PUC

Monografias em Ciência da Computação
nº 22/92

Mecansimos de Herança em Linguagens de Programação: Variedade, Evolução e Proposta

Graciela H. Matich
Sérgio E. R. Carvalho

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22454-970
RIO DE JANEIRO - BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Monografias em Ciência da Computação, Nº 22/92
Editor: Carlos J. P. Lucena Julho, 1992

Mecanismos de Herança em Linguagem de Programação: Variedade, Evolução e Proposta *

Graciela H. Matich
Sérgio E. R. Carvalho

* Este trabalho foi patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República Federativa do Brasil..

Resumo

Este trabalho apresenta uma visão geral sobre mecanismos de herança, enfocando também a relação dos mesmos a conceitos como polimorfismo, reusabilidade e modificação incremental. A herança é considerada conforme o nível de definição, o que permite ajustar melhor a variedade de abordagens. O trabalho analisa também a evolução de critérios e questionamentos importantes sobre o assunto. Formula uma proposta de metodologia visando acompanhar o processo evolutivo de um sistema, de uma fase experimental a uma fase mais estável e otimizada, reunindo múltiplas alternativas de herança.

Palavras Chave: Herança, Delegação, Polimorfimo, Evolução de Sistemas

Abstract

This report presents an overview of inheritance discussing also its relationship to polimorphism, reusability and incremental modification. In this discussion inheritance topics are classified in different definition levels. This classification helps in the understanding of the existing variety of approaches to inheritance. The evolution of the inheritance concept and important related questions are also considered. Finally, a methodology for systems development is proposed, considering several inheritance alternatives to express the system, from an experimental phase to a static, optimized phase.

Key Words: Inheritance, Delegation, Polimorphism, Evolution of Systems.

ÍNDICE

1. INTRODUÇÃO
1.1. O Termo "Herança" no Contexto do Trabalho
1.2. Propósitos dos Mecanismos de Herança
1.3. O que Significa Herdar Atributos?
1.4. Herança em Termos de Polimorfismo
1.5. Incidência no Desenvolvimento de Sistemas Evolutivos

2. VARIEDADE EM MECANISMOS DE HERANÇA
2.1. Níveis de definição
2.1.1. Nível conceitual
2.1.2. Nível das implementações
2.2. Simples ou Múltipla, Completa ou Incompleta
2.3. Tipos e Classes
2.4. Modelos de Implementação Mais Conhecidos
2.4.1. Herança baseada em classes
3.4.1.1. Implementações em linguagens fortemente tipadas
2.4.2. Herança entre objetos

3. EVOLUÇÃO
3.1. Tendência a Linguagens com Maior Grau de Polimorfismo
3.2. Demanda de Maior Flexibilidade
3.3. Demanda de Maior Confiabilidade
3.4. Polêmica: Qual a Melhor Forma de Herança?
3.5. Separação entre hierarquias de especializações e hierarquias de implementações

4. PROPOSTA
4.1. Objetivos
4.2. Linhas Gerais da Proposta
4.3. Modos, Etapas e Ferramentas

5. CONCLUSÕES

6. REFERÊNCIAS BIBLIOGRÁFICAS

1. INTRODUÇÃO

A inclusão de mecanismos de herança constitui-se cada vez mais em fator preponderante e decisivo na modelagem de linguagens de programação.

Atendendo ao significado ordinário, "herdar é receber propriedades ou características de outro, normalmente como resultado de algum relacionamento especial entre o doador e o receptor" [DAN88]. Em linguagens de programação herdar tem também esse sentido, sendo que os relacionamentos são estabelecidos entre estruturas de dados e entre rotinas de comportamento que são atributos do doador e do receptor. O que se "herda", total ou parcialmente, são precisamente esses atributos.

Geralmente a presença de mecanismos de herança aparece em linguagens que suportam o conceito de objeto ("baseadas em objetos"), e considera-se traço seletivo para as chamadas "linguagens orientadas a objetos" [WEG87]. Objetos são entidades que reunem dados e rotinas ou operações para manipular esses dados. Algumas linguagens possuem outros conceitos como o de tipos e/ou de classes que determinam correspondentes níveis de abstração. Tanto tipos como classes, em diferentes aspectos, representam padrões para produzir objetos. Usualmente os objetos são instâncias concretas de uma classe ou tipo determinado.

Existem diversas formas de estabelecer os relacionamentos de herança, e também diferenças de critérios para decidir entre que entidades se estabelecem os relacionamentos: se entre instâncias de objetos diretamente, se entre tipos, se entre classes ou qualquer outro esquema para representar entidades de dados e programas.

A herança pode ser analisada de diversas perspectivas: organização conceitual, implementação, aplicações. Além do mais a modelagem de mecanismos de herança está ligada a um nutrido conjunto de temas, entre eles:

polimorfismo, reusabilidade, modificação incremental, amarração (ou resolução de vínculos) e técnicas para aproveitamento de recursos.

Este trabalho se propõe oferecer, em primeira instância, uma visão geral sobre mecanismos de herança; ajustar a variedade de abordagens conforme o nível de definição, e enfocar a relação da herança a conceitos como polimorfismos, reusabilidade e modificação incremental. Outro objetivo do trabalho é analisar a evolução de critérios e questionamentos importantes sobre a modelagem de mecanismos de herança. Finalmente pretende colocar uma proposta de metodologia capaz de acompanhar a evolução natural de um sistema, de uma fase experimental a uma fase mais estável e otimizada, reunindo múltiplas alternativas de herança.

Numa seção introdutória são esclarecidas convenções, analisados propósitos dos mecanismos de herança, assim como tratada a vinculação da herança ao conceito de polimorfismo e ao desenvolvimento de sistemas evolutivos.

Na seção 2 os mecanismos de herança são enfocados conforme a correspondência de seu nível de definição ao plano conceitual ou ao plano das implementações. São estabelecidas diferenças entre o conceito de tipo e de classe, para possibilitar um melhor entendimento do assunto. Na seção 2.4 tratamos os modelos de implementação mais conhecidos: o baseado em classes, e a delegação como modelo de herança direta entre objetos sem elaborar padrões.

Na seção 3 consideramos a evolução de critérios conforme as tendências e demandas, prentendendo satisfazer simultaneamente objetivos de flexibilidade e de confiabilidade.

Uma proposta é esquematizada na seção 4, através dos seus objetivos e linhas gerais. Sugerimos como parte da proposta os modos, etapas e

ferramentas necessários a um ambiente para acompanhar o processo evolutivo de um sistema, aplicando diversos critérios sobre herança.

1.1. O Termo "Herança" no Contexto do Trabalho

Neste trabalho inicialmente incluímos, nos termos "herança" ou mecanismos de herança, modalidades de realização que serão diferenciadas posteriormente, na seção 2. Esta generalização permite tratar algumas características comuns, além de, com refinamentos, qualquer modelo específico.

