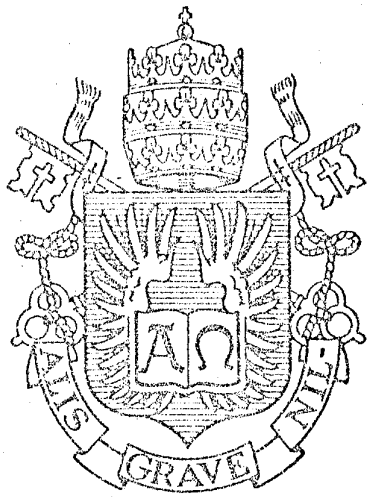



PUC

Série : Monografias em Ciência da Computação
N.11/89

# Princípios e Polêmicas sobre "ORIENTAÇÃO A OBJETOS"

Graciela H. Matich
Sergio Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 — CEP 22453

RIO DE JANEIRO — BRASIL

PUC/RJ - DEPARTAMENTO DE INFORMATICA

Série: Monografias em Ciência da Computação
N.11/89

Editor: Paulo Augusto Silva Veloso JUN, 1989

# Princípios e Polêmicas sobre "ORIENTAÇÃO A OBJETOS" *

Graciela H. Matich **
Sergio Carvalho

* Trabalho parcialmente financiado pela FINEP
** Cursando o programa de doutoramento do DI.

Resumo

O objetivo principal deste trabalho é o estabelecimento de características primárias ou básicas para linguagens orientadas a objetos, discutindo importantes conceitos ligados a este tema, como TIPAGEM, AMARRAÇÃO e HERANÇA.
Palavras Chave: LINGUAGENS ORIENTADAS A OBJETOS, TIPOS DE DADOS, HERANÇA.

Abstract

In this work we primarily try to establish the basic characteristics of object oriented programming languages, discussing important related topics, such as TYPES, BINDING and INHERITANCE.
Keywords: OBJECT ORIENTED LANGUAGES, DATA TYPES, INHERITANCE.

Responsável por Publicações

Rosane Teles Lins Castilho
PUC/RJ-Depto. de Informática
Assessoria de Biblioteca, Documentação e Informação
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ

SUMÁRIO

página

# 1 - Introdução

## 1.1 - Motivação

Quais os fatores que determinam o sucesso explosivo da "ORIENTAÇãO a OBJETOS"?, sendo que encontramos diferentes acepções do termo, diferentes aplicações, discrepâncias, ... e assim por diante.

Evidentemente, "quando ronca trovoada è porque vem vindo tempestade" e nesta tempestade que se espalhou, distinguem-se pelo menos três causas:

(1) o uso embutido de mecanismos de abstração;

(2) a contribuição à REUSABILIDADE;

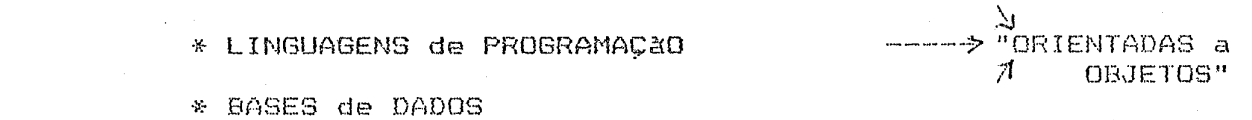
(3) o esquema de classificação ou categorização de domínios de problemas, especialmente apreciado na programação de muito grande porte ("programming in the very large").

Os benefícios de uso de ABSTRAÇOES são amplamente reconhecidos. ABSTRAÇOES DE DADOS vão ganhando prestigio e preponderância sobre as já enraizadas ABSTRAÇOES DE CONTROLE e afirma-se quase universalmente a proposta de reunir estes dois poderosos pilares no conceito de TIPOS ABSTRATOS DE DADOS. Os OBJETOS assentam-se sobre esta base.

Por outro lado, è generalizado o panorama de crises no software. Os sistemas e programas ficam obsoletos em curto período de tempo, com grande decepção dos projetistas e empresários. O custo de projetos de novos sistemas è muito elevado. Existe diversidade de componentes de software (programas, procedimentos e estruturas de dados) similares. Esta situação levou a que os olhos se voltassem para o reaproveitamento de esforços e de código, fomentando as pesquisas em torno da REUSABILIDADE. A programação orientada a objetos proporciona um promissor conjunto de soluções para a REUSABILIDADE. [MEY87]

## 1.2 - "ORIENTAÇÃO a OBJETOS", mesma etiqueta para diferentes produtos

O rótulo de "ORIENTAÇÃO a OBJETOS" aparece cada vez com maior freqüência, ligado a empreendimentos de software, podendo destacar do grande leque, três que tem especial força:




Os AMBIENTES de DESENVOLVIMENTO de SOFTWARE incluem em geral, interfaces especificas com o usuário, sistemas de JANELAS e uma série de facilidades visuais e gráficas, que são de ajuda apreciável para o projetista de software, porém não são relativas ao termo "ORIENTAÇÃO a OBJETOS aplicado a linguagens. Poderia aparecer esse tipo de facilidades em ambientes onde se trabalhe com linguagens que não tem relação nenhuma com a ORIENTAÇÃO a OBJETOS.

As LINGUAGENS ORIENTADAS a OBJETOS, meta de análise do presente trabalho, tem características próprias que as individualizam, podendo ou não estar ligadas a AMBIENTES de DESENVOLVIMENTO "ad hoc".

Sistemas gerenciadores de bases de dados surgiram no mundo informático inscritos na mesma corrente [MEY80], [COP84], [STO86], [AND87], percebendo a importância de incorporar poderosos mecanismos de abstração como ferramentas para assistir à modelagem conceitual, adotando facilidades das ditas linguagens ORIENTADAS a OBJETOS e enfatizando alguns aspectos (como a PERSISTENCIA, ver 1.4.4) que para as linguagens como tais não são vitais.

## 1.3 - No contexto das LINGUAGENS de PROGRAMAÇÃO, diferentes critérios sobre o que significa SER ORIENTADA a OBJETOS

Cultivando um campo propicio para a confusão, o termo "ORIENTAÇÃO a OBJETOS" aplicado a LINGUAGENS de PROGRAMAÇÃO significa diferentes coisas para diferentes pessoas.

Para alguns o conceito não é mais do que uma nova forma de ABSTRAÇÕES de DADOS, onde dados e operações estão encapsuladas em objetos (ou módulos, ou "clusters", ou "packages").

Para outros, OBJETOS e CLASSES são uma forma concreta de teoria de tipos. Nesta visão cada dado (objeto) è considerado como um elemento de um tipo (classe). Tipos complexos formam-se a partir de tipos bàsicos aplicando operações matemàticas, tais como produto cartesiano, disjunções etc, e os tipos podem estar vinculados uns com os outros em relações da forma subtipo e supertipo [HAI87].

Porèm, para alguns outros, a "orientação a objetos" dà ênfase ao tema da organização e formas de compartilhar còdigo.

## 1.4 - Princìpios bàsicos

### 1.4.1 - GENERALIDADES

A abordagem "ORIENTADA a OBJETOS" em verdade, combina e sustenta vàrios conceitos.

E importante distinguir que podemos observar a "ORIENTAÇàO A OBJETOS" atravès de duas perspectivas: como modelo computacional e como filosofìa de desenvolvimento. [TAD88]

Como modelo computacional este estilo se baseia em OBJETOS (càpsulas de dados e procedimentos) que trabalham e cooperam entre si atravès do envio de mensagens para realizar uma tarefa dada.

O conceito de OBJETO certamente nàn è novo, foi montado sobre a idèia de TIPOS ABSTRATOS de DADOS, encapsulando dados e operações. Cada OBJETO tem sua zona da memòria pròpria que reflete o ESTADO do objeto e funções locais que podem alterar o estado do objeto. Se esses objetos sò sàn tratados na linguagem por suas funções especìficas, como è a intenção do conceito, isto resulta em MODULARIDADE e OCULTAMENTO de INFORMAÇàO.

Parte-se do critèrio que tem mais sentido definir caracterìsticas de um grupo de objetos que de cada objeto isoladamente. As caracterìsticas gerais de um grupo de objetos sàn dadas numa CLASSE. Uma CLASSE proporciona toda a informação necessària para construir e usar uma certa coleção de objetos, que seràn suas instàncias. Todo objeto è entàn uma INSTANCIA de uma determinada CLASSE.

Como filosofìa de desenvolvimento, o domìnio de um problema considera-se organizado em classes e pode chegar a resumir-se em um paradigma de classificação. [WEG86]

O programador trabalha usando classes jà existentes e/ou definindo novas classes e dando vida a objetos que sempre pertencem a uma determinada classe.

A HERANÇA torna possivel definir uma classe em termos de uma (ou mais) classes já existentes, reaproveitando caracteristicas de representação e comportamento de niveis superiores de abstração. Para tal finalidade, as classes relacionam-se em uma hierarquia, que podemos visualizar, em sua forma mais rudimentar, como uma estrutura em forma de árvore, onde existe na raiz a classe mãe de todas as classes, a de maior nivel de generalidade conceitual. Os nós não terminais são classes que vão derivando umas das outras numa relação similar a ancestrais e descendentes (existindo em algumas linguagens uma terminologia de subclasses e superclasses para indicar as classes derivadas e as classes progenitoras ou ancestrais respectivamente). Os objetos (instâncias de uma classe) correspondem ás folhas ou nós terminais da árvore.

