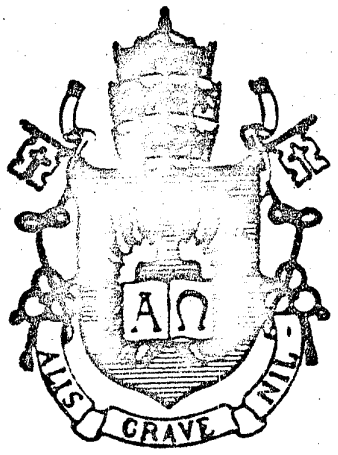


PUC

Série: Monografias em Ciência da Computação,
No. 13/91

UM MAPEAMENTO DE JSD PARA ORIENTAÇÃO A OBJETOS

Werther J. Vervloet

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, No. 13/91
Editor: Carlos J. P. Lucena Junho, 1991

UM MAPEAMENTO DE JSD PARA ORIENTAÇÃO A OBJETOS *

Werther J. Vervloet

* Trabalho patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República.

In charge of publications:

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

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

Abstract:

Object oriented programming models are becoming increaseably popular both in the academic and professional comunities. A mapping from the procedural model proposed by Michael Jackson (JSD) into an object oriented model is shown. The areas covered and the adequacy of both models are discussed.

Keywords:

Object oriented programming, programming models.

Resumo:

Modelos orientados a objetos (MOO's) vêm tendo grande aceitação tanto pela comunidade acadêmica quanto pelo mundo comercial. Este trabalho mostra um mapeamento do modelo procedural proposto por Michael Jackson (JSD) em um MOO. Discute também as abrangências dos dois modelos com algumas considerações sobre adequabilidade.

Palavras-chave:

Progamação orientada a objetos, modelos de programação.

SUMÁRIO

"Everything I say is merely an opinion of my own. The reader can take it or leave it. If he has the patience to read what follows he will see that there is only one thing about which I am certain, and this is that there is very little about which one can be certain."
William Somerset Maugham.

1.0 - Introdução.

Este trabalho descreve uma alternativa para se construir um modelo orientado a objetos (MOO), tendo como partida o modelo proposto por Michael Jackson JSD (JACKSON - 83). Não é uma metodologia, pois para tanto, seria necessário um detalhamento maior no que diz respeito à sua abrangência e escopo de utilização bem como uma definição mais rigorosa de cada um de seus passos, o que fugiria às dimensões deste artigo.

Por outro lado, estabelece as relações existentes entre o modelo de Jackson composto de entidades, suas estruturas, ações e funções e um modelo composto de classes, objetos e mensagens. Propõe finalmente um roteiro para este mapeamento.

As idéias aqui apresentadas são apenas parte de um projeto mais ambicioso: análise de sistemas usando múltiplos paradigmas, que se propõe a identificar qual ou quais os paradigmas mais adequados na representação de partes de um sistema, bem como a especificar as formas de comunicação entre estes.

Na vida real observamos que alguns idiomas são mais adequados para expressar determinados aspectos do cotidiano, como alguns sentimentos, relações ou objetos. Da mesma forma, percebemos que algumas linguagens de programação "expressam" melhor e mais naturalmente alguns problemas que outras. Como os diagramas e roteiros propostos pelas metodologias nada mais são que "linguagens" para representar os sistemas, não é difícil constatar que algumas metodologias mapeiam mais naturalmente alguns sistemas que outras.

O que se observa na maioria das metodologias existentes, é uma tentativa de estender ao máximo seu escopo de aplicação e a justificativa (aceitável) é a padronização. Entretanto, há casos em que a especificação fica bastante "carregada" (quando não confusa) e neste caso, podemos concluir que: ou o usuário não domina a técnica, ou esta não é adequada para representar

aquele problema específico. Acreditamos serem estas as explicações para as preferências nítidas de alguns grupos a respeito das metodologias a usar.

Por tudo que foi exposto, concluímos que a passagem de um modelo para outro pode eventualmente ser bastante traumática. Acreditamos entretanto que não é este nosso caso.

A definição de objeto tal como é ensinada hoje, é uma evolução do conceito de tipo abstrato de dados ao qual se atribuiu um comportamento. Tipos abstratos de dados estão presentes na maioria das metodologias que usam o paradigma procedural, "disfarçados" em entidades externas, entidades, agentes etc. Em algumas destas metodologias seus comportamentos (processos) são definidos externamente (como em Análise Estruturada GANE - 77), em outras internamente (como em JSD).