1.2. Propósitos dos Mecanismos de Herança

O propósito principal dos sistemas de herança é construir estruturas que possam ser Estendidas em muitas formas diferentes [87]. Este enfoque pragmático pode-se completar considerando os propósitos de reusabilidade a vários níveis, e os propósitos de ordem conceitual.

A reusabilidade é uma das metas mais prezadas que os produtos de *software* pretendem atingir. O termo "reusar" tem o significado de poder usar um elemento numa situação diferente da original para a qual foi criado, reduzindo e simplificando esforços. A herança tem implicitamente um propósito de reusabilidade, já que proporciona uma forma de reaproveitar características capturadas por componentes, seja na forma de objetos, tipos ou classes.

A reusabilidade abrange muitas formas possíveis, incluindo o reaproveitamento de código e o reaproveitamento de especificações. Ambos os casos são possíveis de realizar com mecanismos de herança.

A herança, quando abordada também conceitualmente, oferece um marco de organização muito favorável. Agrupando os elementos com base em seus atributos essenciais, e estabelecendo vínculos que guardem uma relação

conceitual, é possível proporcionar um sistema de classificação para o domínio do problema a tratar.

1.3. O Que Significa Herdar Atributos?

Os relacionamentos de herança são estabelecidos, como mencionado acima, entre entidades definidas por conjuntos (finitos) de atributos. Para uma entidade, herdar atributos é assumir ou considerar esses atributos como próprios, embora eles não estejam definidos localmente nessa entidade.

Como na figura 1, definimos que a entidade B tem atributo b1 e além disso herda os atributos da entidade A (estes são {a1 e a2}), para os clientes ou usuários da entidade B o conjunto de atributos oferecidos consiste em {a1, a2 e b1}.

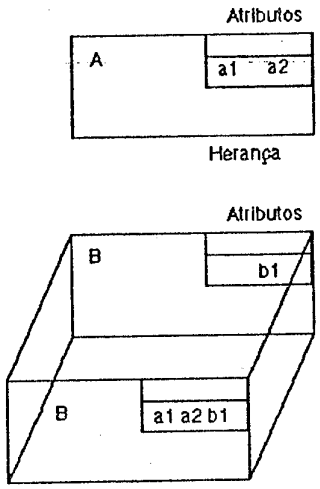

Figura 1 - Visão dos usuários da entidade B

No nível das implementações, os atributos herdados, sejam de comportamento (operações) ou de dados (estruturais) podem ser amarrados à entidade herdeira em forma antecipada à execução (*early binding*) ou em forma tardia, durante a execução (*late binding*).

Esta diferença dos tempos de amarração também tem relação com a forma de internalizar atributos (copiando ou compartilhando); é base para os diferentes modelos, e é motivo de numerosas polêmicas sobre herança e sobre orientação a objetos em geral.

1.4. Herança em Termos de Polimorfismo

Os mecanismos de herança, em diversas modalidades, implementam determinados aspectos da noção de polimorfismo.

Polimorfismo é a habilidade de um elemento adquirir diferentes formas ou se comportar como pertencendo a categorias ou conjuntos diferentes. O conceito de conjunto, por sua vez, se associa em linguagens de programação a tipos de dados. Porém, também expressam polimorfismo, os mecanismos de herança inseridos em linguagens não tipadas (ou que não contam com um sistema de tipos).

Na herança, costumam-se incluir manifestações do chamado "polimorfismo por inclusão" [BLA89]. Basicamente, no polimorfismo por inclusão, um tipo (ou conjunto) está incluído dentro de outro. Esta inclusão faz com que um objeto possa pertencer a mais de um tipo.

Um aspecto do polimorfismo por inclusão aborda o tratamento de objetos. Os objetos que pertencem a um conjunto ou tipo podem ser tratados "como se fossem" de um outro conjunto ou tipo pela propriedade de inclusão, ou por estar na interseção de conjuntos. Para referenciar resta característica freqüentemente usa-se o termo polimorfismo "as-if" [BLA89]. O compartilhamento de comportamento através da propriedade de inclusão de conjuntos acontece de forma implícita , mas em algumas linguagens também podem aparecer descrições explícitas.

O "polimorfismo operacional", por sua vez, apresenta duas manifestações: o sobrecarregamento de operadores e o polimorfismo paramétrico. Delas, o sobrecarregamento de operadores é a acepção comumente achada nos mecanismos de herança (Smalltalk [GOL83], Objective-C [COX86]). O sentido desse sobrecarregamento é que uma operação com o mesmo nome existe em tipos, classes ou protótipos diferentes, podendo mudar seu significado. Uma operação específica é selecionada no momento da amarração ou resolução de vínculos, em relação a uma instância particular.

O polimorfismo paramétrico, que consiste basicamente em pacotes de comportamento genérico, parametrizados, para trabalhar sobre diferentes tipos, não tem *a priori* vinculação direta com herança, se bem que pode ser um complemento interessante, que proporcione mais uma dimensão nas hierarquias. A linguagem Eiffel [MEY87], é um exemplo neste sentido, incluindo polimorfismo paramétrico além do mecanismo de herança.

1.5. Incidência no Desenvolvimento de Sistemas Evolutivos

O processo de desenvolvimento de *software*, numa concepção mais ampla, deve ser considerado como um processo de crescimento e evolução, mais do que um processo de construção e reconstrução de unidades isoladas. Assim, o *software* deve crescer como uma planta, a partir de sementes pequenas, mas com vida potencial.

As técnicas de modificação incremental constituem uma contribuição notável a estes objetivos. Usando uma perspectiva de modificação incremental é possível conceber entidades novas baseando-se em outras previamente existentes. Por exemplo, as especificações recursivas são incrementais, desde que especificam o problema do parâmetro (n + 1) em termos do problema de parâmetro n. O critério evolutivo pode ser aplicado a várias etapas na construção

de *software*: especificação incremental, edição incremental, assim como construção de código e estruturas incrementais.

A herança pode ser descrita como um mecanismo de modificação incremental particular. Wegner oferece esta visão em [WEG89], definindo herança simples da seguinte forma:

1) Seja uma entidade P, ou entidade progenitora, definida pelos atributos (P1, P2, ..., Pp)
2) Seja a entidade M, ou entidade modificadora, definida pelos atributos (M1, M2, ..., Mm)

   A herança permite obter a entidade resultado R, definida pelos atributos (R1, R2, ..., Rr), como uma composição especial da entidade P com a entidade M, que pode expressar-se: R = P + M, onde o símbolo '+' deve ser considerado como operador dessa composição especial, que se interpreta como o resultado R herda P e é modificado por M.

Posteriormente, por sua condição de técnica incremental, a herança permite evoluir a partir de entidades da forma R = P + M, obtendo entidades da forma R1 = R + M1. Os atributos da entidade resultado R tem determinadas características, dependendo das entidades progenitora e modificadora serem independentes entre si, ou terem uma sobreposição de papéis como conseqüência de ter idênticos nomes.