### 1.4.2 - Terminologia especifica

Esta terminologia não è exatamente igual em todas as LOOs, mais existe um sólido consenso.

**OBJETOS:** entidades semiautônomas que se comunicam através de mensagens.

OBJETO reúne tanto **ESTADO** como **COMPORTAMENTO**.

E sinônimo de **INSTANCIA** de **CLASSE**.

**CLASSES:** servem como moldes para criação de objetos.

Uma classe define representação e comportamento de uma coleção de objetos similares (que são suas instâncias).

Classe coincide com o conceito de **TIPO** na maioria das linguagens convencionais onde TIPO è usado principalmente para descrever "tipos embutidos": inteiro, real, caracter, etc.

Cada instância (objeto) pertence a uma classe, mas uma classe pode ter multiplas instâncias.

O conceito guarda idéias bastantes intuitivas, pensemos por exemplo numa classe CORES, algumas de suas instâncias (objetos) poderão ser azul, amarelo, branco, etc.

**METODOS:** è o nome que empregam muitas LOOs para referir-se aos procedimentos que implementam o comportamento operacional de uma classe de objetos.

Um mètodo è invocado enviando uma determinada mensagem a um objeto.

Os mètodos residem nas classes, desde de que todas as instâncias (objetos) de uma classe tem idêntico conjunto de mètodos.

**MENSAGENS:** solicita-se a um objeto realizar uma das operações entendíveis para ele, enviando-lhe uma mensagem. Esta mensagem diz apenas ao objeto o que deve fazer, e ela está dada por uma EXPRESSãO de MENSAGEM. Uma expressão de mensagem è anàloga a uma função comum, sò que quando ela è executada (a expressão) tem um passo extra, um mecanismo de SELEÇãO de METODO.

As expressões de mensagem tem:

- destinatàrio (objeto receptor da mensagem);
- seletor (diz ao objeto o que fazer, tem o nome do mètodo a executar);
- argumentos (opcionalmente e dependendo da mensagem).

1.4.3 - Como operam as mensagens ?

Uma computação, na execução de um programa, se realiza enviando uma mensagem a um objeto, como jà foi mencionado. Na execução da mensagem, o objeto receptor responde à mensagem da seguinte forma:

* procurando a operação (ou mètodo) que implementa o nome da mensagem ou seletor (MECANISMO de SELEÇãO);
* executando esta operação ou mètodo;
* retornando um resultado e o controle ao objeto que enviou a mensagem.

**MECANISMO de SELEÇãO de mètodo:** o objeto procura primeiro no repertòrio de mètodos de sua classe, se aí não acha um mètodo com esse nome, considerando a HIERAQUIA de CLASSES e o mecanismo de HERANÇA (numa versão simples), vai procurar na classe imediata superior ou progenitora. Se também acontece que o mètodo não è do "microcosmos" definido por esta outra classe, continua procurando de forma ascendente pela classe progenitora da última e assim por diante. O primeiro mètodo que achar, com esse nome, è o que executa. Se não acha nenhum que se corresponda com esse nome, responderà que desconhece o mètodo.

1.4.4 - O que é fundamental e o que não é ?, para que uma linguagem de programação entre na categoria de ORIENTADA a OBJETOS ?

Este ponto pode apresentar, como foi mencionado prèviamente, notàveis discordàncias, mas tentou-se estabelecer uma conclusão como resultado de um estudo de diferentes autores e linguagens.

Existem uma sèrie de conceitos que dançam no espaço de modelagem de linguagens orientadas a objetos.

(1) ENCAPSULAMENTO e ABSTRAÇÕES de DADOS;

(2) NOÇàO de CLASSE;

(3) HERANÇA;

(4) CONCORRêNCIA;

(5) "LATE BINDING" ou resolução tardia de vinculos;

(6) PERSISTêNCIA.

Analisando em ordem inversa à da sua apresentação, a PERSISTENCIA (6), è a propriedade dos dados que determina quanto devem ser mantidos, qual sua durabilidade ou tempo de vida. Em geral è aplicado como a SOBREVIVENCIA a um processo ou seção de programa. Em linguagens tradicionais o tempo de vida dos dados geralmente não transcende do tempo de vida de um programa particular (obviamente são consideradas externas as operações de gravar ou salvar em dispositivos auxiliares). PERSISTêNCIA è um fator muito importante para o trabalho com Base de Dados, onde, por natureza ou por seu propòsito, a persistência dos dados deve transcender a programas especificos. Mas è uma caracteristica relacionada às capacidades de equipamentos e que nem toda linguagem orientada a objetos possui.

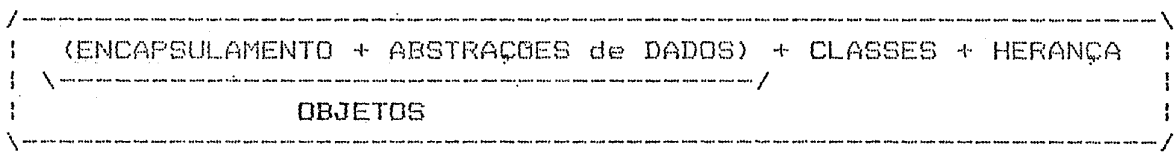
O ponto de como tratar a resolução de vinculos e referências ("BINDING") (5) serà visto na seção 3 (3.3) e na seção 5 (5.2) merecendo especial atenção por tratar-se de um tema bastante polêmico. E dificil determinar que uma modalidade fixa como "LATE BINDING" (que basicamente è a demora de resolução dos vinculos para tempo de execução) seja normativa de um estilo. Existem autores e linguagens como o Trellis/Owl [SCH86] que diferem muito na forma de tratamento da linguagem Smalltalk-80 [GOL83] considerada por alguns como padrão de ORIENTAÇàO de OBJETOS, que por outrp lado tem muitos pontos criticàveis ou em discussão. Estas divergências fazem optar pela exclusão do grupo de caracteristicas fundamentais.

A facilidade para que os objetos trabalhem em forma concorrente (4) è uma caracterìstica interessante e atè desejàvel em muitas situaçõe, mas que tambèm no consenso, sào poucas as linguagens que a possuem. A terminologia 'envio de mensagens' ou 'troca de mensagens' favorece a pensar que a concurrência seja uma caracterìstica primària, mas isto, na maioria dos casos, nào tem relação "nem prò nem contra" (como destaca Cox [COX86] referindo-se a Objetive-C). Wegner [WEG87] considera dentro do universo de linguagens ligadas a objetos, um grupo particular de linguagens que suportam OBJETOS e que estes OBJETOS tèm a qualidade de executar concorrentemente (estendem a autonomia dos objetos a uma autonomia no tempo) citando como alguns exemplos: ORIENT 84/K e ABCL/1.

Sobre os três primeiros pontos, parece nào existir dàvida na literatura e implementaçõe, nem discussào sobre sua importância como fatores determinantes para que uma linguagem seja ORIENTADA a OBJETOS.

Os conceitos de ENCAPSULAMENTO e ABSTRAÇOES de DADOS (1) foram colocados em um item sò pela estreita relação que tem no tema, sendo que constituem as idèias de modelagem elementares para OBJETOS. Serào considerados posteriormente (3.1).

Podemos chegar entào a estabelecer uma fòrmula bàsica ou de partida para rotular uma linguagem de "ORIENTADA a OBJETOS":



## 1.5 - Comparação entre o estilo de programação convencional (PROCEDIMENTAL) e o estilo "ORIENTADO a OBJETOS"

Os programas e sistemas bem estruturados desenvolvidos na base de linguagens como Pascal, Cobol, Fortran, C, etc, consistem de coleção de subprogramas (ou equivalentes), principalmente porque o SUBPROGRAMA, PROCEDIMENTO ou FUNÇÃO é a única unidade estrutural de construção [BOO86]. Essas linguagens então, estão encaminhadas a tècnicas de "descomposição funcional", que se concentram em ABSTRAÇOES ALGORITMICAS.

Mas Guttag observou que "a natureza das abstraçôes que podem ser convenientemente tratadas pelo uso de subrotinas è limitado". SUBROTINAS ou PROCEDIMENTOS são adequados para o tratamento de eventos, açôes ou operaçôes, são fundamentalmente ABSTRAÇOES de CONTROLE, mas não são um suporte adequado para ABSTRAÇOES de DADOS.

Os objetos são um termo mèdio que concentram e suportam esses dois tipos de abstraçôes: CONTROLE e DADOS.

Enquanto que no estilo de programação convencional procedimentos ativos são vinculados e atuam sobre dados passivos, no estilo "ORIENTADO a OBJETOS" foi mudada esta concepção, considerando ao OBJETO uma espècie de DADO ATIVO, solicita-se a um objeto que execute operaçôes por si mesmo.

"Não è apropriado nem possivel tratar os objetos da mesma forma como os programadores convencionais tratam o dado" [COX86]. Podemos aplicar uma função a um dado, mas não a um objeto.

