```

Figura 4.4: Classe A em sintaxe de C++

```
  }
class B[T] {
  A[T]  vet;
  ...
  };
A[int]  xx;
```

A declaração de `xx` produz vetores `VetT` de dimensões crescentes. Este caso mostra que para evitar recursão infinita é necessário examinar um tipo ou procedimento e tudo o que ele utiliza.

Cm [DdS87] [Fur91] permite usar tipos e valores *default* para os parâmetros da classe. Quando os parâmetros não são especificados na declaração de uma variável da classe, são utilizados os tipos e valores *default* presentes na interface da classe. Figura 4.3 apresenta uma classe `stack` cujo tipo *default* é `int`. O número máximo de elementos da pilha é dada pelo parâmetro `N` de `stack`, cujo *default* é 100. Uma classe não parametrizada pode ser transformada em parametrizada sem afetar os seus clientes. As declarações antigas de variáveis da classe não a utilizam com parâmetros. Com a colocação de parâmetros (com *defaults* para cada um deles) na classe, essas declaraçôes passam a utilizar a classe com os tipos e valores *default* para cada parâmetro. Sem os *defaults*, haveria erro de compilação, pois a classe exigiria a especificação dos seus parâmetros na declaração de suas variáveis. Há um problema é no ajuste do código da própria classe. Suponha que o parâmetro `T` a ser introduzido na classe seja utilizado na classe como inteiro. Algumas declarações de variáveis

```
/* parametro "default"  'e int */
class  A[type T = int]  {
  ...
  void M() {
    T aa;
    int ii ;
    ...
    }
  };
```

Figura 4.5: Classe parametrizada A com parâmetro T cujo *default* é `int`

```
class C1
      var x : object;
end
let C2 = C1[ object ← ring ]
class D1
      var c : C1, a : PointerTo D1 ;
      proc p(arg : object)
      begin c.x := arg end
end
let D2 = D1[ C1 ←`C2 ]
let D3 = D2[ ring ← booleanring]
```

Figura 4.6: Criação de C2, D2, D3 a partir de C1 e D1

inteiras devem ser convertidas para T e outras não. Figuras 4.4 e 4.5 apresentam uma classe antes e depois da conversão para parametrizada, em sintaxe de C++. A declaração das variáveis `aa` e `ii` foi desdobrada em duas.

Qualquer classe pode ser herdada, mas uma classe só é parametrizada se ela foi projetada especialmente para esta finalidade. Para contornar este problema, Petersen e Schwartzback [PS90] propõe um novo mecanismo, substituição de tipo:
Se C, A_i e B_i são classes, $D = C[A_1, \ldots A_n \leftarrow B_1, \ldots B_n]$ especifica D por substituição de tipo a partir de C. Diz-se $C \lhd D$. Os A_i são os componentes de C e B_i, os de D. Em uma forma simplificada de substituição de tipo, cada componente (campo[2]) A_i de C é substituído por B_i, resultando em uma classe D. A_i e B_i devem estar na relação $A_i \lhd B_i$.

$X \lhd Y$ indica que código para X pode ser utilizado com Y. Esta é a relação de subtipo definida por Cardelli [Car84] (Pág 14), com a restrição que se dois tipos em X são iguais, os

[2]Na figura 4.6, os componentes de D1 são c, a e p.

```
class C2
    var x : ring ;
end
class D2
    var c : C2, a : PointerTo D2;
    proc p( arg : ring )
    begin
    c.x := arg
    end
end
```

Figura 4.7: Expansão de C2 e D2

```
class ring
    var value : object
    proc plus ( outro : ring )
    ...
end
class booleanring inherits ring[ object ← boolean ]
    proc plus /* define metodos */
    ...
end
```

Figura 4.8: Classe ring e sua subclasse booleanring

correspondentes em Y também devem ser.

Figura 4.7 mostra C2 e D2 após substituição de tipos da figura 4.6. A substituição não é textual. Para a construção de D2, C2 substitui C1 em D1. Como ring substitui object em C2, este também substitui object em D2. Variável a é um ponteiro para D1 em D1 e ponteiro para D2 em D2 – a recursividade da declaração é preservada. Apesar da classe D2 ser uma instanciação de D1, ela também é parametrizada. D3 e criada a partir de D2 pela substituição de ring por booleanring (fig 4.8). Classe booleanring é subclasse de ring.

A figura 4.9 apresenta um problema com este modelo. A declaração de D2 substitui dois parâmetros em D1, C1 e C3. integer e boolean substituem object em C2 e C3, respectivamente. Ambos devem substituir object em D2. Há um conflito, pois object deve ser substituído por integer (por causa de C2) e também por boolean (por causa de C4). Uma solução seria tornar mais específica a declaração do método p da classe D1:

```
/* type(c.x) retorna o tipo de c.x   */
```

```
class C1
    var x : object;
end
class C3
    var x : object;
end
let C2 = C1[object ← integer];
let C4 = C3[object ← boolean];
class D1
    var c : C1, d : C3;
    proc p( arg : object) ...
end
let D2 = D1[ C1 ← C2, C3 ← C4];
```

Figura 4.9: Conflito entre as 2 substituições de D2

```
proc p(arg : type(c.x) ) ...
       ou
proc p(arg : type(d.x) ) ...
```

A primeira opção causaria a substituição de object por integer, e a segunda, por boolean. Se as variáveis de instância só são manipuladas por métodos, o poder expressivo deste modelo diminui consideravelmente. Os exemplos das figuras 4.6, 4.7, 4.8, 4.9 são adaptações de exemplos de [PS90].

A falta de ortogonalidade de algumas linguagens impede a utilização mais abrangente de classes parametrizadas. Em Eiffel, a instrução `a := b` possui dois significados:

- Com tipos básicos (INTEGER, ...) é cópia.

```
class A[T]
  feature
    a : T ;    --  a 'e uma variavel de instancia e M 'e um metodo
    M is
      ...
      a := b;
    end -- M
    ...
end -- A
```

Figura 4.10: Limitação no uso de parâmetro T

```
VetorReg (# { Isto 'e um pattern virtual }
  { Define tipo Reg como Register }
  Reg :< Register;
  R : @Reg;
  ...
  #) { fim VetorReg }
```

Figura 4.11: *Pattern* virtual VetorReg

```
{VetorPessoa 'e subclasse de VetorReg}
VetorPessoa : VetorReg (#
  Reg :< Pessoa;
  { R agora 'e do tipo Pessoa }
  ...
  #)
```

Figura 4.12: Especialização de VetorReg e Reg

- Se a e b denotam objetos de uma classe, ambos passam a se referir a um mesmo objeto após a operação.

Assim, o parâmetro T da classe A da figura 4.10 deve ser instanciado apenas com tipos básicos ou apenas com classes, por causa dos diferentes significados da operação :=. Do mesmo modo, operação = em C++ possui significado distinto para escalares e vetores.

Em C++, pode-se definir operações como +, *, <= em uma classe. Isto estende a utilização de classes parametrizadas por dois motivos:

- Fornece uma convenção de nomes. Por exemplo, operações de comparação menor do que provavelmente seriam definidas com nomes diferentes em classes diferentes (Less, Menor, A_Menor_A, ...). Uma classe parametrizada teria que admitir que o seu parâmetro utiliza um desses nomes, excluindo a sua utilização com classes que utilizam outros nomes. Observe que Ada permite utilizar um *package* parametrizado com nomes diferentes do descrito na sua interface.

- Um parâmetro de uma classes parametrizada pode ser utilizado por tipos básicos (int, float, ...) e também por classes.

BETA [KOLM87] [MMP89] possui um mecanismo semelhante a classes parametrizadas. É o *pattern*[3] virtual. Basicamente, ele permite a especialização de atributos (variáveis de instância, outros *patterns*) na subclasse. Figura 4.11 mostra *pattern* VetorReg, um vetor de registros. Reg é um tipo que é associado ao tipo Register[4], e pode ser redefinido em

[3]Classe

[4]Definido com declaração semelhante a typedef de C++ ou type de Pascal

```
// Classe A com parametro T cujo "default"  'e  Register
class A[ T = Register] {
  ...
  void M(T x); // void M( Register x);
  };
// B possui T como Pessoa
class B : A[ T = Pessoa] {
  ...
  void M(T x); // void M( Pessoa x);
  };
```

Figura 4.13: B não é subclasse de A. Exemplo de BETA com sintaxe de C++

uma subclasse de VetorReg para uma subclasse de Register. R é uma variável de instância do tipo Reg. A figura 4.12 mostra subclasse VetorPessoa de VetorReg, onde Pessoa é subclasse de Register. R, automaticamente, torna-se do tipo Pessoa. especificando-se Reg como object[5], em VetorReg, com especificado abaixo,:
Reg :< object
qualquer especialização de Reg em subclasses de VetorReg é válida. Neste caso, apenas as operações sobre object, que são as triviais, são utilizadas com R em VetorReg. Comparando com classes parametrizadas, *patterns* virtuais oferecem mais checagem semântica.

Figura 4.13 mostra, com sintaxe de C++, um problema com *patterns* virtuais. Por causa de M, B não é subclasse de A (Cf. seção 2.2). Por este motivo, erros de tipos em execução podem acontecer e são sinalizados adequadamente. O compilador de BETA insere testes para conferência de tipos em execução. Necessidade de testes em execução para prevenção de erros causados pela combinação de classes parametrizadas e herança são discutidos em [MM90].

Tomemos 3 classes : A, SubA e B - todas com métodos e variáveis de instância públicos de mesmo nome. SubA é subclasse de A, B não relacionada com nenhuma outra. Agora, consideremos duas classes parametrizadas, com parâmetros T e U, respectivamente. A primeira exige que T seja substituído por classe A ou seus descendentes. A segunda, exige que U seja substituído por classe que possua a mesma interface que A (mesmas operações públicas). Uma classe parametrizada com parâmetro R (que deve ser uma classe) assume determinado significado para os métodos de R e que determinados relacionamentos semânticos entre os métodos existem. Assim, a primeira classe parametrizada (com parâmetro T) é mais confiável que a segunda (U) porque é mais provável que a semântica das operações de SubA seja mais próxima de A do que é a semântica das operações de B. Por este motivo, classes virtuais de BETA são mais seguras que classes parametrizadas normais.

[5]Superclasse de todas as classes

4.2 Classes Parametrizadas e Herança

Segundo Meyer[Mey86], herança é mais poderosa que parametrização de classes, pois a primeira pode simular a segunda, mas não vice-versa. O modelo de herança utilizado por ele, que é o da linguagem Eiffel, admite que:

- É possível uma declaração por associação:

```
-- codigo dentro de classe A
x : like Current;
...
```

  declara `x` como do mesmo tipo que a classe corrente, que é `A`. Em uma subclasse `B` de `A`, `x` é objeto da classe `B`.

- Uma variável de instância pública pode ser redefinida na subclasse com um tipo que é subtipo do original.

4.2.1 Simulando Herança com Parametrização de classes

Para simular a herança da classe `A` por classes `B` e `C`, a classe `A` deve ser declarada como

```
class  A[ tp : char ]   feature
  M is do ...  end -- Isto 'e  um m'etodo
  -- tp = tipo da classe = 'A', 'B' ou 'C'
  -- Aqui devem estar os campos da classe A - comuns a B e C
  -- Isto 'e um registro variante, como em Pascal
  case tp of
      -- variaveis de instancia de B
    'B' :  Campos_B;
      -- variaveis de instancia de C
    'C' :  Campos_C;
    end -- case
end -- classe A

...
/*  Declara  x  da "classe" B  e y de  C */
local  x : A['B'];
       y : A['C'];
...
```

O registro variante contém os campos específicos de `B` ou (exclusive) `C`. Apenas as variáveis de instância de `B` ou de `C` estarão presentes em objetos de `A`. A variável de instância `tp` indica qual o tipo real de um objeto da classe `A`, que pode ser classe `A`, `B` ou `C`. Funções virtuais são simuladas escolhendo o código a ser executado (De `A`, `B` ou `C`) em execução. O exemplo

```
class  A[T] feature
  vet : ARRAY[T] ;
  M( x : T ) : T is
    local
      a, b : T;
    do
      ...
      if b.menor(a) then
      ...
    end -- M
  N( x : A[T]) : A[T] is do ... end
  ...
end -- classe A
```

Figura 4.14: Classe parametrizada com sintaxe de Eiffel

abaixo mostra a codificação do método `M` de `A`. O comando `Case` desvia a linha de execução de acordo com o tipo do objeto corrente, dado pela variável `tp`.

```
method  A::M is
  begin
  Case tp of
      'A' :  ...  // classe A
      'B' :  ...  // classe B
      'C' :  ...  // classe C
    end -- Case
  end -- metodo M
```

Para cada nova subclasse de `A`, deve ser acrescentado um campo no registro variante desta classe com as variáveis específicas desta subclasse. E cada método virtual deve ser modificado. Métodos específicos de B ou C, que são descritos na interface da classe `A` devem produzir um erro em execução se forem invocados por objetos da classe errada. Não se pode considerar que este modelo simule herança, pois a criação de subclasses envolve modificações na classe. A parametrização de classes é facilmente dispensável nesta simulação.

4.2.2 Simulando Parametrização de Classes com Herança

Usaremos um exemplo para tornar a discussão mais didática. A classe parametrizada da figura 4.14 será simulada com herança. Apenas a sintaxe do exemplo é de Eiffel, mas esta linguagem não suporta alguns dos mecanismos utilizados. Há três construções que devemos considerar na classe `A`:

[1] Declaração de variável de instância `vet`, parâmetro `x` de `M`, valor de retorno `M` e variáveis `a` e `b` de `M` com tipo `T`.