2. VARIEDADE EM MECANISMOS DE HERANÇA

2.1. Níveis de Definição

Para discutir a variedade de modelos de herança é conveniente estabelecer pautas que ajudem a esclarecer o panorama. Existem trabalhos que abordam herança em níveis de referência e definição diferentes, e em conseqüência apontam a tópicos e linguagens de expressão diferentes.

Reconhecemos propostas orientadas a aspectos do nível conceitual de desenvolvimento [CAR85] [WEG88] [CLE88] enquanto outras tratam exclusivamente questões de implementação [GOL83] [HAI87] [LAL86] [LIE86]. Raramente surge um modelo que intente cobrir esses dois níveis.

2.1.1. Nível conceitual

A herança, idealmente, deve permitir a formação de hierarquias ou redes que respeitam uma intencionalidade conceitual entre componentes, especificando comportamento e fazendo uso de ferramentas de apoio à modelagem conceitual, como por exemplo ambientes e sistemas especialistas que orientem a busca e definição de entidades. Nem todas as propostas observam alguma finalidade conceitual na derivação de entidades.

Neste nível, podemos destacar regras que governam a derivação e modificação de entidades, dando lugar a modelos conceituais de herança. Em [WEG88] achamos uma discussão completa sobre regras ou marcas de permissão para a modificação incremental. Mencionamos aqui três delas que consideramos básicas:

Compatibilidade de comportamento

Compatibilidade de características sintáticas

Compatibilidade de nomes

A **compatibilidade de comportamento** coloca fortes restrições no mecanismo de herança. As entidades participantes são tipos caracterizados por modelos algébricos e axiomáticos. A compatibilidade de comportamento entre tipo derivado (ou subtipo) e tipo progenitor é completa quando os domínios das operações são idênticos, e para todas as operações do tipo progenitor existem resultados correspondentes para argumentos correspondentes.

Para expressar uma semântica completa de herança, fundamentalmente no caso de impor compatibilidade de comportamento, deve-se recorrer a especificações formais. Dentro dos modelos matemáticos que podem ser fundações para herança, a noção de tipos como ideais (*types like ideals*) baseada na teoria de conjuntos, é bastante usada. Sobre estes conceitos Cardelli [CAR84] desenvolveu uma semântica formal do herança em termos de tipos e subtipos. Também no nível das especificações formais para herança Clerici [CLE88] propõe a linguagem GSBL, de corte algébrico, que permite especificações incompletas, refletindo uma noção de especialização de conceitos e refinamentos mais próxima ao nível da programação.

Na **compatibilidade de características sintáticas** as entidades vinculadas são definidas por moldes sintáticos e podem não ter uma especificação semântica do seu comportamento. Em particular, a maioria das linguagens de programação com sistemas de tipos, definem a noção de subtipo em termos de restrições sobre moldes sintáticos verificáveis em tempo de compilação [SCH86] [MAT90].

Na **compatibilidade de nomes** as entidades relacionadas são moldes especificados diretamente por suas implementações (geralmente na forma de

classes). Esta norma não requer verificação adicional nem em tempo de compilação nem em tempo de execução. Por esse motivo constitui o único modelo aplicável a linguagens de programação que não são fortemente tipadas [GOL83] [COX86].

2.1.2. Nível das implementações

A herança no nível das implementações é basicamente um mecanismo de reaproveitamento de *software* que envolve várias alternativas ou pontos de discussão, como:

* Entre que entidades se estabelecem os vínculos para reaproveitar Características ? Entre grupos de objetos caracterizados por uma descrição comum (como podem ser as classes) ou entre objetos isolados diretamente, sem necessidade de criar intermediários?

* De que forma o atributo herdado é incorporado ou internalizado pela entidade herdeira: copiando ou compartilhando? Como é amarrado o atributo: em forma estática e fixa, ou em forma dinâmica e até variável?

A extensão é uma técnica de implementação de herança muito usada, que permite correspondência com o critério do nível conceitual de especialização ou refinamentos de características. Estender uma entidade já existente permite gerar uma entidade nova descrevendo só as características ou atributos adicionais àqueles da entidade progenitora. Um dos problemas que devem resolver-se nesta técnica é se manter ou não o encapsulamento da entidade progenitora para a visão da entidade derivada [SNY86].

Com a herança, além dos usuários externos de uma entidade aparecem os usuários "internos", que são aquelas entidades que dentro da organização devem incorporar atributos por herança. Deve fixar-se uma política para esses

usuários "internos"? Eles tem acesso a determinados detalhes de implementação, como estrutura de dados empregada? É possível necessário ou conveniente manter o encapsulamento também para eles? Estes questionamentos são pontos críticos na maioria das implementações de herança [MAT90].

2.2. Simples ou Múltipla - Completa ou Incompleta

Na modalidade HERANÇA SIMPLES é gerada uma organização hierárquica arborizada, onde cada entidade tem um único progenitor do qual pode herdar propriedades. Este modo é o mais comum, adotado entre outras linguagens por Smalltalk-80 (*standard*) e Objective-C.

Na HERANÇA MÚLTIPLA a organização gerada não fica restrita à forma de árvore, sendo em geral uma rede ou grafo dirigido acíclico. Cada entidade pode ter mais de um progenitor. Cada entidade é um nó do grafo, com arcos saindo para seus progenitores que expressam o relacionamento herda-de". Exemplo:

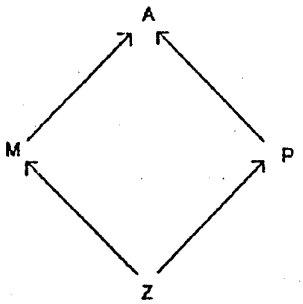

Figura 2

A entidade Z, no exemplo herda atributos de M e de P, enquanto M e P, por sua vez, herdam atributos de A. A herança múltipla oferece maior flexibilidade e alternativas que a herança simples, porém é usada com menor

freqüência por apresentar alguns problemas de implementação que devem ser resolvidos com a escolha de estratégias bem definidas.

Se uma entidade herda atributos do mesmo nome, por exemplo operações de dois ou mais progenitores, chega-se a um ponto de conflito. Pode-se colocar a restrição de que todos os nomes de operações devem ser diferentes, com a conseqüente perda de polimorfismo. Uma estratégia mais razoável consiste em estabelecer uma ordem de prioridades sobre os progenitores para resolver este conflito. O grafo da herança pode também ser transformado numa cadeia linear, e então trabalhado com regras de herança simples.

No caso de HERANÇA COMPLETA uma entidade herda absolutamente todos os atributos do progenitor. Esta modalidade é a mais usual. Na HERANÇA INCOMPLETA, ou parcial, uma entidade pode herdar parte dos atributos de outra entidade, não necessariamente todos. A implementação desta modalidade não é freqüente.