Nosso objetivo principal é "aproveitar" a modelagem existente nos primeiros 4 passos de JSD e obter uma especificação do sistema usando classes e objetos.

No Capítulo 2 apresentamos uma breve descrição de JSD com considerações sobre os pontos comuns com elementos de MOO's.

No capítulo 3 apresentamos uma breve descrição das metáforas usadas em MOO's. Para tanto, usamos como base o modelo adotado em TOOL (ver [CARVALHO - 91], [COELHO - 91] e [VERVLOET - 91]). É importante lembrar que, ao falarmos de TOOL, estamos falando das abstrações que nos levaram a implementar este ou aquele mecanismo e não da linguagem propriamente dita, afinal, JSD é um método para desenvolvimento de sistemas e TOOL uma linguagem de programação.

No capítulo 4 mostramos o mapeamento das entidades, ações e funções em classes, objetos, mensagens e métodos com observações sobre herança, polimorfismo e sincronismo.

O capítulo 5 conclui este trabalho.

2.0 - Inspecionando JSD.

2.1 - Motivação.

As razões que nos levaram a escolher o JSD como ponto de partida para exercitar nossa tese de que MOO's são adequados para modelar sistemas que, por sua vez, podem ser modelados por metodologias que usam o paradigma procedural se basearam nos seguintes aspectos:

- Jackson, no passo I de sua metodologia, define ***Entidades*** como todo componente de um sistema no mundo real capaz de "gerar" ou "sofrer" ações. Existem "tipos" de entidades que representam o conjunto dos componentes do sistema ("instâncias") que partilham das mesmas ações. Isto nos pareceu bastante semelhante ao conceito de classes e objetos.
- ***Ações***, em sua definição, são eventos que devem ser tratados pelas entidades às quais afetam. Lembramos que em MOO's objetos trocam mensagens entre si.
- Diferentemente das metodologias de análise e projeto estruturados ([GANE - 77], [DEMARCO - 79] e [YOURDON - 78]), JSD não é Top-Down. Muitos aspectos de modelagem em orientação a objetos são essencialmente Bottom-Up, pois partem da construção de componentes para obter objetos mais complexos.
- Acresça-se o fato de que JSD já nos era familiar e temos uma "boa" (pelo menos assim nos pareceu) justificativa para a escolha.

2.2 - Uma breve introdução ao JSD.

JSD é, nas palavras de seu autor, "um método para especificar e implementar sistemas em computador" que leva o usuário desde os passos de especificação até a implementação, tentando com isso resolver o problema de interface entre o que se convencionou chamar de "projeto lógico" e "projeto físico" existentes nas metodologias acima citadas.

Inicia fazendo uma inspeção do que chama de "Mundo Real" estabelecendo a fronteira entre este e o sistema que se quer modelar. Construir um modelo JSD do mundo real envolve duas tarefas distintas:

- Criar uma descrição abstrata do mundo real.
- Montar, no computador, uma representação desta descrição.

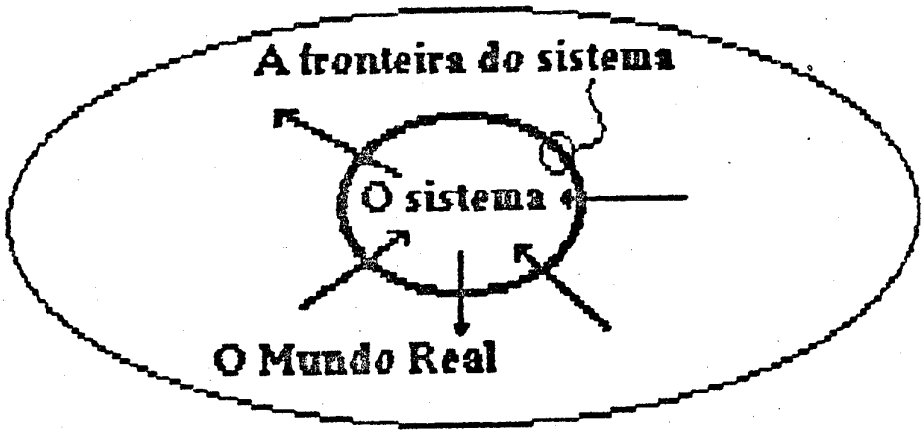


Fig. 1 - O "mundo real" e o sistema em JSD.