O dado forma parte do objeto e è privativo dele, a única forma de ter acessar a esse dado è requerir ao objeto que faça isso, executando um de seus procedimentos ou mètodos na terminologia especifica.

# 2 - Panorama existente em LINGUAGENS ORIENTADAS a OBJETOS (LOOs)

## 2.1 - O "pout-pourri" pode ser classificado

Como existe uma notàvel variedade de linguagens que estão associadas ao conceito de OBJETO e que em maior ou menor medida podem se encaixar na "ORIENTAÇàO a OBJETOS", dando lugar em muitos casos a dûvidas ou discrepâncias de critèrios, foi considerado de real utilidade um trabalho de classificação dessas linguagens desenvolvido por Peter Wegner [WEG87] que serà sintetizado a seguir. Wegner demonstra uma aplicação auto referencial da metodologia orientada a objetos, isto vem resultar uma espècie de exercìcio da filosofìa de desenvolvimento mencionada na introdução (1.1).

Em primeiro lugar, e como mais alto nivel de generalidade, uma linguagem chama-se "BASEADA em OBJETOS se suporta o conceito de objeto como caracterìstica ou traço natural. Isto define uma classe de linguagens que incluem numerosas instâncias: Ada [DOD80], MODULA2 [WIR82], CLU [LIS81], Smalltalk [GOL83], Simula [DAH68], Objective-C [COX86], C++, etc, mas não incluem linguagens como Pascal, Algol ou Fortran.

Os objetos em Ada realizam-se atravès de "packages" ou pacotes, os objetos em Mòdula2 chamam-se de "mòdulos", os objetos em CLU sào instâncias de "clusters", em todos estes casos existem mecanismos para abstração de dados.

Suportar objetos è uma condição necessària, mas não suficiente para ser "ORIENTADA a OBJETOS". Linguagens orientadas a objetos devem suportar adicionalmente: CLASSES e HERANÇA de CLASSES.

Uma linguagem "BASEADA em OBJETOS" se diz "ORIENTADA a OBJETOS" se seus objetos pertencem a classes e podem definir-se incrementalmente HIERARQUIAS de CLASSES usando mecanismos de HERANÇA. Reduzindo a fòrmula dada em 1.4.4 pela substituição do grupo (encapsulamento + abstração de dados) por OBJETOS, temos:

```
/-------------------------------------------------------\
| ORIENTAÇàO a OBJETOS = OBJETOS + CLASSES + HERANÇA     |
\-------------------------------------------------------/
```

CLASSES agrupam objetos por suas caracterìsticas comuns. Por sua vez, a HERANÇA serve para agrupar ou ordenar as classes por seu comportamento similar.

A classe de linguagens "ORIENTADAS a OBJETOS" è mais estreita do que a classe "BASEADA em OBJETOS", excluindo linguagens como Ada, Mòdula2 ou CLU e incluindo Smalltalk, Trellis/Owl, Objective-C.

De acordo com a definição dada, Ada è uma linguagem "baseada em objetos", mas não uma linguagem "orientada a objetos" porque seus objetos ("packages") não tem conceito de classes.

O conceito de classes tem uma versão limpa e elegante nos TIPOS ABSTRATOS de DADOS PARAMETRIZADOS. Este è o mecanismo usado por CLU em seus "clusters", que servem como padrões (ou "templates" no original) para criar instâncias e permitir que suas instâncias sejam objetos "de primeira" no sentido que podem ser:

* atribuídos a variàveis

* passados como parâmetros

* ser componentes de estruturas

Mas CLU não tem mecanismo de HERANÇA que permita definir "relações hierárquicas entre CLUSTERS".

Seguindo com a tarefa de classificar convenientemente as linguagens do espectrum, è caracterizado o grupo de linguagens "BASEADAS em CLASSES": são linguagens "BASEADAS em OBJETOS" nas quais todo objeto pertence a uma classe.

As linguagens "baseadas em classes" formam um subconjunto das linguagens "baseadas em objetos", enquanto que as linguagens "orientadas a objetos" formam por sua vez um subconjunto das linguagens "baseadas em classes".

Módula2 è um exemplo de linguagem "baseada em objetos", mas não è uma linguagem "baseada em classes". Enquanto que CLU è um exemplo de linguagem "baseada em classes" que não è uma linguagem "orientada a objetos".

No seguinte esquema ilustra-se esta hierarquia de classes de linguagens:

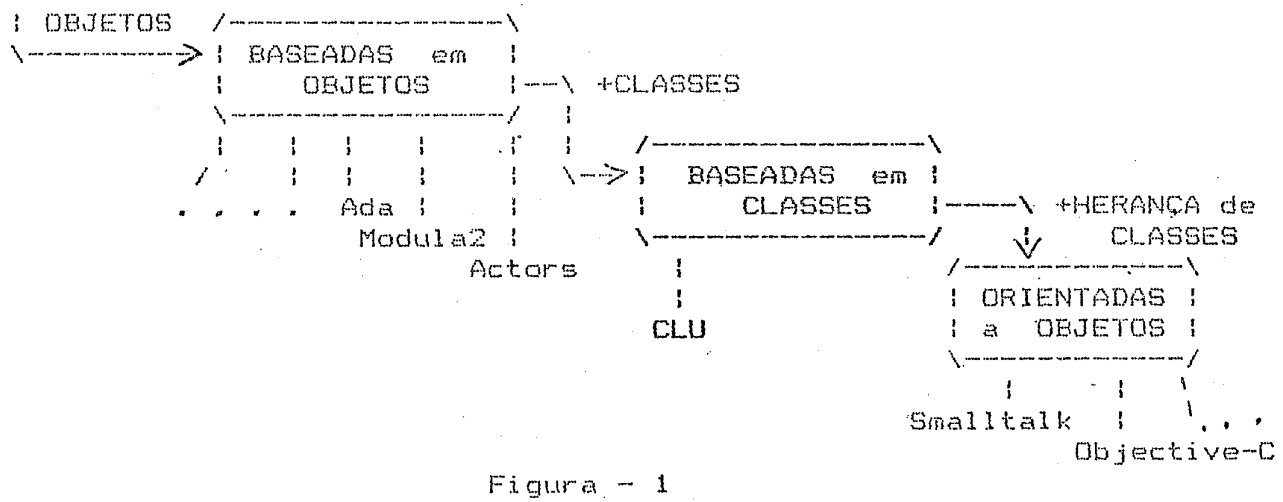



Figura - 1

Podemos pensar que as linguagens "baseadas em classes" herdam atributos das linguagens "baseadas em objetos", e as linguagens "orientadas a objetos" herdam atributos ou propriedades de ambas ("baseadas em objetos" e "baseadas em classes").

Esta auto-aplicação do desenvolvimento orientado a objetos ás linguagens do estilo é uma mostra do poder que tem na CLASSIFICAÇãO e ORGANIZAÇãO GERAL do CONHECIMENTO, outorgando uma visão substancial dentro de um domínio particular.

Neste exemplo, classificam-se linguagens em uma hierarquia impondo progressivamente restrições mais fortes ás caracteristicas que as linguagens possuem. Esse tipo de classificação é um princípio chave da HERANÇA nos sistemas ORIENTADOS a OBJETOS.

Podemos extender ainda mais esta classificação considerando uma outra restrição: a "TIPAGEM FORTE".

Uma linguagem è FORTEMENTE TIPADA se è testada a compatibilidade de tipos de todas as expressões que representam valores.

Linguagens "ORIENTADAS a OBJETOS com TIPAGEM FORTE" são uma classe mais restrita da que até agora consideramos como ORIENTADA a OBJETOS. Esta classe inclui a linguagem Trewllis/Owl. [SCH86]

Também a partir das linguagens "baseadas em objetos" distinguimos um grupo, de notavel relevância, que são as que suportam OBJETOS e que estes objetos podem trabalhar em forma concorrente, chamaremos a elas de "BASEADAS em OBJETOS CONCORRENTES" (exemplos: ABCL/1, ORIENT84/k).

Esta última classe dá lugar a distinguir um subconjunto com caracteristicas mais particulares ainda, que são as chamadas "LINGUAGENS de ATORES" [AGH86]. O modelo de atores foi concebido com o objetivo de explorar paralelismo em larga escala, portanto apoia-se na computação por mensagens. Existem ainda matizes diferentes dentro das linguagens de atores que vão das PRIMITIVAS puramente baseadas em semântica de passagem de mensagens a linguagens de mais alto nivel que incorporam algum tipo de HERANÇA para organização conceitual, mas não há uma noção explicita de classes no modelo de atores. [TAD88]

## 2.2 - Linguagens orientadas a objetos "PURAS" e "HIBRIDAS"

Em função das caracteristicas de uma linguagem desde suas origens foram estabelecidos dois grupos:

LOOs "PURAS"

LOOS "HIBRIDAS"

- LOOs "PURAS": são as que suportam os traços marcantes da ORIENTAÇàO a OBJETOS (OBJETOS, CLASSES e HERANÇA) desde seu nascimento, ou melhor ainda, desde sua concepção.