2.3. Tipos e Classes

A maioria dos mecanismos de herança implementados em linguagens precursoras no tema, como Simula, Smalltalk, Objective-C e outras, sustentam hierarquias de classes. O conceito de classes aparece muitas vezes associado ao de tipos de dados. Tipos e classes definem coleções de objetos com atributos uniformes, mas tem em essência funções diferentes, se bem que na prática freqüentemente eles apareçam tratados sem distinção.

Os tipos modelam o domínio de aplicação, definindo comportamento de agrupamentos de objetos. Um tipo pode ser definido por predicados que permitam reconhecer posteriormente expressões válidas para esse tipo.

As classes, por sua vez, caracterizam um conjunto de objetos a nível de implementação. Uma classe é uma entidade abstrata, um molde que define uma representação possível para os objetos, e implementa as operações que manipulam esses objetos, inclusive a operação que cria um objeto particular. Nesta ótica um objeto é sinônimo de instância de classe.

O comportamento expressado por um tipo teoricamente deve ser o comportamento essencial e invariante ante qualquer implementação. Expressar isso não é fácil. Para ser rigoroso é necessário usar linguagens formais que especifiquem a semântica do comportamento. Quando o comportamento é muito difícil de especificar em forma precisa e concreta, então se aproxima por um molde sintático, opção muito usada.

Por estes motivos, nas linguagens de programação práticas, usualmente, tipos tem uma semântica de verificação (*type checking*) enquanto classes tem uma semântica relativa a criação e manipulação concreta de objetos.

Em relação à herança, se tipos são definidos por predicados, a geração de subtipos corresponde à modificação de predicados. Subtipos são definidos em termos de restrições que determinam um subconjunto do conjunto definido pelos predicados do progenitor. A formação de subclasses, porém, pode expressar-se como modificação de padrões.

2.4. Modelos de Implementação Mais Conhecidos

2.4.1. Herança baseada em classes

O modelo baseado em classes é o precursor em matéria de implementação de mecanismos de herança. A linguagem Simula-67 [DAH67] introduz a terminologia de "classes", sendo adotada posteriormente por Smalltalk [GOL83], a mais difundida das linguagens orientadas a objetos. Uma classe (2.3)

define um gabarito para a criação e manipulação de objetos. Usando mecanismos de herança é possível formar incrementalmente, hierarquias de classes. Uma classe pode definir-se a partir de outra classe existente (progenitora ou "superclasse" em algumas terminologias). Geralmente mediante uma técnica de extensão a classe derivada ou subclasse, acrescenta características ou atributos específicos as características recebidas por herança de classes ancestrais. Uma classe pode ser considerada instância de outra classe, assim como um objeto nasce como instância de uma classe. Na Figura 3 vemos um esquema simplificado de hierarquias de classes.

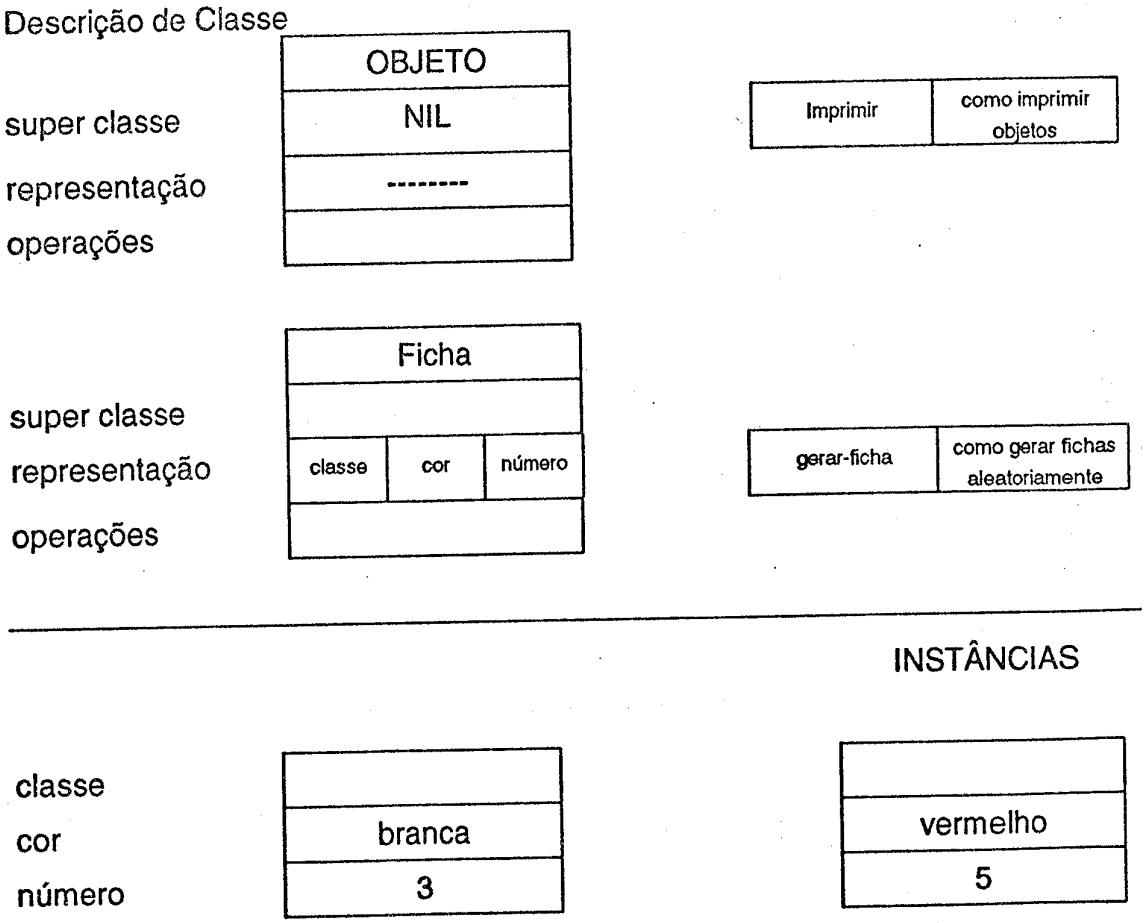

Figura 3 - Uma Hierarquia de classes

Na maioria das implementações, as operações (atributos de comportamento) de uma classe podem ser redefinidas em subclasses. Desta forma operações com o mesmo nome aparecem com significado diferente, também chamado "sobrecarregamento de operadores", não respeitando desta forma uma compatibilidade de comportamento. A compatibilidade de características sintáticas fica limitada àquelas implementações em linguagens tipadas.

Em muitas linguagens que adotam este modelo, a amarração de atributos se realiza dinamicamente ou em forma tardia (*late binding*), como em Smalltalk. O objetos são amarrados a uma classe no momento em que são criados, em tempo de execução, não existindo declarações estáticas prévias para associar objetos a classes.

Quando deve ser realizada determinada operação (ou método) é desencadeado um algoritmo de busca desse atributo. Primeiro a operação é procurada na classe à qual o objeto pertence; caso não esteja na classe, a busca continua pela superclasse imediata, depois pela superclasses desta e assim sucessivamente. Devido a redefinições, atributos de um nível mais baixo na hierarquia escondem atributos do mesmo nome de um nível superior.