Está dividido em 6 etapas chamadas "passos", das quais as 4 primeiras tratam basicamente de especificação, deixando a implementação para as duas últimas. Estas etapas são:

- i - *Entity-Action step*. A partir de uma inspeção do "Mundo Real" afeto àquela aplicação, são identificadas as entidades e ações que devem constar do modelo.
- ii - *Entity-Structure step*. À cada entidade é associada uma estrutura representando um processo. Este processo indica os eventos (ações) possíveis para esta entidade e descreve a sequência em que podem ocorrer e ser tratados.
- iii - *Initial Model step*. São estabelecidas as ligações entre as entidades mostrando as comunicações possíveis. Estas ligações podem estar representadas por mensagens ou por áreas partilhadas.
- iv - *Function step*. Adiciona ao modelo as funções necessárias para produzir informações específicas, sempre que uma certa combinação de eventos acontece.

- v - *System Timing step*. Analisa as respostas do sistema sob a dimensão tempo. Define as condições de contorno do que pode ser considerado aceitável sob este aspecto. Produz uma série de restrições a serem usadas como premissas pelo passo seguinte na definição do modelo de implementação.
- vi - *Implementation step*. O modelo multiprocesso definido nos 4 primeiros passos é então adequado ao hardware disponível, respeitadas as restrições de contorno.

2.3 - O "Entity-Action step".

Da observação do "Mundo Real" são identificados os eventos e os agentes que os produzem ou são afetados por eles. Em JSD, estes agentes são as *Entidades* e os eventos, as *Ações*. Outra definição é que entidades são caracterizadas pelas ações que "produzem" ou "sofrem" e a ordem em que estas podem ocorrer. Este passo produz como saídas as listas de entidades e ações do sistema.

Um ponto de interessante semelhança com MOO's é a distinção explícita entre os conceitos de "tipo" e "instância" aplicados por Jackson a entidades e ações. Tipos de entidades representam os conjuntos de entidades de mesmas características i.e. às quais se aplicam o mesmo conjunto e mesma sequência possível de ações. Tipo de ação é um evento aplicável ao conjunto de entidades de mesmo tipo. Instância de uma ação é um evento "produzido" ou "sofrido" por uma instância particular de entidade.

☐ *Em MOO's este universo pode ser naturalmente mapeado em classes e objetos (tipos e intâncias de entidades) e em mensagens (tipos de ações). Relacionamos instância de uma ação a uma mensagem enviada ou recebida por um determinado objeto. (ver figura 2)*

Uma entidade em JSD deve:

- "Produzir" ou "sofrer" ações numa sequência conhecida.
- Existir no mundo real e não apenas dentro do sistema.
- Ser individual, podendo ser identificada entre as demais entidades de mesmo tipo.