Precursora deste grupo è a linguagem Simula/67 [DAH68] desenhada por uma equipe de cientistas noruegueses.

Simula/67 surge com o objetivo de ser uma ferramenta para simulação, nesses anos nem se falava em ORIENTAÇàO a OBJETOS, mas è ela quem introduz nas linguagens de programação o conceito de classes. Seus objetos são INSTANCIAS de CLASSES e suporta HERANÇA.

A mais amplamente difundida das LOOs è sem dûvida Smalltalk. Desenvolvida por um grupo da XEROX PARC, durante a dècada dos 70, pegou elementos de Simula e foi evoluindo entorno do conceito e forma de trabalho dos OBJETOS e criando tambèm um ambiente especifico.

Smalltalk-80 [GOL83] è adotada por vàrios autores e especialistas na àrea, como padrão das LOOs, mas tem pontos severamente criticados como violações ao encapsulamento e controvèrsias não menos importantes, sobre os beneficios ou contras que acarretam o "LATE BINDING".

Entre as LOOs "PURAS" tambèm podemos distinguir Trellis/Owl [SCH86] da DEC, 1985.

- LOOs "HIBRIDAS": Estas linguagens vem da incorporação dos ingredientes pròprios da ORIENTAÇàO a OBJETOS a linguagens jà existentes e criadas sobre a base de propòsitos diferentes em alguns casos e com pontos coincidentes em outros.

As vezes vantagens de peso, acepitação ou "performance" de uma determinada linguagem são razões que favorecem à extensão para um outro paradigma.

Apareceram assim diversas liguagens "HIBRIDAS" como extensão de outras jà afiançadas no mercado, como de Pascal por exemplo: Object-Pascal; a partir de Lisp: Loops, a partir de C existem duas C++ e Objective-C.

## 3 - Conceitos importantes

### 3.1 - ABSTRAÇOES de DADOS e ENCAPSULAMENTO

ABSTRAÇOES de DADOS e ENCAPSULAMENTO são conceitos com acepções diferentes, mas que estão relacionados na idèia de TIPOS ABSTRATOS de DADOS que constitui a fundação para OBJETOS.

ABSTRAIR è o processo de identificar as qualidades e propriedades importantes do fenômeno sendo modelado, neste caso, os DADOS.

Nas primeiras linguagens que elaboram algum conceito de abstração entorno de dados, isto se dà na forma de suas similaridades estruturais (Exemplos: tipos definidos pelo usuàrio como vetores de caracteres ou registros). Mas foi reconhecida posteriormente a importância de classificar abstrações de dados nem sò pelas suas similaridades de estruturas de representação, mas sim por seu comportamento.

Exitem "dois nìveis de abstração: no nivel pròximo ao problema trabalhamos com uma abstração que desempenha para os dados, de certa forma, papel anàlogo ao que procedimentos (rotinas) desempenham para o fluxo de controle;ao nivel de uso da abstração interessa "o que", ao nivel de implementação interesa "o como" (Guttag)". [VEL86]

#### 3.1.1 - Diferenças entre ABSTRAÇOES de DADOS e ENCAPSULAMENTO

ENCAPSULAMENTO tem a ver com a forma de definir os detalhes de um objeto e ABSTRAÇOES de DADOS tem a ver com a produção de um modelo para gerar e manipular objetos.

Os objetos são mòdulos de informação, como jà foi mencionado em seções anteriores, que agrupam dados e todos os procedimentos para manipular esses dados.

De acordo ao PRINCIPIO de INVISIBILIDADE de Parnas, um mòdulo deve esconder informação no seguinte sentido:

(a) ocultando detalhes de representação (que devem ser invisìveis fora dele);

(b) fornecendo sò informação atravès de suas funções de acesso.

Veloso em [VEL86] distingue rigorosamente dois aspectos deste tema:

(1) ENCAPSULAMENTO, que è a representação dos objetos e implementação das operações reunidas numa ùnica região de programa;

(2) PROTEÇàO, que è a restrição colocada pelos objetos de sò serem manipulados, por meio das operações de um elenco prè-estabelecido, fazendo com que a representação fique invisivel e protegida contra manipulações indisciplinadas.

Mas è comum achar as idèias de ENCAPSULAMENTO e PROTEÇàO tratadas por um termo sò: ENCAPSULAMENTO.

Para Snyder [SNY86], "ENCAPSULAMENTO" è a tècnica para minimizar interdependências entre mòdulos escritos separadamente definindo interfases externas. A interfase externa de um mòdulo serve como uma espècie de contrato entre o mòdulo e seu usuàrio. ENCAPSULAMENTO è uma ferramenta poderosa, pois se os usuàrios sò dependem da interface externa, o mòdulo pode ser reimplementado sem afetar estes usuàrios.

ENCAPSULAMENTO assegura aos desenhadores compatibilidade nas mudanças, o qual facilita a evolução e manutenção de programas beneficios muito estimados em grandes sistemas.

Para maximizar as vantagens do ENCAPSULAMENTO, deve minimizar-se a exposição dos detalhes de implementação nas interfaces externas: destàcar a importância da PROTEÇàO.

Por tràs da idèia de ABSTRAÇàO de DADOS està a de definir um padrão para objetos, da mesma forma em que tipos definidos pelo usuàrio definem padrões de estruturas de dados em linguagens como Pascal, por exemplo.

Objetos pertencem a um padrão em particular e podem herdar todos os atributos definidos por este padrão.

ABSTRAÇOES de DADOS extendem de certa forma o conceito de ENCAPSULAMENTO de DADOS da mesma forma que tipos definidos pelo usuàrio extendem o conceito de tipos embutidos (inteiros, reais, caracter, etc).

Outra forma de ver ABSTRAÇOES de DADOS è como uma extensão de "TIPOS" em um amplo sentido.

ABSTRAÇOES de DADOS são aplicados no conceito de classe e os objetos respondem como INSTANCIAS ENCAPSULADAS de tais abstrações.

Em geral, todas as LINGUAGENS ORIENTADAS a OBJETOS, suportam ABSTRAÇàO de DADOS vinculada a ENCAPSULAMENTO, sò que nem todas respeitam rigorosamente o aspecto de PROTEÇàO da CAPSULA.

### 3.1.2 - TIPOS ABSTRATOS de DADOS

TIPO ABSTRATO de DADOS è uma manifestação de ABSTRAÇàO de DADOS que permite modelar grupos de elementos similares em forma encapsulada.

Considerando o estrito sentido de TIPO ABSTRATO de DADO ele è fundamentalmente uma ESPECIFICAÇàO ALGEBRICA, que consiste em duas partes:

TAD
- uma especificação de nomes de tipo, nomes de operações e aridade de funções;
- uma especificação de operações que pode dar-se por um conjunto de AXIOMAS ou em forma ALGORITMICA.

O atributo "abstrato" è justificado pelo fato de concentra-se no aspecto de comportamento dos elementos e abstrair-se dos detalhes de implementação.

Houve uma extensão na aplicação do termo TIPO ABSTRATO de DADOS em enorme parte da literatura especifica e pode ser encontrada para referir-se a CAPSULAS que descrevem representação e operações sobre uma coleção de dados.

Os objetos foram assentados sobre a base de ABSTRAÇOES de DADOS que permitem suportar duas visões:

* VISàO EXTERNA

* VISàO INTERNA

A visão externa do objeto serve para capturar o comportamento (abstrato) do objeto, dando què coisas podem ser feitas sobre esses objetos ou què coisas esses objetos podem fazer.

A visão interna indica o como esse comportamento està ou serà implementado. [BOOB6]

Quando se desenha um sistema, o primeiro passo que interessa em nosso processo de desenvolvimento è a visão externa, importando- nos o que se pode fazer com determinados elementos.

### 3.1.3 - Conclusões sobre estes conceitos para as LOOs

Para gozar dos benefícios do ENCAPSULAMENTO em todos os sentidos é desejável, manter um mecanismo de PROTEÇÃO rigoroso.

Na modelagem de ABSTRAÇÕES de DADOS é bom ter em conta os princípios de TIPOS ABSTRATOS de DADOS tendendo a distinguir claramente um módulo de especificações.

E interessante manter na cápsula de definições duas partes bem diferenciáveis segundo a proposta apresentada em [VEL86]:

* MODULO ABSTRATO (ou PROGRAMA ABSTRATO:PA) na forma puramente de especificação;
* MODULO de IMPLEMENTAÇÃO que trata sobre representação de objetos e implementação de operações.

Essa fatoração tráz consequências sobre: LEGIBILIDADE, MODIFICABILIDADE e PORTABILIDADE desejáveis.

## 3.2 - Tipos - Hierarquia de Tipos - Herança

### 3.2.1 - Tipos

O conceito de TIPOS é um forte princípio de organização na programação orientada a objetos. [HAL87]

Com uma visão mais global é importante rever o que significa proporcionar a uma linguagem um SISTEMA de TIPOS.

"Um sistema de tipos é um conjunto de regras que permitem associar um tipo com toda expressão ou subexpressão semânticamente significativa dentro de uma linguagem de programação" [WEG86]. Quer dizer que um SISTEMA de TIPOS dita normas que regulamentam a interconexão ou relacionamento de objetos dentro de um sistema.