2.4.1.1. Implementações em linguagens fortemente tipadas

Nem todos os modelos de herança baseada em classes tem verificação de tipos de dados. Do ponto de vista de programação, um sistema de tipos proporciona uma cobertura de proteção que regulamenta a modalidade operacional entre objetos. As linguagens em que se verifica consistência de tipos de todas as expressões são chamadas fortemente tipadas.

Dentro do modelo de herança baseada em classes, exemplos de linguagens fortemente tipadas são Trellis/Owl [SCH86], Eiffel [MEY87] e Tool

[CAR92]. Estas linguagens tem declarações para associar objetos a determinado tipo ou classe, que permitem realizar uma amarração adiantada de atributos (*early binding*). A verificação de que uma operação existe e está dentro das possibilidades de comportamento de um determinado objeto se realiza em tempo de compilação, o que otimiza o desempenho da linguagem e oferece maior segurança.

2.4.2. Herança entre objetos: Delegação

Modelos alternativos estabelecem os relacionamentos de herança diretamente entre objetos (instâncias de dados e comportamento). O mecanismo mais conhecido para definir incrementalmente objetos similares é conhecido como DELEGAÇÃO [LIE86] [STE87], inicialmente implementado em algumas linguagens de atores como ACT1 [AGH86] e posteriormente adotado com variantes em outras propostas como a linguagem SELF [UNG87].

A delegação é uma forma menos restrita de reaproveitamento de código que propicia um compartilhamento mais direto de atributos e valores, sem criar moldes nem intermediários. Não existe diferenciação entre classes e instâncias. Existem simplesmente objetos, representando itens individuais do mundo real. Alguns desses objetos podem servir de referência para a criação de outros objetos que tenham características similares. Usualmente esses objetos-referência são chamados de "protótipos".

As instâncias similares ao protótipo e que se referenciam a ele são chamadas de objetos extendidos. Cada objeto estendido tem uma parte própria, que define comportamento e características desse objeto em particular, e uma parte extensão. A parte extensão consiste numa lista de ponteiros aos protótipos que usa como referência. Esta parte pode ser compartilhada por outros objetos, total ou parcialmente.

Este mecanismo baseia seu nome no fato de permitir aos objetos "delegar" responsabilidades em outros. Um objeto pode passar a responsabilidade de realizar uma operação a um ou mais ancestrais ou protótipos. Quando um objeto recebe uma mensagem (ou ordem de executar determinada tarefa), ele primeiro tenta responder a mensagem usando o comportamento armazenado na parte pessoal. Se o objeto verifica que com esse comportamento não pode responder, automaticamente "delega" a mensagem aos protótipos que ele referencia.

Através do processo de delegação, os recursos são compartilhados dinamicamente durante a execução. Isto se realiza mediante a amarração dinâmica da auto-referência do objeto (o SELF). Podemos considerar, para cada objeto, a existência de uma variável SELF que contem uma referência a ele própria (por isso auto-referência). Quando uma mensagem é delegada, a variável SELF é passada junto com a mensagem como referência ao objeto que originariamente recebeu a mensagem. A mensagem então, pode ser respondida com operações e/ou valores do protótipo, que no momento da execução são tratados como propriedades do objeto que originariamente recebeu a mensagem. Dessa forma, uma operação pode pertencer a diferentes entidades em diferentes momentos de execução.

A delegação pode apresentar-se como um modelo mais geral [STE87] que permite implementar e simular a funcionalidade proporcionada pelas classes. Para este propósito, o comportamento comum a uma coleção de objetos pode combinar-se num único objeto, que desempenhe o papel de classe.

3. EVOLUÇÃO

3.1. Tendência a Linguagens com Maior Grau de Polimorfismo

As linguagens de programação tem evoluindo notavelmente, desde seus começos, a respeito do critério sobre polimorfismo. A tendência inicial para programação de grandes sistemas era usar linguagens monomórficas com uma verificação estrita de tipos. Podemos considerar linguagem monomórficas [BLA89], aquelas que possuem tipos de dados embutidos completamente independentes, sem interseção nem possibilidade de ter igual comportamento.

Em linguagens como Ada e CLU aparecem traços de polimorfismo operacional, seja na forma de pacotes genéricos aplicáveis a diferentes tipos de dados, ou pelo sobrecarregamento de operadores.

As linguagens funcionais, como Lisp e as linguagens em lógica, como Prolog, não tem sistema de tipos e permitem que suas funções e procedimentos se apliquem a todo um espectro de parâmetros que cumpra seus predicados. Mas, estas linguagens podem considerar-se casos especiais, por ser pouco empregados para sistemas convencionais.

A formação de hierarquias incrementais e organizações em rede, onde um tipo pode reunir características e comportamento de um ou vários, junto com o interesse crescente por reusabilidade, imprimem forte impulso ao polimorfismo.

Atualmente, existem questionamentos e uma busca de caminhos para conseguir linguagens cada vez mais ricas em termos de polimorfismo, perfilando-se em algumas propostas uma migraçao de critérios, indo de polimorfismos com fortes restrições, como o polimorfismo "as-if", ou polimorfismo "is-a", a uma série de variantes mais permissivas nas formas de compartilhar comportamento.

3.2. Demanda de Maior Flexibilidade

As pessoas dedicadas a uma programação experimental, valorizam muito a flexibilidade proporcionada por uma linguagem ou mecanismo em particular. Neste estilo de programação, também se desejam ferramentas que ofereçam rapidez na construção, avaliação e modificação de *software*. As linguagens tipadas e a verificação de tipos em geral interferem na prototipagem rápida porque dificultam mudanças de decisões. Estes motivos fomentam o fato de não se querer declarações estáticas que amarem um objeto a um tipo/classe, e argumentam a favor de mecanismos de herança com amarração dinâmica de atributos.

3.3. Demanda de Maior Confiabilidade

Confiabilidade, corretude e fácil manutenção são requisitos que a comunidade na área de producão de sistemas relativamente convencionais impõe às ferramentas de construção de *software*. Deste ponto de vista, as linguagens com sistemas de tipos oferecem maior segurança, porque colocam restrições sintáticas e semânticas que servem para a própria proteção de objetos. Além do mais, um dos objetivo de um alto nível na definição de tipos é capturar invariantes do sistema, fundamentais para provas de corretude.

Na programação exploratória pode ser até vantajoso um mecanismo que permita que mudanças arbitrárias sejam propagadas instantaneamente. A programação para a produção, porém, requer um sistema mais controlado. O compartilhamento direto de estruturas e valores por parte de objetos, como acontece em algumas propostas sobre delegação, pode ser arma perigosa, neste sentido. Os objeto ficam dependentes uns dos outros, podendo ocasionar dificuldades ou efeitos inesperados, ao produzir-se a modificação de algum deles.