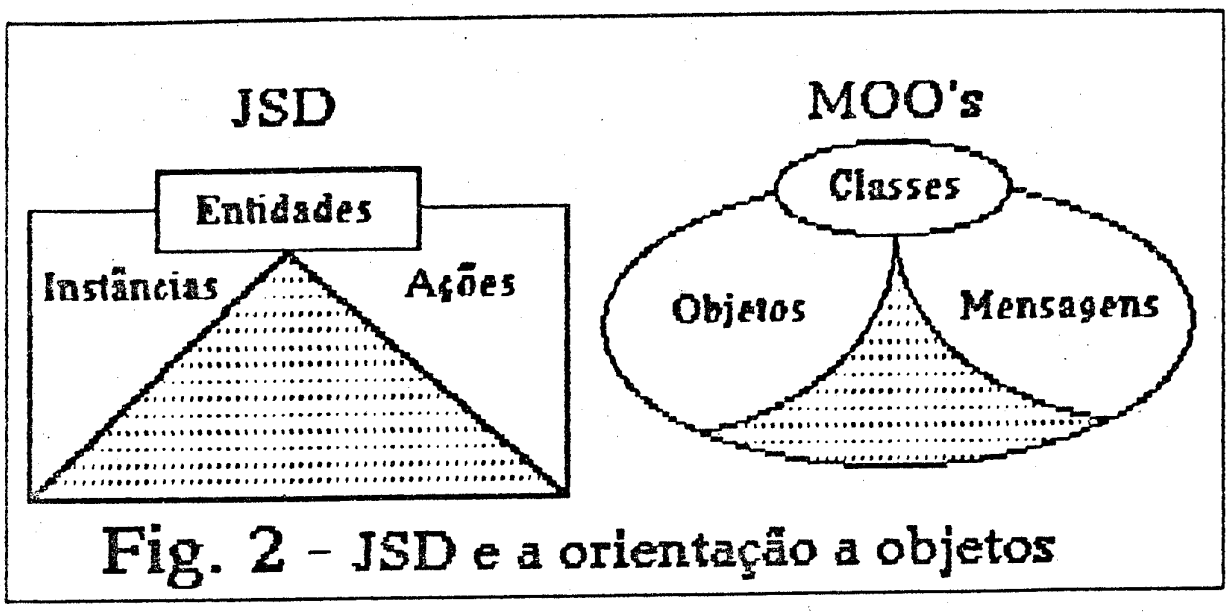


Fig. 2 - JSD e a orientação a objetos

Uma ação em JSD deve:

- Ser vista como acontecendo num particular instante e não atuar durante um período de tempo.
- Ser ação do mundo real e não uma ação particular do sistema propriamente dito.
- Ser vista como atômica, não devendo ser dividida em sub-ações.

Baseado nestas assertivas, estabelece as diferenças entre ações, estados e funções. Ações são eventos instantâneos que ocorrem no mundo real. Estados são características que perduram e podem eventualmente modificar o comportamento da entidade por um espaço de tempo. Funções são respostas do sistema a eventos ou combinação destes.

2.4 - O "Entity Structure step".

Neste passo, a cada entidade é associada uma estrutura representando a sequência em que as ações podem ocorrer. Aqui a representação gráfica das estruturas segue os mesmos conceitos apresentados em JSP (Jackson Structured Programming [JACKSON - 75]). As entidades devem ter suas estruturas especificadas levando em consideração seu ciclo completo de vida no mundo real. Desta forma, entidades que podem ser criadas e destruídas várias vezes, devem ter este aspecto mostrado em sua estrutura.

☐ *É o caso do aluno de uma universidade que, algum tempo após ter concluído seu curso, volta para cursar outro e o regulamento prevê que o sistema deve levar em consideração seu histórico escolar anterior na tomada de decisões sobre matrícula em disciplinas, valor da mensalidade, concessão de bolsas etc.*

Na descrição das estruturas das entidades, pode-se eventualmente descobrir que uma entidade, na realidade, representa melhor o mundo real se for desmembrada em duas ou mais.

☐ *Para este aspecto podemos citar o aluno matriculado em mais de um curso no mesmo estabelecimento de ensino, o cliente de um banco com mais de uma conta etc. Nestes casos deve-se dissociar a entidade* ***matrícula*** *da entidade* ***aluno*** *e a entidade* ***conta*** *da entidade* ***correntista****. Jackson chama estas novas entidades identificadas neste passo de* ***Entidades Marsupiais****.*