Para analisar que propriedades deve ter um sistema de tipos como totalidade ou um tipo em particular é necessário ter em conta diversas ópticas do assunto.

Desde o ponto de vista do sistema de programação e de questões de segurança, TIPOS proporcionam uma cobertura de proteção, que oculta a representação subjacente e que restringe a modalidade operacional entre objetos. Quando o SISTEMA de TIPOS é violado, a representação do dado fica exposta a manipulações não desejadas e pode acarretar resultados desastrosos.

Enfocando a questão da VERIFICAÇÃO, os TIPOS impõem restrições que forçam á CORRETUDE de programas.

TIPOS tem equivalência conceitual com CLASSES, portanto ás vezes se usam os termos indistintamente. Desde a perspectiva de organização e estruturação de domínios de aplicação os TIPOS facilitam uma base natural para modularização , porque são comumente usados para modelar entidades do universo do problema. Cada porção de código, nas LOOs normalmente, esta associado a um TIPO ("CLASSE") em particular. A decisão do programador de que tipos usar ou definir afeta á estrutura geral e tem importantes consequências para a evolução do programa.

Observando, em particular, o áspecto de evolução de um sistema orientado a objetos, os TIPOS (CLASSES) dão especificações de comportamento que podem ser modificadas incrementalmente, formando novos TIPOS (CLASSES), idéia base das HIERARQUIAS de TIPOS que constituem o suporte do mecanismo de HERANÇA.

### 3.2.2 - Hierarquias de tipos como base de HERANÇA

A noção de TIPOS se encontra enriquecida na programação "orientada a objetos", pois além de estar ligada a ABSTRAÇÃO de DADOS, podem expressar-se interessantes relações entre TIPOS ("CLASSES") capturando similitudes entre coleções ou agrupamentos de tipos. Isto permite formar, numa versão simples, HIERARQUIAS de CLASSES (TIPOS) que expressam tipos novos como descendentes de outros predefinidos.

A nivel conceitual, pode existir um critério de estabelecer níveis de restrições, considerando tipos novos como composição de tipos menos restritos em comportamento. Determina-se assim uma espécie de HIERARQUIAS de RESTRIÇÕES.

As HIERARQUIAS de TIPOS que suportam a HERANÇA sustentam numa mesma organização dois aspectos: GENERALIZAÇÃO e ESPECIALIZAÇÃO.

A HERANÇA fomenta a que os programadores criem novos tipos (classes) e em consequência objetos, que são especializações ou casos particulares de outros tipos (classes). Isto é conhecido como SUBTIPADO ("SUBTYPING" ou "SUBCLASSING") [HAL87] [PAS86]. O novo tipo (classe) é um subtipo (subclasse) do tipo já existente e este último por sua vez é um supertipo (superclasse) do novo.

Do outro lado da moeda, considera-se que o SUPERTIPO é uma fatoração de características comuns entre os SUBTIPOS em que esta particionado.

SUBTIPADO é a forma mais frequênte de hierarquias conceituais, geralmente estas hierarquias tentam refletir um modelo do mundo real com um esquema de CLASSIFICAÇãO. A idèia chave è que um programa pode ser construido, modelando primeiro os tipos, começando pelos conceitos mais gerais do domínio de aplicação e depois tratando os casos especiais através de subtipos especializados. Esta classificação de tipos proporciona uma metodologia de desenho geral baseada em refinamentos sucessivos por especialização, comparável com a popular metodologia de "decomposição por refinamentos sucessivos" implantada para programação estruturada, mais comumente aplicada a abstrações de controle.

Em alguns casos o SUPERTIPO è um tipo parcial, pode ter fatoração de caracteristicas comuns de seus subtipos, mas ele por si mesmo è uma descrição incompleta, incapaz de descrever um objeto útil. Os subtipos incrementam comportamento transformando o tipo parcial em algum tipo útil.

## 3.3 - Conceito de AMARRAÇãO

O conceito de amarração ("BINDING") è usado para indicar associação de atributos a entidades [GHE85]. Geralmente para cada entidade a informação sobre essa amarração è guardada em um DESCRITOR. Nas linguagens de programação o conceito de amarração define uma parte chave de sua semântica. Em geral o momento em que a AMARRAÇãO ocorre (TEMPO de AMARRAÇãO) define tal caracterlstica de ESTATICA ou de DINAMICA.

Considera-se AMARRAÇãO ESTATICA quando è estabelecida antes da execução do programa e DINAMICA, se è estabelecida em tempo de execução e pode ser mudada em função das regras da linguagem.

No caso das LOOs tem especial importância o tema da AMARRAÇãO de objetos a mètodos. No mundo de objetos que por natureza è bastante dinâmico è comum substituir esses adjetivos (DINAMICA ou "ESTATICA") por outros de AMARRAÇãO ANTECIPADA ("EARLY BINDING") ou AMARRAÇãO TARDIA ou ATRASADA ("LATE BINDING") que essencialmente indicam mesmos momentos. para realizar a associação que no caso anterior antes ou durante a execução respectivamente.

As linguagens ESTATICAMENTE TIPADAS (ver 5.1) (aquelas que realizam checagem de tipos em tempo de compilação) tem um "EARLY BINDING"em que os identificadores em aberto são amarrados na fase de "linkedição" e carga do compilador.

A amarração atrasada, "LATE BINDING", è como uma amarração sobre demanda, que pode dar maior flexibilidade, mas cobra um preço caro em tempo de execução. [TAD88]

O estilo frequênte nas LOOs, tomado quase como norma no início, è a amarração atrasada ("LATE/DYNAMIC BINDING"), usada por Smalltalk, Flavors,LOOPS e outras.

Argumenta-se este tipo de amarração para que as mensagens idênticas (com o mesmo nome) possam ser processadas distintamente para cada tipo de objeto para o qual a mensagem è enviada (ter em conta que nas linguagens mencionadas não existem declarações). Este aspecto è conhecido como **COMPORTAMENTO POLIMORFICO.**

Exemplo: Seja a seguinte mensagem,

GRUPO . pròximo,

que poderia indicar que se solicita ao objeto GRUPO executar o mètodo "pròximo". Esse mètodo "pròximo" pode não ser ùnico no sistema de classes e existir vàrios "pròximo", correspondendo a classes diferentes (exemplo, PRÒXIMO da classe FILA, tem o significado de entregar o pròximo elemento de uma fila, jà pròximo da classe PILHA; tem tambèm esse significado, mas outro comportamento).

Uma forma de suportar COMPORTAMENTO POLIMÒRFICO com um "EARLY BINDING" è chamado "SOBRECARREGAMENTO de OPERADOR", adotado por Ada e CLU, que não deixam solta a decisão de achar a associação para o momento de execução, mas sim que permitem que um operador seja redefinido. De todos os significados que podem ter um operador de comportamento polimòrficò vai valer aquele associado ao tipo (ou classe) do objeto demandante.

## 3.4 - Alaumas consideraçòes sobre ADMINISTRAÇÀO de MEMORIA

E comum nas LOOs uma administração de memòria baseada em alocação de objetos no HEAP e uso de endereços ou variàveis-ponteiro. Existe toda uma estrutura de PONTEIROS montada para resolver as referências. Objetos são criados explicitamente e cobram vida quando è executada a operação especifica de criação (do tipo NEW) portanto alocados dinâmicamente e posteriormente referenciados via ponteiros. Em casos como Smalltalk, esses apontadores (OP "object pointers") passam por uma zona intermediària de descrição sucinta (TABELA de OBJETOS) antes de alcançar o endereço no HEAP, onde o objeto està realmente alocado.

Para detalhes do mecanismo e sistema completo de alocação e tratamento de informação de classes e mètodos ver [GOL83] [PAS86] [TAD88].

Em C++, porèm, existe uma mistura de tipos de alocação. Novos objetos podem ser alocados dinâmicamente e depois ser referênciados por seu endereço ou tambèm podem ser alocados estàticamente e depois ser referênciados por nome. Conta com operação de CRIAÇàO e DESTRUIÇàO de objetos.

Linguagens como Smalltalk, Loops e outras baseadas em Lisp incluem um mòdulo de COLETA de LIXO ou "GARBAGE COLLECTION" AUTOMATICO. Quando uma nova instância de uma classe è criada (um objeto), è ativada a requisição de um espaço de memòria. Serà realizada entàO uma pesquisa para tentar atender à demanda; se nàO houver bloco de memòria do tamanho procurado, serà disparada uma COMPACTAÇàO de MEMORIA. Na especificação do Smalltalk-80 a COLETA de LIXO està acompanhada por um sistema de CONTAGEM de REFERENCIAS.

Um sistema de CONTAGEM de REFERENCIAS mantèm a pista do nûmero de ponteiros para cada objeto no sistema. Quando o contador de um objeto no sistema chega a zero (quer dizer que jà nàO tem referência nenhuma) o objeto è deletado. Mas este sistema leva muito trabalho para manter atualizados os contadores e tràs um efeito de deterioração da linguagem.