Nos modelos de herança baseada em classes, os objetos somente compartilham atributos através do padrão representativo, que é a classe, mas cada objeto armazena seus valores específicos nas variáveis de instância, mantendo estados independentes e proporcionando em conseqüência maior confiabilidade.

3.4. Polêmica: Qual a Melhor Forma de Herança?

É difícil determinar qual a melhor forma de herança, porque diferentes situações de programação precisam de diferentes combinações de características. Para ambientes de programação exploratória ou experimental é desejável facilitar a flexibilidade de sistemas, o dinamismo e o compartilhamento direto de recursos entre objetos, enquanto que para ambientes de produção de *software* convencional as opções indicadas são outras: a confiabilidade que oferece a verificação de tipos, a eficiência da amarração estática e os modelos baseados em compartilhamento de características por grupos de objetos (como tipos ou classes).

É possível modelar uma linguagem e um mecanismo de herança que combine as qualidades exigidas por um e outro estilo [PAI89]? A resposta não é trivial nem imediata, porque em primeira instância as metas se apresentam como contrapostas, mas existem algumas sugestões a respeito. Uma proposta que pretende satifazer este questionamento é formulada na seção 4.

Snyder [PAI89], por exemplo, destaca que um ponto de convergência em que se beneficiam as duas áreas é no incremento da reusabilidade. Reusando componentes se acelera a construção de *software* e se promove a confiabilidade, desde que se assuma que os componentes reusados são confiáveis e eficientes. Uma sugestão visando incrementar a reusabilidade é dar mais importância às interfaces para herança, que podem ir desde interfaces de especificações formais a interfaces práticas para verificação de algumas compatibilidades, como aridade

e tipo de argumentos de funções e procedimentos. Assim também, a alternativa de separar os módulos de especificações e implementações, criando em conseqüência hierarquias independentes, favorece a flexibilidade e melhora o reaproveitamento [IER90].

Outra pergunta interessante que afeta a qualidade de herança é: O que se deve herdar: as estruturas de implementação e comportamento (isto é, um maior reaproveitamento de componentes), ou o comportamento somente? (Isto significa se abstrair de implementações). O tema do encapsulamento, sem dúvida, é relevante à confiabilidade de sistemas.

3.5. Separações Entre Hierarquias de Especificações e Hierarquias de Implementações

Toda abstração bem elaborada, consta de uma especificação e um módulo de implementação. A especificação descreve o que faz, mas omite qualquer informação sobre como é implementada a abstração em questão. Uma implementação, por sua vez é correta se proporciona o comportamento básico definido pela especificação. Observando o que acontece com as hierarquias geradas por vínculos de herança, quando as hierarquias de implementações e instâncias são forçadas a combinar com as hierarquias de especificações, acabamos com relacionamentos bem compreendidos, mas geralmente com implementações ineficientes. Ao contrário, quando a hierarquia de especificações deve se adaptar à hierarquia de implementações, podemos acabar com implementações eficientes, mas com relacionamentos lógicos errados. A organização lógica de um sistema não pode ser modelada pensando na melhor forma de reaproveitar as implementações.

Separar especificações de implementações é fundamental para efeitos de alterações, extensibilidade, provas de corretude, contribuindo também à

reusabilidade. Considerando "tipo" estritamente como especificação de comportamento para objetos e "classe" como implementação possível para um tipo, a idéia extende-se em permitir hierarquias de tipos independentes de hierarquias de classes [IER90]. Mas, em outros modelos, como por exemplo o formulado no sistema Actra [LAL86] o nome "classe" é usado para uma descrição mais abstrata, independente de representações, surgindo o termo exemplares (*exemplars*) para padrões de implementação. Nesse modelo os objetos são criados por cópia de *exemplars* e permite que um objeto possa ter múltiplas representações, o que valoriza a alternativa como bastante completa.

A representação de cada módulo de especificações para grupo de objetos (chame-se de "tipo" ou de outra forma) pode contar com uma parte ou lista que associe este módulo a um ou mias módulos de implementação, onde estará descrita uma representação em particular para objetos e comportamento em função dessa representação.

4. PROPOSTA

4.1. Objetivos

Esta proposta pretende satisfazer a demanda de mecanismos de herança adequados a interesses e formas de trabalho diferentes num mesmo ambiente de desenvolvimento, que permita acompanhar a evolução natural de um sistema desde sua origem. Geralmente os sistemas vão evoluindo de uma fase experimental e desorganizada sobre pouco volume de dados, a uma fase mais estável sobre maior volume de dados, desejando um melhor rendimento para a produção. Deseja-se, então, a possibilidade de ter subconjuntos de linguagem com formas de herança e comportamento apropriados a cada etapa, junto comn ferramentas que facilitem o trânsito por elas.

4.2. Linhas Gerais da Proposta

A proposta está baseada num incremento de restrições, indo de uma forma mais livre e dinâmica, a formas mais padronizadas e controladas para otimizar rendimento e confiabilidade. O modelo consiste num ambiente de desenvolvimento de programas, onde existem modos de definição e de operação. Dependendo do modo, se habilita um subconjunto de linguagem, seu nível de restrições e permissões, com o mecanismo de herança apropriado. Os modos sugeridos inicialmente são três:

. Modo objeto

. Modo padrão

. Modo tipo

Para acompanhar a passagem de um modo a outro se propõem algumas ferramentas com os seguintes propósitos:

(1) Para geração de padrões (ou classes) a partir de objetos isolados e protótipos.

(2) Para seguimento dos padrões de implementação (ou classes) definidos no sistema, mantendo um diretório de padrões com seus propósitos, com possibilidade de sugerir declarações.

(3) Para definir interfaces de comportamento.

4.3. Modos, Etapas e Ferramentas

Na Figura 4, apresenta-se um esquema desta proposta com modos, etapas e ferramentas. Supomos o início de desenvolvimento de um sistema ingressando no MODO OBJETO. Este modo facilita o trabalho experimental, permitindo uma modelagem rápida e facilmente alterável, sendo os objetos as únicas entidades deste modo. Habitalita-se um subconjunto de linguagem

inserido num sistema dinâmico, com alto grau de permissões, no sentido de redefinições de comportamento, sem declarações nem verificações sintáticas em tempo de compilação, que permite modelar similaridades entre objetos, sem criar moldes. Neste subconjunto podem ser definidos objetos protótipos e objetos estendidos, contando com um mecanismo de delegação como o exposto em (2.4.2). Na Figura 5 sugerimos uma representação para todo objeto neste modo e para as entidades representativas dos outros modos.