```
class Ordem feature
  menor( t : like Current ) : boolean is  deferred end
end -- classe Ordem

class A
  freeze ref
  feature
    ref : Ordem;
    vet : ARRAY[ like ref ];
    M( x : like ref ) : like ref is
      local
        a, b : like ref;
      do
        ...
        if b.menor(a) then
        ...
      end
      N( x : like Current) : like Current is
      do
      ...
      end
end -- classe A
```

Figura 4.15: Classe que simula uma classe parametrizada

[2] Declaração do parâmetro `x` de `N` e valor de retorno com tipo `A[T]`.

[3] Invocação de método `menor` de `T` no método `M` de `A`.

Vejamos como estas construções são modificadas na simulação da classe parametrizada `A` por herança, mostrada na figura 4.15:

[1] Variável `ref` de `A` é uma referência para declaração de outras variáveis. Declarações do tipo `T` na classe `A` da figura 4.14 transformam-se em declarações do tipo `like ref` na figura 4.15. Esta variável é congelada - `freeze`. É uma variável de classe, compartilhada por todos os objetos de `A`. Redefinindo `ref` para um tipo `OrdemInt` em uma subclasse `B` de `A`, todas as definições que o utilizam são atualizadas automaticamente. Assim, `vet` tem o tipo `ARRAY[ OrdemInt ]` na classe `B` da figura 4.16.

[2] `A[T]` é substituído por `like Current` na figura 4.15. Se `A[T]` fosse substituído por `A`, haveria erro de tipos em uso de `B` da figura 4.16:

```
z, y : B;
...
```

```
class OrdemInt
  inherits Ordem
  feature
    valor : integer;
    menor( t : like Current ) : boolean is
      do
      if valor < t.valor then
        Result:= true
      else
        Result:= false
      end
end -- classe OrdemInt

class B inherits A
  ref : OrdemInt;
end -- classe B
...
y : B;
```

Figura 4.16: Uso de herança para simular a instanciação de uma classe parametrizada

```
class OrdemInt feature
  valor : integer ;
  menor( t : OrdemInt ) is ...
end -- classe OrdemInt
...
y : A[OrdemInt];
```

Figura 4.17: Uso de classe parametrizada `A`

```
z:= y.N(z);
```

Variável da classe `B` (`z`) recebe objeto (`y.N(z)`) da superclasse `A` de `B`. Como apresentado na figura 4.15, método `N` da classe `B` de 4.16 retorna objeto da classe `B`.

[3] É criada uma classe abstrata com todos os métodos utilizados em `A`. Esta classe é `Ordem`, e corresponde ao parâmetro `T`. A palavra chave `deferred`, na declaração do método `menor`, indica que a classe `Ordem` é abstrata e que este método será implementado nas subclasses. tPara a utilização de A com um "parâmetro" diferente de `Ordem`, deve-se criar uma subclasse desta classe, como `OrdemInt`, mostrada na figura 4.16.

Declaração de `y` nas figuras 4.17 e 4.16 são equivalentes. A figura 4.17 utiliza a classe `A` da figura 4.14.

Observe que o uso da classe `A` da figura 4.14 é mais fácil. Com a classe `A` da figura 4.15, deve-se criar uma subclasse de `Ordem` para cada instanciação. Há um problema se for necessário utilizar a classe `A` com um "parâmetro" que não é subclasse de `Ordem`. Deve-se criar uma classe que herde desta classe e de `Ordem`.

Capítulo 5

Uma Extensão a C++

5.1 @Classes e @métodos

Esta seção apresenta uma sugestão de extensão a C++ chamada @classe.

Uma @classe é uma classe com pelo menos um @método. Quando ela é herdada por uma classe `B`, cada @método é recompilado dentro do contexto de `B`. Um @método é um método normal cujo nome é prefixado por @. Os métodos que `B` herda, então, são os métodos recompilados. A palavra chave `like` é utilizada para amarrar o tipo de uma variável a outra. Se uma variável `y` é declarada como tendo o tipo `like(x)`, `y` possui o mesmo tipo que a variável `x`. Na definição de uma @classe, o nome da @classe é prefixado com o caracter @. Sintaxe análoga é empregada com @métodos. Um exemplo com @classes é dado na figura 5.1, em sintaxe de C++. `this` é um ponteiro para o objeto que recebe a mensagem (*self*). `like(this)` é um ponteiro para a classe corrente. Na classe `A`, os métodos `N` e `M` são dos seguintes tipos:

```
A *N();
void M(A elem);
```

Em B são:

```
B *N();
void M( B elem );
```

Outros métodos de `A` podem invocar `M` normalmente, pois o envio de uma mensagem não exige conhecimentos dos parâmetros `X` e `Max` de `M`. Este método não deve ter parâmetros reais[1]. O método *poderia* ter parâmetros formais[2] que utilizazem os seus parâmetros `X` e `Max`. Contudo, nenhum outro método de `A` teria permissão para invocar `M`. O motivo é que a interface do método `M` poderia se tornar diferente em diferentes subclasses de `A`. A interface de `M` é formada pelo seu nome e pelos nomes e tipos de seus parâmetros formais. Um exemplo é dado na figura 5.3. A interface de `P::M` e `Q::M` são, respectivamente:

[1] Aqueles passados ao método durante a execução do programa e antes de sua invocação

[2] Os descritos entre os parenteses que se seguem ao nome do método no seu cabeçalho na interface da classe