C++ tenta evitar o uso de COLETA de LIXO e de fato nàO tem esta ferramenta, mas reconhece que quando se tem objetos multiplamente referenciados, via pointers, è necessàrio contar com esse apoio. Em C++ existe um mecanismo que nàO deixa formar referências mûltiplas, copiando objetos cada vez que eles sàO referenciados e destruindo objetos cada vez que saem do escopo.

Toda vez que um objeto (jà criado) è referenciado em um outro ambiente (por exemplo passando como argumento de algum mètodo) gera e aloca uma nova còpia, exatamente como uma "CHAMADA POR VALOR". Mas isso è fatível para objetos pequenos, caso contràrio o OVER HEAD pode-se tornar intolerável.

# 4 - HERANÇA

## 4.1 - Noção de Herança

HERANÇA è uma forma de modificar comportamento em forma incremental, adaptando-se à evolução do sistema. E um mecanismo que permite que um tipo (classe) possa ser definido com base em tipos ou classes predefinidas, incorporando estruturas e operações especificados para o outro tipo (classe).

O termo è aplicado então no sentido de que uma classe (tipo) "HERDA" propriedades de uma ou varias classes, conforme o tipo de HERANÇA. "Em cima desta HERANÇA a classe pode definir novas propriedades, aumentando as estruturas de dados, criando novos mètodos ou modificando algumas operações herdadas". [IER87]

## 4.2 - Herança e reusabilidade

"O termo "REUSAR" tem o significado de poder usar um produto em uma situação diferente da original para a qual foi criado, sem o esforço que demanda a criação de um novo produto". Peter Freeman.

Em reusabilidade de software achamos vàrias formas e acepções. Reaproveitamento de còdigo è uma forma de reusabilidade de software. Os especialistas no tema reconhecem a necessidade de organizar grandes inventàrios de software para que o còdigo reusàvel seja fàcil de localizar e intercambiar.

O sucesso da reusabilidade de fragmentos de còdigo è devido, em parte, a capacidade que estes tenham de ENCAPSULAR FUNÇOES de DOMINIO ESPECIFICO (estes fragmentos seriam capsulas de propòsito especìfico) e a tècnica geral seria criar bibliotecas com estes componentes classificados. Para o problema de selecionar entre um conjunto de componentes potencialmente reusàveis similares, deve ser realizado uma avaliação das caracteristicas comuns.

Na programação ORIENTADA a OBJETOS, TIPOS (CLASSES) ajudam a organizar o dominio do problema, reconhecendo caracteristicas comuns e criando padrões de representação e comportamento. A HERANÇA proporciona a forma de vincular e reaproveitar as caracteristicas capturadas por estes tipos em dois niveis: especificação e implementação.

## 4.3 - Herança simples

**HERANÇA SIMPLES** está montada sobre uma organização HIERARQUICA de TIPOS (ou CLASSES) e cada classe tem um único supertipo ou tipo progenitor do qual pode "herdar" propriedades.

Uma boa parte deste conceito foi explicado no item 3.2.2 ("Hierarquias de Tipos como base de Herança").

Freqüentemente as características herdadas podem ser modificadas (ultrapassadas por redefinição). Métodos do supertipo podem ser redefinidos com o mesmo nome (mesmo seletor), em outras palavras podem ser SOBRECARREGADO (ver item 3.3).

Diferentes linguagens apresentam uma variedade de graus de controle sobre o que pode ser herdado e o que pode ser ultrapassado. Por exemplo, em Smalltalk-80, as operações definidas em um supertipo sempre são herdadas e podem ser redefinidas em um subtipo.

Comparativamente em Trellis/Owl operações e campos de representação de memória podem ser declarados como PUBLICOS e PRIVATIVOS. Declarações de elementos PRIVATIVOS são visíveis dentro, mas não fora, e um subtipo não pode usar operações ou campos privativos do supertipo.

Snyder [SNY86] apresenta uma discussão interessante sobre considerações de ENCAPSULAMENTO e HERANÇA, destacando que podem entrar em conflito se não são observados pontos de segurança e lembrando que a HIERARQUIA de TIPOS não deve ser afetada por câmbios em um determinado TIPO ,ou CLASSE, isto é, essencialmente para suportar a evolução de grandes sistemas e dados de longa vida, como se propõe a programação ORIENTADA a OBJETOS.

## 4.4 - Herança múltipla

**HERANÇA MULTIPLA** significa que um tipo (classe) pode ter mais de um progenitor. De acordo com o isso, a relação supertipos - subtipos não fica restringida á forma de ARVORE (como na HERANÇA SIMPLES) senão que é tratada por um **GRAFO DIRIGIDO ACICLICO.**

Neste grafo, que chamaremos GRAFO de HERANÇA , podemos distinguir um tipo que será o tipo-raiz, cada tipo (classe) é um nó do grafo e existem arcos desde cada tipo a seus progenitores.

Exemplo:

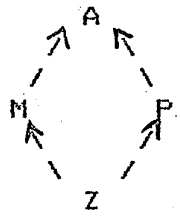


Figura - 2

No exemplo, Z herda dos tipos M e P, e os tipos M e P por sua vez herdam caracteristicas do tipo A.

Apresenta-se o seguinte problema: o que acontece se um tipo herda operações do mesmo nome de dois ou mais progenitores ? Pode colocar-se a restrição de que todos os nomes de mètodos de supertipos devem ser diferentes, ou bem è estabelecida uma ordem de prioridades sobre os supertipos para resolver estes conflitos.

Existem dois tipos de estratègias para administrar HERANÇA MULTIPLA. A primeira tenta tratar diretamente com o grafo de herança dirigido. A segunda estratègia transforma o grafo em uma cadeia linear e depois trabalha com essa cadeia usando regras de HERANÇA SIMPLES.

## 4.5 - Herança multidimensional

Hailpern e Nguyem em [HAI87] definem um outro estilo de HERANÇA que chamam de HERANÇA MULTIDIMENSIONAL. Neste caso a herança depende do contexto do requerimento, quer dizer do objeto que requer o mètodo e do mètodo por si mesmo.

Por exemplo, considere um objeto que administra uma base de dados de anàlises clinicas de pacientes de um hospital. Se um paciente solicita os resultados de uma certa anàlise (para levar fora do hospital, por exemplo) os resultados serão informados de uma forma e se um mèdico do hospital solicita os resultados da anàlise, estes serão informados de uma outra forma, provavelmente com certas conclusões de guia.

## 4.6 - Outras formas de Herança

Até agora vimos uma acepção de HERANÇA com o sentido de compartilhar recursos entre classes. Mas o termo, às vezes é usado vagamente para indicar outras formas de compartilhar recursos.

Existe uma noção mais geral, independente do conceito de classe, para compartilhar recursos dinâmicamente que è a DELEGAÇàO.

DELEGAÇàO è um mecanismo que permite que os objetos deleguem a responsabilidade de realizar uma determinada operação em um ou mais ancestrais "DESIGNADOS" para tal propòsito.

Uma característica chave è que a auto-referência em um ancestral, denota o objeto delegante, permitindo assim ao ancestral ser parte da entidade extendida do objeto delegante.

A "AMARRAÇàO DINAMICA" ("DYNAMIC BINDING") da auto-referência dà lugar á realização do compartilhamento e reusabilidade, permitindo que recursos de um ancestral sejam parte da entidade extendida de diferentes objetos delegantes em diversos pontos de execução.

A idèia central deste tema è a INTERNALIZAÇàO de OPERAÇOES DELEGADAS de maneira tal que possam ser tratadas como parte da prôpria extensão.

A DELEGAÇàO è só uma forma de compartilhar recursos, que permite que esses recursos compartilhados sejam vistos como pertencentes á entidade a favor da qual estão sendo executados. Este efeito se consegue com a "AMARRAÇàO DINAMICA" prôpria de cada entidade.

Desta forma, uma operação dada pode pertencer a diferentes entidades em diferentes momentos de execução. Isto pode ser uma vantagem em aspectos de flexibilidade, mas por outro lado è uma sèria violação aos conceitos de encapsulamento e proteção.

DELEGAÇàO pode dar-se associada a comportamento de PROTOTIPOS. Lieberman [LIE86] apresenta um modelo involucrando PROTOTIPOS e DELEGAÇàO.

E importante ter em conta este conceito de PROTOTIPOS, tambèm muito nomeado, para saber em que se diferencia do conceito de CLASSES.

PROTOTIPO è uma forma de modelar conhecimento que representa o comportamento "default" de um determinado conceito.

Um objeto pode usar a informação armazenada no PROTOTIPO ou pode especificar como ele difere do PROTOTIPO.

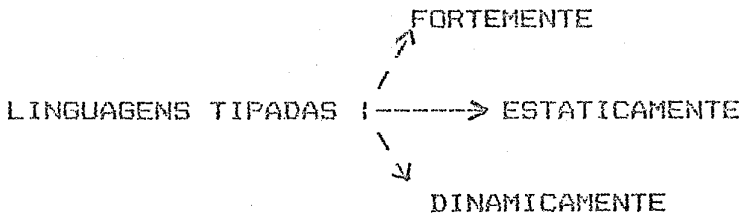
5 - Polêmicas

## 5.1 - Discussões relativas a TIPAGEM

Os tipos impõem restrições sintáticas e semânticas sobre expressões de forma tal que operandos e operadores que compõem estas expressões sejam compatíveis.