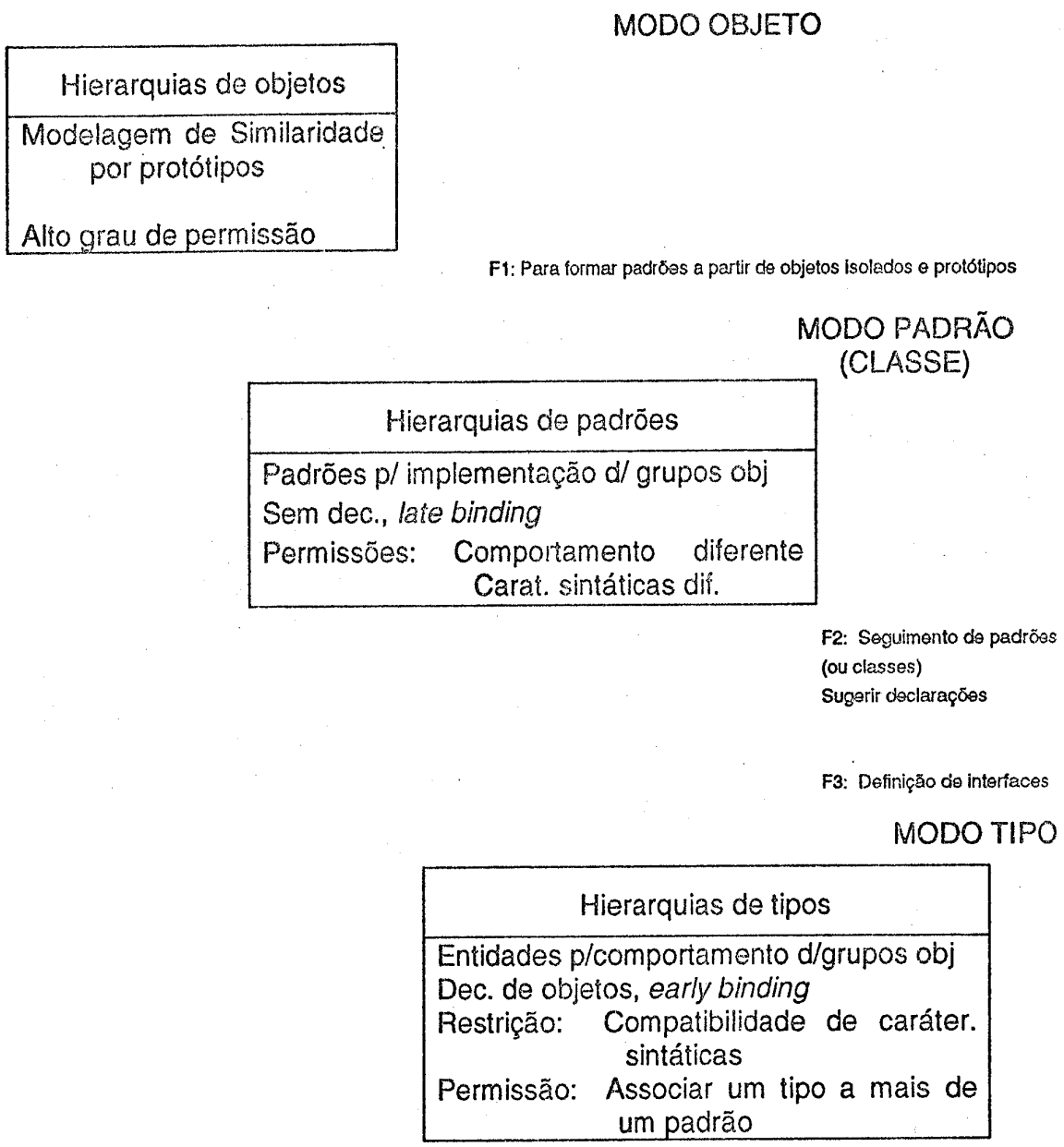

Figura 4 - Modos, etapas e ferramentas na proposta

Se a prototipagem gerada no modo objeto oferece resultados medianamente satisfatórios, e pretende-se trabalhar com coleções mais numerosas de dados, pode empreender-se a etapa seguinte passando ao MODO PADRÃO. Para acompanhar esta passagem, propõe-se a definição e emprego

de uma ferramenta de *software* para a formação de padrões a partir de objetos isolados e protótipos. Estes padrões, podem também chamar-se de classes ou *clusters*, desde que tenham o sentido de padrões de implementação para objetos, que definem uma representação ou estrutura (o REP) e caracterizam o comportamento (operações para manipular objetos) em função do REP definido. No modo padrão, o subconjunto da linguagem habilitado também não tem declarações para associar objetos a padrões em tempo de compilação. Os objetos são criados e associados a algum padrão em tempo de execução, mediante a execução de uma operação de criação (por exemplo: New-P1 ( ) seria a operação para a criação de um objeto, conforme o padrão P1). São permitidas redefinições de operações (comportamento diferente) e características sintáticas diferentes.

A etapa seguinte, que é passar ao MODO TIPO é sugerida quando os sistema atinge uma determinada estabilização e podem realizar-se algumas otimizações, como por exemplo amarrações adiantadas (ou em tempo de compilação), e também com o objetivo de aumentar a flexibilidade do sistema permitindo diferentes representações para um objeto. Esta passagem apresenta-se bastante complicada. Para viabilizar esta idéia considera-se necessário o apóio de duas ferramentas: uma para manter um diretório de padrões de implementação, com a capacidade de sugerir declarações e definições de características sintáticas (F2) e outra para definir interfaces de comportamento (F3), esta última capaz de fazer algumas abstrações para extrair conclusões independentes da implementação usada. O modo TIPO tem por objetivo a definição de entidades descritivas de comportamento para grupos de objetos. Inicialmente consideramos somente a possibilidade de uma descrição aproximada do comportamento por características Sintáticas ou *signatures*. A ferramenta F2 permitiria introduzir algumas declarações e com F3 pretendemos facilitar a descrição de entidades TIPO, se bem que esta perspectiva não se

apresenta fácil. Os objetos no modo TIPO, são declarados como pertencendo a um tipo e por sua vez, cada tipo deverá declarar-se associado no mínimo a um padrão, oferecendo também a opção de associar-se a mais de um padrão.