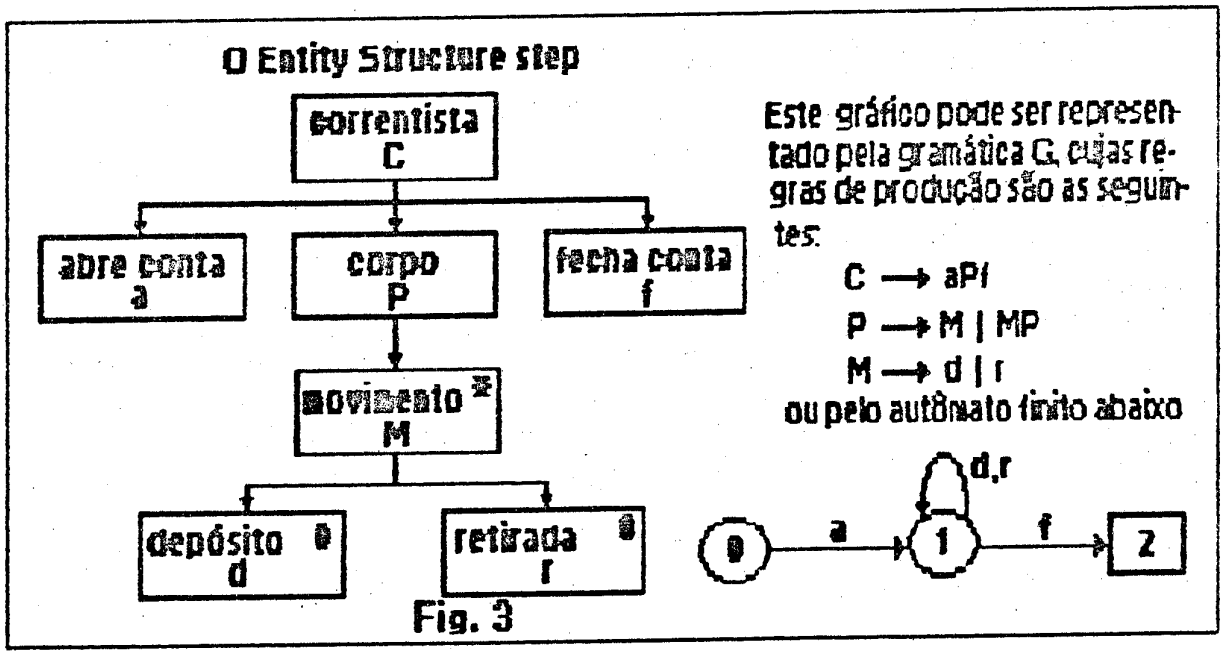


Fig. 3

As estruturas básicas apresentadas aqui são a sequência, a iteração e a seleção. Esta notação é apenas uma forma gráfica de se representar reconhecedores equivalentes a autômatos finitos (ver figura 3).

No desenvolvimento de software básico podemos "aproveitar" a metáfora de Jackson e descrever, por exemplo, o que seria o "mundo real" na tradução (compilação) de programas. Certamente uma das primeiras entidades

identificadas seria o programa fonte. Programas declaram, referenciam e desativam unidades, variáveis, constantes e pragmas.

A descrição da sequência possível das ações como requisito básico do modelo não tem um mecanismo obrigatório equivalente em MOO's, mas pode ser mapeada através de consultas às variáveis de estado pelos tratadores de eventos. (ver 2.5 e 3.4)

2.5 - O "Initial Model step".

Neste passo, Jackson começa efetivamente a construir o sistema especificando para cada entidade um conjunto de processos sequenciais que representam seus comportamentos. É neste ponto que, para que o modelo reflita corretamente o mundo real, são estabelecidas as comunicações entre as entidades, vistas agora como processos.

Existem dois tipos de comunicações abordados em JSD:

- comunicação via "data stream".
- comunicação via "vetor de estados".

Na primeira, um processo grava um conjunto ordenado de mensagens e dados que serão lidos pelo outro processo. Na segunda, um processo tem, de alguma forma, a possibilidade de "ver" o vetor de estados do outro. Este vetor de estados é o conjunto das variáveis locais internas ao processo.

> ☐ *Pensando em MOO's, mais uma notável semelhança se apresenta. Estas comunicações descritas acima, mapeiam naturalmente as mensagens e os blocos partilhados entre entidades. O vetor de estados, pela própria definição, pode perfeitamente ser visto como a representação do objeto. Neste nível de abstração, JSD ainda não menciona de que maneira um processo avisa o outro de que há mensagem para ser tratada, adiando esta definição para a etapa de implementação.*

Esta etapa produz como saída os diagramas chamados SSD (system specification diagrams) mostrando estas conexões.

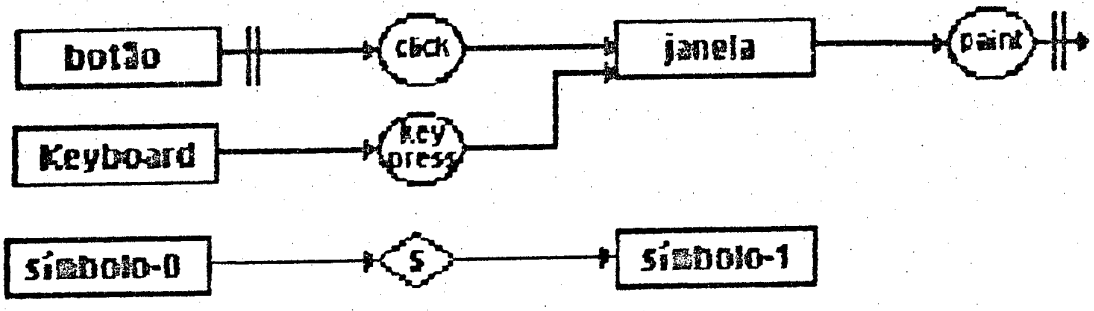


Fig. 4 - O System Specification Diagram (SSD)

☐ *Este tipo de diagrama provê a visão dos relacionamentos entre as entidades mostrando, além da forma de comunicação, os tipos de relacionamentos (1 para n, n para m etc) entre elas. Supre, de uma forma elegante, uma lacuna deixada pelas Análises Estruturadas quanto a este particular, fazendo uma breve incursão no modelo proposto por Peter Chen. [CHEN - 76].*

2.6 - O "Function step".