As linguagens tipadas (que tem um sistema de tipos) contêm normas para a integração de um objeto (instâncias de tipos) que servem para a própria proteção dos objetos.

De acordo com a forma em que è realizada a CHECAGEM de TIPOS, as linguagens de programação podem enquadrar-se nas seguintes categorias:




Linguagens nas quais è checada a consistência de todas as expressões são chamadas FORTEMENTE TIPADAS. [CAR85]

As linguagens de programação nas quais o tipo de cada expressão pode ser determinado por anàlise estàtica do programa em tempo de compilação são ditas ESTATICAMENTE TIPADAS.

TIPAGEM ESTATICA è uma propriedade útil porque prevê erros e economiza tarefas de execução, mas o requerimento de que todas as variàveis ou expressões sejam amarradas a um determinado tipo em tempo de compilação è ás vezes restrito demais. No caso das LINGUAGENS FORTEMENTE TIPADAS a condição è mais dèbil, sendo que dependendo da situação o tipo pode ser estaticamente desconhecido, resolvendo o problema com a introdução de alguma checagem de tipo em tempo de execução.

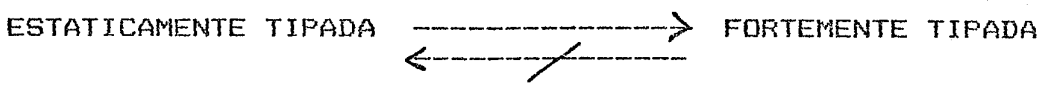
Linguagens com TIPAGEM DINAMICA pura, permitem que um identificador seja usado para referenciar objetos de qualquer classe ou tipo e pode passar a referenciar objetos de qualquer outra classe ou tipo a qualquer momento. Na pràtica, algumas linguagens DINAMICAMENTE TIPADAS checam, em tempo de execução, os tipos de toda expressão, quer dizer que cumprem com a condição de serem também FORTEMENTE TIPADAS, mas também achamos as que sò fazem amarração de tipos em tempo de execução com checagem não rigorosa.

O PROTOTIPO è tambèm uma instância, sò que è uma instância que se toma como ponto de referência, não existe uma ABSTRAÇàO GERAL que indique um padrão de comportamento como em TIPOS ou CLASSES.

Objetos podem delegar responsabilidades de realizar certas operações no protòtipo.

CLASSES e PROTOTIPOS são duas abordagens diferentes de representação de conhecimento, que correspondem a níveis de abstração diferentes, merecem ser analisadas e comparadas para ter em conta o que aplicar dependendo da situação. Enquanto que PROTOTIPOS enquadram mais na linha da chamada PROGRAMAÇàO EXPERIMENTAL, as CLASSES parecem ser mais apropriadas para uma programação STANDARD e de grande escala, onde a quantidade de objetos a tratar justifica criar uma ABSTRAÇàO de alto nível.

Observa-se que è verdade que toda linguagem ESTATICAMENTE TIPADA è tambèm FORTEMENTE TIPADA, mas a inversa nem sempre è verdadeira.




Considerando as diversas variações chega-se a um esquema com o seguinte aspecto:

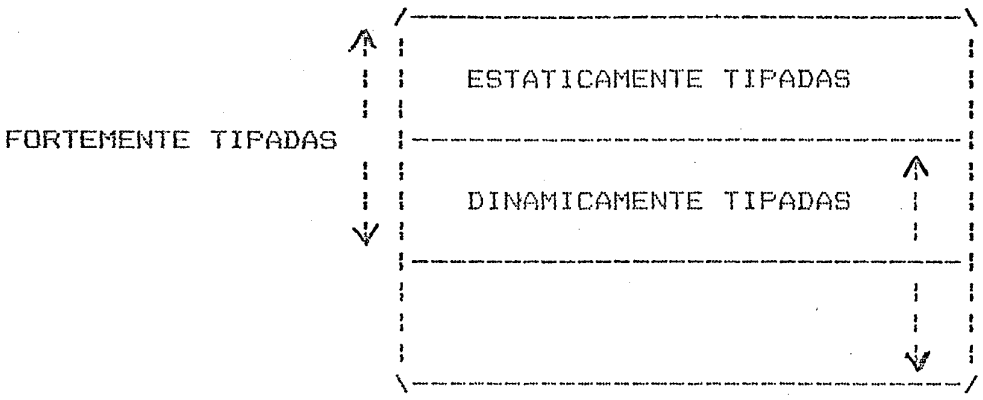



Figura - 3

Uma tendência desejável è assegurar uma linguagem FORTEMENTE TIPADA, adotando TIPAGEM ESTATICA, sempre que seja possivel.

## 5.2 - SOBRE TEMPOS DE AMARRAÇàO ("BINDING")

O que è melhor ter AMARRAÇàO ANTECIPADA ("EARLY") ou ATRASADA ("LATE")?

Segundo Cox [COX86], "EARLY BINDING" e "LATE BINDING" são fundamentalmente ferramentas diferentes para trabalhos diferentes. Este autor (um dos desenhadores do OBJECTIVE-C) afirma que para fazer uma estimação de qual o tipo de AMARRAÇàO mais beneficiosa, è conveniente considerar a relação existente entre TEMPO de AMARRAÇàO e FORÇA de ACOPLAMENTO.

COLEÇOES de ACOPLAMENTO FORTE ("TIGHTLY COUPLED") são por exemplo vetores, retangulos, meses, etc, onde o tipo de componente uma vez fixado è HOMOGENEO.

COLEÇàO de ACOPLAMENTO FRACO ("LOOSELY COUPLED") são grupos de elementos que podem ser caracterizados por um comportamento similar, mas tem enorme flexibilidade nas componentes, são agrupamentos de tipo HETEROGENEO como: ENVELOPE, PASTA, CAIXA POSTAL, AGENDA, etc. Uma CAIXA POSTAL, por exemplo, representa uma COLEÇàO de ACOPLAMENTO FRACO, pois não è possivel nem desejàvel estabelecer antecipadamente que itens ela contèm ( pode ser uma CARTA, um IMPRESSO, uma CONTA a PAGAR, um CATALOGO, etc).

As linguagens LIGADAS ESTATICAMENTE ou ANTECIPADAMENTE são apropriadas para construir coleções de ACOPLAMENTO FORTE, mas tem desvantagens e dificuldades para o tratamento apropriado de coleções de ACOPLAMENTO FRACO.

Quando linguagens com "EARLY BINDING" tratam coleções de ACOPLAMENTO FRACO como as mencionadas, vão ter de ser resolvidas associações basicamente dinâmicas, como decisões a serem tomadas só conhecendo o tipo de componente da coleção (que no caso è dado em execução) para o qual geralmente se inserem testes no pròprio còdigo de programa, da forma:

```
CASE tipo de componente
CARTA, XXXXX
CONTA a PAGAR, XXX  etc
```

Isto leva a discriminar os tipos de componentes da coleção nessa porção de còdigo. Uma consequência disso è a perda da extensibilidade, se quer extender o conteúdo da agrupação para mais um tipo de elemento, deve modificar esse pedaço do còdigo (por exemplo inserir um tipo CARTàO para a coleção CAIXA POSTAL) e também a limitação frente a possibilidade de reuso de còdigo por parte de outras coleções de elementos.

Por outra lado, se bem o "LATE BINDING"oferece a flexibilidade desejada para tratamento de coleções heterogêneas não previsiveis antecipadamente, è enormemente questionado o elevado custo de tempo de execução que este mecanismo consome, devido ao trabalho de seleção de mètodos em relação ao contexto.

Como conclusão, a conveniência sobre o mecanismo de AMARRAÇãO a adotar, fica em parte relacionado ao tipo de problemas mais freqüentes a tratar. Também nestes casos podemos distinguir entre uma PROGRAMAÇãO EXPERIMENTAL que se identifica com um "LATE BINDING" e uma PROGRAMAÇãO que chamaremos STANDARD, resoluvel e até aconselhavel por "performance" com um mecanismo de tipo "EARLY BINDING".

## 5.3 - Controvèrsias sobre HERANÇA

Assim como existe confusão em torno do termo "ORIENTAÇãO a OBJETOS", a HERANÇA que joga um papel de relevância neste contexto, não escapa a critèrios divergentes. Podemos considerar algumas discussões entorno á HERANÇA que vale a pena analisar como alternativas.

* Como vimos existe uma forma mais geral de HERANÇA independente do conceito de tipo ou classes, que è conhecida como **DELEGAÇãO**. A delegação como conceito desvinculado de classes realiza um compartilhamento dinâmico de recursos em uma **HIERARQUIA de INSTANCIAS** enquanto que HERANÇA realiza um compartilhamento de recursos em uma **HIERARQUIA de CLASSES (ou TIPOS)**. Uma manifestação da opção de DELEGAÇãO pode encontrar-se nas linguagens de ATORES ("ACTOR LANGUAGES" no original) que não tèm noção de classes, motivo pelo qual foram colocadas na categoria de "BASEADAS em OBJETOS" (ver seção 2.1). Uma discussão de DELEGAÇãO no contexto de ATORES è dada em [AGH86].