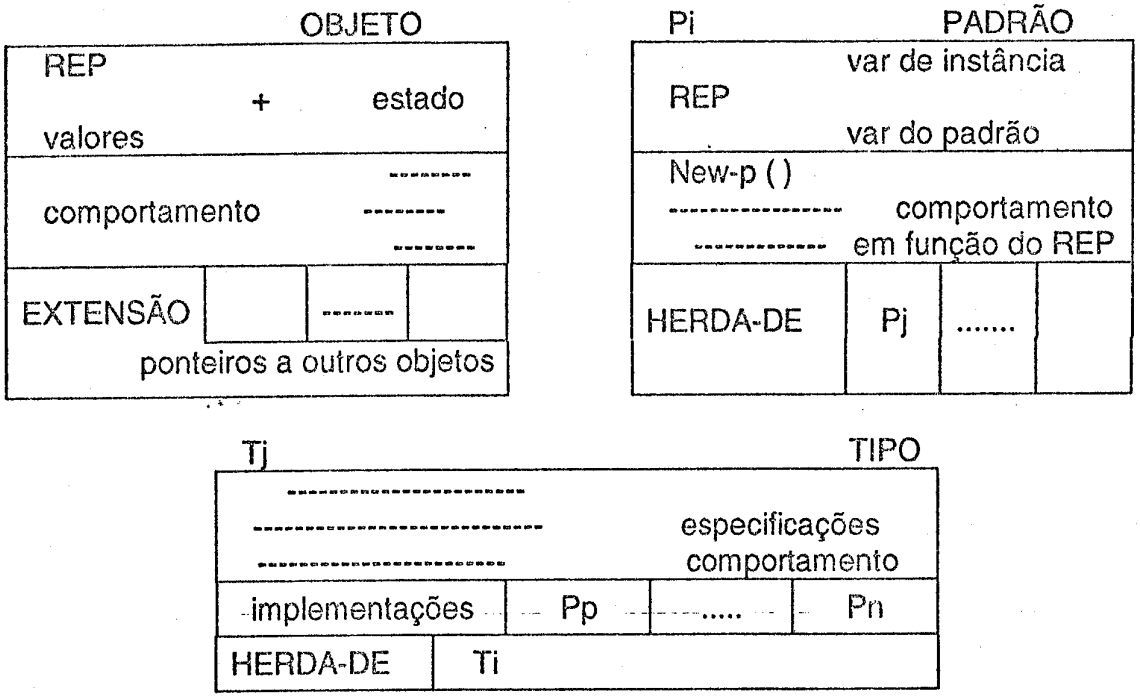

Figura 5 - Representação de entidades na proposta

5. CONCLUSÕES

Os mecanismos de herança oferecem no contexto do desenvolvimento de *software* uma prezada contribuição ao reaproveitamento de componentes. Esta contribuição pode atingir o reaproveitamento da definição conceitual de um módulo, pode estar baseada simplesmente no reaproveitamento de código, ou pode apontar ambos os objetivos.

Desde que módulos novos podem ser definidos como extensão ou combinação de outros, previamente existentes, os mecanismos de herança

constituem uma ferramenta que propicia a modificação incremental e a evolução de sistemas.

No plano das implementações, os mecanismos de herança, baseados na definição de moldes ou padrões referenciais (as classes) para o reaproveitamento, marcam uma diferença notável a respeito daqueles baseados no reaproveitamento direto de atributos de objetos. Os mecanismos de herança implementados em linguagens tipadas oferecem maior confiabilidade e previsão no comportamento de um sistema, enquanto aqueles associados a linguagens não tipadas com amarração dinâmica de atributos são os preferidos pelos usuários que apreciam a flexibilidade.

As demandas de maior flexibilidade, e de maior confiabilidade no desenvolvimento de sistemas parecem encaminhadas em sentidos opostos. Conseqüentemente modelar mecanismos de herança para satisfazer ambas as demandas, parece uma meta utópica. Porém, é possível considerar um ambiente de desenvolvimento, no qual possam ser combinados subconjuntos de linguagens com modalidades de herança favoráveis a diferentes etapas na evolução de um sistema. Podemos acompanhar a etapa inicial de desenvolvimento, por mecanismos de herança sem definição de padrões, nem verificação de tipos, e incrementar restrições, na medida que o sistema vai-se estabilizando. O trânsito de um modo de trabalho, a outro, auxiliado por ferramentas apropriadas, permitiria observar os objetivos de flexibilidade e confiabilidade sem entrar em colisão, desde que em fases diferentes na vida de um sistema.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[AGH86] - Agha Gul,"An Overview on actor languages". Sigplan Notices, October 1986.

[BLA89] - Blair Gordon, S., Gallayker John e Malik Javad. "Genericity vs Inheritance vs Delegation vs Conformance ...". JOOP. September/October 1989.

[CAR85] - Cardelli, L. e Wegner, P. "An Understanding Types, Data Abstractions and Polimorphism". computing Surveys, Vol 17, N4, Dec 1985

[CAR92] - Carvalho, Sergio. "Linguagem de programação Tool". Comunicação Pessoal, 1992.

[CLE88] - Clerici Silvia e Orejas Fernando. "GSBL: an Algebric Specification Language Based on Inheritance". ECCOP. Proceedings - Oslo, 1988.

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

[DAH68] - Dahl Ole, J., Myrhaug B., Nygaard J. "Simula 67. Common based language". Nowegian Computer Center, Oslo, 1968.

[DAN88] - Danforth, S. e Tomlinson, C. - "Type Theories and Object - Oriented Programming", ACM Computing Surveys 20(1), 1988.

[GOL83] - Golberg Adelle e Robson, D. "Smalltalk - 80: The Language and Its Implementation". Addison - Wesley, 1983.

[HAI87] - Hailpern Brent, Van Nguyen. "A Model for Object - Based Inheritance". IBM Watson Research Center. Technical Report. Mar, 1987.

[IER90] - Ierusalimschy Roberto. "O=M: Uma lingaugem orientada a objetos para desenvolvimento rigoroso de programas". Tese de Doutorado. Depto. Informática. Pontifícia Universidade Católica do Rio de Janeiro, setembro 1990.

[LAL86] - Lalonde Wilf R., Thomas e Pugh. "An exemplar based on Smalltalk". OOPSLA'1986 Proceedings.

[LIE86] - Lieberman, H. "Using prototipical objects to implement shared behavior in Object-Oriented Systems". OOPSLA'86 Proceedings.

[LIR88] - Lieberherr Karl e Riel, A. "Demeter: A case study of software growth through parametrized classes". 10th International Conference on Software Reusability, April 11-15, 1988 - Singapore.

[MAT90] - Matich Graciela H. "Modelagem de uma extensão da linguagem CLU para Orientação a Objetos". Dissertação de Tese de Mestrado. Pontifícia Universidade Católica do Rio de Janeiro, fevereiro 1990.

[MEY87] - Meyer Bertrand. "Object Oriented Software Construction". Prentice Hall, 1987.

[PAN87] - Panel P2: "Varieties of Inheritance". OOPSLA'87 Proceedings.

[PAI89] - Panel: "Inheritance: Can We Have Our Cake and Eat it Too?" OOPSLA'89 Proceedings.

[SCH86] - Schaffert Craig et alt. "An Introduction to Trellis/Owl". OOPSLA'86 Proceedings.

[SNY86] - Snyder Alan. "Encapsulation and Inheritance in object-oriented programming languages". OOPSLA'86 Conference Proceedings, Portland, Oregon, Set 1986.

[STE87] - Stein Lynn. "Delegation is Inheritance". OOPSLA'87 Proceedings.

[STR86] - Stroustrup, Bjarne. "The C++ Programming language". Addison-Wesley. MA, 1986.

[UNO87] - Ungar David e Smith Randall. "Self: The Power of Simplicity". OOPSLA'87 Proceedings.

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

[WEG88] - Wegner Peter e Zdonick Stanley. "Inheritance as an Incremental Modification Mechanism"or "What Like Is an Isn't Like". ECOOP'88.