```
class @A {
  ...
  public:
    like(this)  @N() { ... }
    void @M( like(this)  elem ) { ... }
  };
class B : public @A { ... }
...
B   p, *r, *s ;
p.M(r);
s = p.N();
```

Figura 5.1: Um exemplo com @classes

```
class @A {
  ..
  public:
  void @M[ type  X = A* ; int Max] () { ... }
  ...
  };
class E : A {
  ...
  public:
  @= M[ G, 10];
  };
class @B : A {
  ...
  public:
   @= M[H,10];
    void @M[int n]() { ... }
  };
class F : B {
  ...
  public:
    @= M[20];
  };
```

Figura 5.2: @Classes e métodos parametrizados

```
class @A {
  ...
  public:
    void @M[ type X = A*, int Max] ( X a) { ... }
    ...
  };
class P : A {
  ...
  public:
    @= M[P,10];
  };
class Q : A {
  ...
  public:
    @= M[Q,20];
  };
```

Figura 5.3: Métodos de A não podem invocar M

```
void M( P a );
   e
void M( Q a );
```

O compilador estaria incapacitado de fazer as conferências de tipos com relação ao parâmetro de M em outros métodos de A que utilizam M. O mesmo ocorre com a construção "like(this)" da figura 5.1. As subclasses de A das figuras 5.1 e 5.3 não são consideradas subtipos de A. Não é possível utilizar um ponteiro para B onde se espera um ponteiro para A.

A semelhança de @classes com classes parametrizadas instanciadas em compilação nos conduz à criação de métodos parametrizados. Estes últimos são @métodos cujos parâmetros reais devem ser fornecidos nas subclasses. A instanciação (ato de fornecer parâmetros reais) de um @método é sinalizado pelos símbolos @= antes do nome do @método. Com os parâmetros contendo valores reais em subclasses, o @método é compilado. Um exemplo é dado na figura 5.2, onde o @método M de A recebe os parâmetros efetivos G e 10 na classe E. Este método é então compilado para a classe E. O código resultante é associado a E::M. O @método M de A contém dois parâmetros. O primeiro, X, deve ser um tipo qualquer dentre os permitidos pela linguagem, e possui valor *default* "A *" (ponteiro para A). O segundo, Max, deve ser uma constante inteira e não possui valor *default*. Uma @classe pode herdar de outra @classe, como acontece com B. Esta @classe instancia o @método M herdado de A e o redefine com outros parâmetros. A classe F herda de B instanciando M redefinido por B.

Esta facilidade não possui muita utilidade apenas como foi apresentada. Porém, um método normal herdado pode se transformar em um @método e vice-versa, desde que a quantidade e tipos dos parâmetros do método não sejam afetados. Não é utilizada nenhuma notação especial para esta transformação. Um tipo de método (@método ou normal) é

```
class A {
  ...
  virtual void N();
  };
class @B : A {
  ...
  virtual void @N[int Max = 10] () {
    float vet[Max];
    ...
    };
  };
class C : B {
  ...
  @= N[20];
  };
class D : B {
  ...
  virtual void N() { ...}
  };
...
C *h;   A *f;
f = h;
f->N();  //  invòca B::N com parametro 20.
```

Figura 5.4: Transformação método normal para @método e vice-versa

simplesmente redefinido como de outro tipo em uma subclasse. Na figura 5.4, o método normal N de A transforma-se em @método em B. E @método N de B torna-se normal em classe D.

A importância de @classes está na transformação de métodos normais em @métodos. Deste modo, de uma classe normal pode ser criada uma subclasse que possui @métodos como parâmetros. Soluções específicas dadas por uma superclasse podem tornar-se soluções mais gerais (para infinitos tipos ou valores) nas subclasses. Um método virtual redefinido como @método em uma subclasse continua sendo virtual. Assim, a instrução "f->N()" na figura 5.4 causa a execução de B::N instanciado com parâmetro Max = 20, tomado de C.

Qualquer classe pode ser herdada, mas nem todas são parametrizáveis - página 45. @Classes não resolvem este problema, mas oferecem mais flexibilidade que classes normais.

Semelhante a classes parametrizadas, uma @classe sem valor *default* para os parâmetros de seus @métodos não pode ser usada para a criação de objetos.

5.2 Linguagem de macros

Temos uma estrutura S, em C++:

```
struct { int x, y; } S;
```

aa e bb são variáveis do tipo S. Queremos copiar TODOS os campos de aa para bb. Isto pode ser feito da seguinte forma:

```
bb.x = aa.x ;
bb.y = aa.y ;
```

O código acima implica que os campos x e y de aa são copiados para bb mas não determina a copia de TODOS os campos de aa para bb. Um pré-processador embutido no compilador com uma linguagem própria resolveria este problema:

```
/* Ate aqui ha codigo normal de C++ */
@begin /* inicio do codigo para o pre-processador */
  /* Este for 'e um comando do pre-processador.
     cp 'e uma variavel, S 'e o nome da estrutura.
     aa e bb sao os nomes das variaveis.
     fields 'e uma funcao pre-definida que retorna o conjunto dos
     membros de uma estrutura. */
  for cp in  fields(S) do
    write  bb.cp = aa.cp;
@end /* fim do codigo para o pre-processador - continua codigo em C++ */
```

A saída do pré-processador é dado pelo comando write dentro do for. O código entre @begin e @end produziria:

```
bb.x = aa.x;
bb.y = aa.y;
```

que seria então compilado. Todas as informações disponíveis na tabela de símbolos do compilador estão disponíveis para o pré-processador.

Chamaremos a linguagem empregada no pré-processador de L@. A princípio, deve ser um subconjunto da linguagem em que ela é embutida.

Esta facilidade permite "propagar" modificações. Mudanças, como acréscimos ou retiradas de campos de S seriam refletidas no código acima. Outros exemplos com o uso de L@ incluem teste de tipo. De acordo com o tipo de uma variável, escolhe-se um algoritmo:

```
// L@ embutida em C++
@begin
  // strtype retorna string com o nome do tipo do parametro.
  if strtype(t[0]) == "int"
  then
    // copie t para s com funcao memcpy
```

```
type matriz(lin, col : integer) is array(1..lin, 10..col) of real;
...
a : matriz(10,20);
b : array[1..a'lin] of real;
c : array[ a'RANGE(1) ] of real ;
```

Figura 5.5: Uso de informações do compilador em código fonte

```
    write  memcpy( s, t, N*sizeof(int) );
  else
    //  t[0] pode ser objeto de uma classe. Copie usando
    //  a forma mais geral possivel (Usando operador [] dos tipo t e s)
    write for (int i = 0;  i < N ;  i ++)
    write    s[i] = t[i];
  endif
@end
```

Situações com esta ocorrem na prática. O propósito desta idéia é colocar tanto quanto possível a semântica do problema em código fonte. Observe a distância entre a semântica do problema e o código em C++ no primeiro exemplo. A real intenção do programador pode ser expressa em L@.

A seção 4.1 (fig. 4.3, 4.4, 4.5) apresenta classes parametrizadas com parâmetros *default* e um exemplo de conversão de uma classe normal em parametrizada. Variáveis `aa` e `ii`. que são declaradas juntas na figura 4.4, são separadas após a colocação do parâmetro `T`, na figura 4.5. Este último código expressa melhor as restrições que o problema oferece. Variável `aa`, por exemplo, não pode mais ser usada como índice de um vetor.

Ada permite utilizar algumas informações do compilador no código. Na figura 5.5[3], há uma declaração de um tipo `matriz` e variáveis `a`, `b` e `c`. `matriz` é uma estrutura indexada parametrizada. O número de linhas de `b` é amarrado ao número de linhas de `a`. `a'RANGE(i)` são os limites dos índices da i-ésima dimensão de `a`:

```
a'RANGE(1) 'e   1..10
a'RANGE(2) 'e   10..20
```

O primeiro e último valor de um tipo enumerado é obtido pelos "atributos" `FIRST` e `LAST`. Acréscimos de nomes ao enumerado não invalidam qualquer código que usa estes atributos. Eles continuam a indicar o primeiro e o último membro do enumerado.

[3] Adaptado de exemplo de [Mey88]

Capítulo 6

Conclusão

6.1 Sobre a Tese

Orientação a objetos é a evolução mais importante de linguagens de programação desde o surgimento de programação estruturada[1]. Ela permite a organização de um sistema em classes, estruturando-o. Herança traz flexibilidade às classes, admitindo a criação de subclasses com redefinição de métodos herdados. O reaproveitamento de *software* é conseguido com as relações de subtipo e classes parametrizadas. Subtipo é associado a subclasse em muitas linguagens.

Existem muitos problemas com as linguagens orientadas a objeto existentes e com o modelo convencional de orientação a objetos. Algumas dessas falhas são:

- Falta de suporte para programação de baixo nível.
- Presença de construções que dificultam ou impossibilitam a otimização, como reflexividade da linguagem, e não exigência de declaração de tipos para variáveis.
- Mecanismos deficientes para proteção de informação. O ideal é a presença de três níveis de visibilidade (público, privado e protegido), presente em algumas linguagens (C++, Simula).
- Ausência de módulos. Classes são uma unidade de pequena abstração, insuficientes para refletir a estrutura lógica de um sistema.

As contribuições desta dissertação são:

- Identificação dos problemas da seção 3.
- Parametrização de métodos, apresentada na seção 5.1.

Esta dissertação possui inúmeras falhas, pois o estilo do autor não é de fácil compreensão. Há muitas frases soltas seguidas de uma conclusão, com poucos indicativos de como estas frases conduzem a esta conclusão.

[1] Apesar de, a rigor, ter nascido antes dela, considerando-se as datas de criação de Simula-67 e publicação do artigo [Dij68]

A continuidade deste trabalho envolve a tentativa de resolução de outros problemas com orientação a objetos (capítulo 3). Outra direção de pesquisa são as linguagens paralelas orientadas a objeto. Computadores multiprocessados estão se tornando uma realidade e existe a necessidade do aproveitamento da pontencialidade dessas máquinas através de linguagens adequadas.

Bibliografia

[Ack82] William B. Ackerman. Data Flow Languages. *Computer*, 15(2), February 1982.

[Ada81] *The Programming Language Ada*, volume 106 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Agh86] Gul Agha. An Overview of Actor Languages. *SIGPLAN Notices*, 21(10), October 1986.

[AH] Timothy Andrews and Craig Harris. Combining Language and Database Ad ..nces in an Object-Oriented Development Environment.

[AT88] Mehmet Aksit and Anand Tripathi. Data abstraction mechanisms in Sina/st. *SIGPLAN Notices*, 23(11), September 1988. Object Oriented Programming, systems, languages and Applications.

[AvdL90] Pierre America and Frank van der Linden. A parallel Object-Oriented Language with Inheritance and Subtyping. *SIGPLAN Notices*, 25(10), October 1990. ECOOP/OOPSLA.

[BC90] G. Bracha and W. Cook. Mixin-based Inheritance. *SIGPLAN Notices*, 25(10), October 1990. Object Oriented Programming, systems, languages and Applications.

[C+88] Luca Cardelli et al. Modula-3 Report. Technical report, Digital, Systems Research Center, August 1988.

[Car84] Luca Cardelli. The Semantics of Multiple Inheritance . In *Colloquium on the Semantics of Data Types*. Springer-Verlag, 1984. Em Lecture Notes in Computer Science Num. 173. Pág. 51-58.

[CG90] Bernad Carré and Jean-Marc Geib. The Point of View Notion for Multiple Inheritance . *SIGPLAN Notices*, 25(10), October 1990. Object Oriented Programming, systems, languages and Applications.

[CUL89] Caraig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dinamically-Typed Object- Oriented Language Based on Prototypes. *SIGPLAN Notices*. 24(10), October 1989. OOPSLA. Pág. 49-52.

[CW85] Luca Cardelli and Peter Wegner. On Understandign Types, Data Abstration and Polimorphism. *ACM Computing Surveys*, 17(4), December 1985. Pág. 471-483.

[Dah73] Ole Dahn. A Comparison of Simula I and Simula 67, November 1973. Norwegian Computing Center, OSLO 3 – Norway, Publication No. S-57.

[DdS87] Rogério Drummond and Fábio Q.B. da Silva. Manual de Referência Cm. Technical report, Unicamp-IMECC-DCC, 1987.

[DG88] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. *SIGPLAN Notices*, 23(11), September 1988. Object Oriented Programming, systems, languages and Applications.

[Dij68] Edsger W. Dijkstra. Go to Statemente Considerad Harmful, March 1968.

[DK82] Alan L. Davis and Rovert M. Keller. Data Flow Program Graphs. *Computer*, 15(2), February 1982.

[Fur91] Carlos Alberto Furuti. Um Compilador para uma Linguagem de Programação Orientada a Objetos. Master's thesis, DCC-Unicamp, July 1991.

[GG77] David Gries and Narain Gehani. Some ideas on Data Types in High–Level Languages. *Communications of the ACM*, 20(6), June 1977.

[GR83] Adele Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation* . Addison-Wesley, 1983.

[Har] Robert Harper. Introduction to Standard ML. Technical report, Laboratory for Foundations of Computer Science. University of Edinburgh. Pág. 1-13, 20, 22.