Este tema de HERANÇA versus DELEGAÇãO è anàlogo á discussão de:

- HERANÇA baseada em tipos (que reaproveita còdigo como consequência);
- Usar alguma forma de puro compartilhamento e reaproveitamento de còdigo.

Das situações planteadas acima podem achar-se casos bem polarizados ou bem uma combinação na qual embora exista conceito de tipos ou classes nem sempre è mantido o critèrio de "herdar" propriedades a nivel conceitual (como denotam as HIERARQUIAS de TIPOS) senão que dão maior importância a um reaproveitamento de còdigo ou representação a nivel de implementação.

* HERANÇA estrita versus HERANÇA não-estrita.

Herança estrita requer que os descendentes tenham comportamento compativel com os ancestrais ou progenitores, enquanto que a HERANÇA "não-estrita" permite que operações dos ancestrais sejam arbitrariamente redefinidas respondendo mais a uma idèia de SIMILARIDADE do que compatibilidade de comportamento.

Wegner e Zdonik dão a seguinte terminologia para diferenciar os conceitos:

- COMPATIBILIDADE de COMPORTAMENTO -----> relação"IS-a";

- SIMILARIDADE não estrita -----> relação "Like".

* HERANÇA SIMPLES versus HERANÇA MULTIPLA.

Desde que a perspectiva de que a HERANÇA MULTIPLA permite a um objeto herdar caracterìsticas de multiples ancestrais, brinda uma maior flexibilidade no aspecto de composições do que a HERANÇA SIMPLES que fica restrita a herdar de um ùnico ancestral.

* HERANÇA com INTERFACE ABSTRATA ou COMPARTILHAMENTO de CODIGO DIRETO.

Este tema da INTERFACE ABSTRATA ou EXTERNA tem a ver com mànter uma HIERARQUIA de ESPECIFICAÇOES.

Uma forma de herança via INTERFACES EXTERNAS è dada no seguinte exemplo: se temos a classe LISTA com as seguintes especificações de comportamento operacional:

LISTA = (VISITAR_PROXIMO_NO, LISTA_VAZIA)

operações que retornam o prôximo nò de uma lista e consulta se uma lista esta vazia ou não, respectivamente. Podemos definir uma subclasse de LISTA que seja LISTA DUPLA com o seguinte comportamento:

LISTA DUPLA = (VISITAR_PROXIMO_NO, VISITAR_NO_ANTERIOR,
LISTA_VAZIA)

O comportamento da subclasse è uma extensão da especificação da superclasse.

## 6 - Conclusões

A anàlise dos princìpios de PROGRAMAÇàO ORIENTADA a OBJETOS e o estudo de algumas polèmicas desatadas em relação a caracterìsticas das linguagens de programação alinhadas no tema, deixa ver que este estilo oferece um bom conjunto de conceitos e tècnicas desejàveis para a programação em grande escala.

"Linguagens orientadas a objetos podem favorecer significativamente a manutenção, extensibilidade e reusabilidade de software. Mas o uso apropriado e cuidadoso de tipos e herança è crìtico para o sucesso de programas orientados a objetos". Halbert e O'Brein em [HAL87]

No tema da modelagem de TIPOS (CLASSES) e HERANÇA, observa-se que se ESPECIFICAÇàO e IMPLEMENTAÇàO estão suficientemente defasadas, de maneira tal que as especificações possam ser ligadas a implementações de uma forma flexìvel, isto reflete modularidade, portabilidade e reaproveitamento do còdigo, mantendo uma organização conceitual.

Tambèm è interessante achar uma saìda que permita chegar a um termo mèdio na questão dos TEMPOS de AMARRAÇàO, flexibilizando a AMARRAÇàO ADIANTADA ("EARLY BINDING") para permitir a formação de coleções pouco acopladas ("LOOSELY COUPLED"), mas manter, sempre que for possìvel, a performance da AMARRAÇàO ADIANTADA.

Por ùltimo, è importante não permitir que a implementação simultânea de tècnicas que suportam conceitos fortes, faça com que estes entrem em conflito, que os benefìcios que pode acarretar um, destruam os princìpios do outro, como pode acontecer com HERANÇA E ENCAPSULAMENTO.

7 - Referências Bibliográficas:


[AGH86] - Agha Gul
"An overview of Actor languages", Proceedings of the Object Oriented Workshop (Yorktown Heights), ACM, pag.58-67, Jun 1986.

[AND87] - Andrews Timothy e Harris Craig
"Combining Language and Database Advances in an Object-Oriented Development Environment", OOPSLA 87 Conference Proceedings.

[BOO86] - Booch Grady
"Object-oriented Development", IEEE Transactions on Software Engineering, Vol.12, N2, pag.211-221, Feb 1986.

[CAR85] - Cardelli, L. e WEGNER, P.
"On understanding Types, Data Abstraction and Polimorphysm", ACM Computing Surveys, Vol17, N4, pag 471-522, Dez 1985.

[COH84] - Cohen A. Toni.
"Data Abstraction, data encapsulation and object oriented programming", ACM Sigplan Notices, Vol.19, N1, Jan 1984.

[COP84] - Copeland George e Maier David.
"Making Smalltàlk a Database System", Sigmod Record, Vol.14, N12, pag.316-324, 1984.

[COX86] - Cox, Brad.
"Object Oriented Programming an evolution approach", Addison Wesley, 1986.

[DAH68] - Dahl O.J., Myrhaug B., Nygaard K.
"Simula 67. Common Base Language", Norwengian Computer Center, Oslo, 1968.

[DOD80] - Ada Referenca Manual.
U.S.Department of Defense, Jul 1980

[DUF86] - Duff Charles.
"Designing an efficient language", Byte, pag.211-224, Ago 1986.

[GHE82] - Ghezzi, C. e Jazayeri, M.
"Programming Language Concepts", John Wiley & Sons, 1982.

[GOL83] - Goldberg Adele e Robson, D.
"Smalltalk 80. The language and its implementation", Addison Wesley, 1983.

[HAI87] - Hailpern Brent e Van Nguyen.
"A Model for Object-based Inheritance", Technical Report, IBM Watson Reseach Center, New York, Abr 1987.

[HAL87] - Halbert Daniel e O'Brien Patrick.
"Using Types and Inferitance in Object-Oriented Programming", IEEE Software, Vol.5, Set 1987.

[IER87] - Ierusalimschy Roberto.
"Abstração de Dados em linguagens de programação: Tipos e objetos", Monografias em Ciência de Computação PUC/RJ, Dez 1987.

[KAE86] - Kaefler Ted e Patterson Dave.
"A Small Taste of Smalltalk", Byte, pag.145-159, Ago 1986.

[LAL85] - Lalonde Wilf e Pugh John.
"Specialization, Generalization and Inheritance", Sigplan Notices, Vol.20, N8, Ago 1985.

[LIE86] - Lieberman Henry.
"Using prototypical objects to implement shared behavior in object oriented systems", OOPSLA'86 Conference Proceedings, Portland, pag.214-223, Set 1986.

[LIS81] - Liskov Barbara, Atkinson R., Bloom T. et al.
CLU Reference Manual, Springer Verlag, 1981.

[MAD86] - Madsen Ole Lehrmann.
"Block Struture and Object Oriented Languages", Technical Report, Computer Science Department, Aarhus University, Aarhus, Denmark, 1986.

[MEY87] - Meyer Bertrand.
"Reusability. The Case for Object Oriented Design", IEEE Software, pag.50-61, Mra 1987.

[MYL80] - Mylopoulos John e Wong Harry.
"A language facility for desigming Database Intensive Applications", Transaction on Database Systens (ACM), Vol.15, N2, pag.185-207, Jun 1980.

[PAS86] - Pascoe, Geoffrey.
"Elements of Object Oriented Programming", Byte, pag.139-143, Ago 1986.

[SCH86] - Schaffert Craig, et al.
"An introduction to Trellis/Owl", OOPSLA'86 Conference Proceedings, Portland, Oregon, Set 1986.

[STO86] - Stonebraker, Michael e Rowe, Lawrence.
"The Design of Postgress", Sigmod Record (ACM), Vol.15, N2, Jun 1986.

[SNY86] - Snyder Alan.
"Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA'86 Conference Proceedings, Portland, Oregon, Set 1986.

[TAD88] - Tadão Takahashi.
"Introdução à programação orientada a objetos", III EBAI, Curitiba, Jan 1988.

[TES85] - Tesler Larry.
"Object Pascal Report", Structured Language World, Vol.9, N3, 1985.

[VEL86] - Veloso, Paulo A.
"Tipos (abstratos) de Dados: Programação, Especificação, Implamentação", V Escola Brasileira de Computação, Belo Horizonte, 1986.

[WEG86] - Wegner, Peter.
"Classification in Object-Oriented Systens", Sigplan Notices, Vol.21, N10, Oct 1986.

[WEG87] - Wegner, Peter.
"Dimensions of Object Based Language Design", OOPSLA'87 Conference Proceedings, 1987.

[WIR82] - Wirth, Nicholaus.
"Programming in Modula 2", Spriger Verlag, 1982.