[HO88] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Languages. In *European Conference on Object Oriented Programming*. Springer-Verlag, 1988. Em Lecture Notes in Computer Science Num. 322.

[Ing86] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. *SIGPLAN Notices*, 21(11), November 1986. Object Oriented Programming, systems, languages and Applications.

[Kae81] Ted Kaehler. Virtual Memory for an Object-Oriented Language. *BYTE*, August 1981.

[KG87] Gail E. Kaiser and David Garlan. MELDing Data Flow and Object-Oriented Programming . *SIGPLAN Notices*, 22(12), December 1987. Object Oriented Programming, systems, languages and Applications.

[KM87] Jϕrgen Lindskov Knudsen and Ole Lehrmann Madsen. Teaching Object-Oriented Programming Language is More than Teaching Object–Oriented Programming Languages. *Lecture Notes in Computer Science*, 276, 1987.

[Knu88] J. Lindskov Knudsen. Name Collision in Multiple Classification Hierarchies . In *European Conference on Object Oriented Programming*. Springer-Verlag, 1988. Em Lecture Notes in Computer Science Num. 322.

[KOLM87] B.B. Kristensen and K. Nygaard Ole Lehrmann Madsen, Birger Moller-Pedersen. *Research Directions in Object-Oriented Programming*, chapter The BETA Programming Language. MIT, 1987. 7-27.

[Lis87] Barbara Liskov. Data Abstraction and Hierarchy. *SIGPLAN Notices*, 1987.

[LSRS77] B. Liskov, A. Snyder, R.Atkinson, and C. Schaffert. Abstration Mechanisms in CLU. *Communications of the ACM*, 20(8), August 1977.

[Mey86] Bertrand Meyer. Genericity versus Inheritance. *SIGPLAN Notices*, 21(11), November 1986. Object Oriented Programming, systems, languages and Applications.

[Mey87] Bertrand Meyer. Eiffel: Programming for Reusability and Extendibility. *SIGPLAN Notices*, 22(2), February 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction* . Prentice Hall, 1988.

[MM90] Ole Lehrmann Madsen and Boris Magnuson. Strong Typing of Object-Oriented Languages Revisited. *SIGPLAN Notices*, 25(10), October 1990. ECOOP/OOPSLA.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual Classes - A Powerful Mechanism in Object-Oriented Programming . *SIGPLAN Notices*, 24(10), October 1989. Object Oriented Programming, systems, languages and Applications.

[Moo86] David A. Moon. Object–Oriented Programming with Flavors. *SIGPLAN Notices*, 21(11), November 1986. Object Oriented Programming, systems, languages and Applications.

[MR87] Naftaly H. Minsky and David Rozenshtein. A Law-Based Approach to Object-Oriented Programming . *SIGPLAN Notices*, 22(12), 1987. Object Oriented Programming, systems, languages and Applications.

[PS90] J. Palsberg and M. I. Schwartzbach. Type Substitution for Object-Oriented Programming . *SIGPLAN Notices*, 25(10), October 1990. ECOOP/OOPSLA.

[RID90] R.Helm, I.M.Holland, and D.Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems . *SIGPLAN Notices*, 25(10), October 1990. Object Oriented Programming, systems, languages and Applications.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, A. P. Black, et al. Emerald: A General Purpose Programming Language. *Software-Pratice and Experience*, 21(1), January 1991.

[S+86] Craig Shaffert et al. An Introdution to Trellis/Owl. *SIGPLAN Notices*, 21(11), November 1986. Object Oriented Programming, systems, languages and Applications.

[SHBR89] Shah, Hamel, Borsari, and Rumbaugh. DSM: An Object-Relationship Modeling Language. *SIGPLAN Notices*, 24(10), October 1989. Object Oriented Programming, systems, languages and Applications.

[Sny86] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Language . *SIGPLAN Notices*, 21(11), November 1986. Object Oriented Programming, systems, languages and Applications.

[Sny87] Alan Snyder. *Research Directions in Object-Oriented Programming*, chapter Inheritance and The Development of Encapsulated Software Components. MIT, 1987.

[SWL77] Mary Shaw, William A. Wulf, and Ralph L. London. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. *Communications of the ACM*, 20(8), August 1977. Pág. 553-555.

[Tim86] Panel: The Learnability of Object-Oriented Programming Systems, November 1986. Object Oriented Programming, systems, languages and Applications.

[TL90] Tadao Takahashi and Hans K.E. Liesenberg. *Programação Orientada a Objetos.* VII Escola de Computação, São Paulo. Julho 1990.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *SIGPLAN Notices*, 22(12), December 1987. Object Oriented Programming, systems, languages and Applications.

[WBW88] Allen Wirfs-Brock and Brian Wilkerson. An overview of modular Smalltalk. *SIGPLAN Notices*, 23(11), September 1988. Object Oriented Programming, systems, languages and Applications.

[Weg87a] Peter Wegner. Dimensions of Object-Based Language Design. *SIGPLAN Notices*, 22(12), December 1987. Object Oriented Programming, systems, languages and Applications.

[Weg87b] Peter Wegner. *Research Directions in Object-Oriented Programming*, chapter The Object-Oriented Classification Paradigma. MIT, 1987. Partes I, II e IV.

[Wir83] Niklaus Wirth. *Programming in Modula-2* . Springer-Verlag, 1983.

[WZ87] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *European Conference on Object Oriented Programming*. Springer-Verlag, 1987. Em Lecture Notes in Computer Science Num. 276.