Neste passo são definidas o que Jackson chama de ***Funções do Sistema.*** São elas as responsáveis por executar as tarefas decorrentes de um evento (ou uma combinação destes), uma mudança de estado ou até mesmo um pedido explícito (que pode ser perfeitamente entendido como um evento). O propósito desta etapa é dar ao sistema a capacidade de produzir saídas.

Na realidade, o que Jackson define como funções do sistema são processos (e não procedimentos) capazes de ser instanciados, enviar e receber mensagens, enfim, semelhantes às entidades a menos de suas existências no mundo real. Estas funções podem ser embutidas, normais ou impostas.

Funções ***embutidas*** são aquelas que dependem exclusivamente de uma única instância de entidade para executar. São acionadas por esta em função de alguma combinação de condições e dispõem de todas as informações necessárias para sua execução.

Funções são ditas ***impostas*** quando sua conexão com uma entidade se faz através do vetor de estados e ***normais*** quando a conexão se faz via "data

stream" (mensagem). Usa também o atributo ***simples*** para indicar que a função se conecta com apenas uma instância de entidade.

- ☐ *MOO's possuem métodos, definidos nas classes, capazes de executar as tarefas decorrentes do recebimento de mensagens ou da mudança do valor de uma variável na representação do objeto. Observem que JSD ainda não trabou do aspecto sincronismo. A linguagem TOOL [CARVALHO - 91] possui a este respeito mecanismos específicos para tratar eventos síncronos e assíncronos (ver 3.4 e 3.5).*

Esta etapa produz como saídas "explosões" no SSD mostrando as funções inseridas no sistema.

Os próximos dois passos tratam da adequação do modelo especificado às restrições impostas pela dimensão tempo e ao hardware disponível. Embora extremamente relevantes como parte da metodologia, já se situam além da fronteira do nosso "mundo real" de mapeamento JSD x MOO's. Todos os conceitos neles apresentados valem também para a implementação de nossos modelos, sem contudo influir mais em sua forma. Faremos deles então apenas uma breve descrição.

2.7 - O "System Timing step".

Neste passo são feitas as considerações sobre tempo de resposta e sincronismo. Como saída, são produzidas especificações para o passo seguinte com as restrições do sistema em relação a estes aspectos. Pontos de sincronismo entre processos são acrescentados ao SSD.

2.8 - O "Implementation step".

Neste último passo JSD faz as considerações sobre o hardware, principalmente no que diz respeito ao número de processadores disponíveis. Toda ênfase desta etapa é dada a como partilhar muitos processos entre os poucos processadores (que podem ser apenas um) de que normalmente se dispõe.

3.0 - Usando as metáforas de MOO's como universo a ser mapeado.

Este capítulo trata dos conceitos usados em orientação a objetos. Não pretende exaurir o assunto, apenas apresentar o outro lado da fronteira, isto é, os aspectos de MOO's nos quais se mapeiam as abstrações de JSD já apresentadas no capítulo anterior.

Para tanto, usamos como base o modelo adotado na implementação da linguagem TOOL que, ao nosso ver, contempla a maioria destes conceitos.

3.1 - Classes, Classes Estendidas e Objetos.

O conceito básico em orientação a objetos é a ***classe***. Classes em MOO's atuam como construtores de instâncias chamadas ***objetos***.

Uma classe define uma estrutura de dados chamada ***representação*** (REP) e uma coleção de procedimentos chamados ***métodos*** que modelam o comportamento de seus objetos. Podem eventualmente definir uma única estrutura de dados comum a todos os seus objetos. Esta estrutura, normalmente conhecida por ***área compartilhada***, serve a dois propósitos principais: estabelece um meio de comunicação entre seus objetos e tira a classe da posição de mero construtor, concedendo-lhe um status de super-objeto possibilitando, de uma certa forma, que a classe tenha "consciência" da existência de suas instâncias.

- ☐ *Em MOO's a forma convencional de comunicação entre objetos é a mensagem, que pode ser vista adiante em 3.4.*

No modelo que adotamos para definir TOOL, classes estão divididas em dois grupos: as classes convencionais ou simplesmente ***classes*** e as classes estendidas ou ***xclasses***. As razões desta classificação podem ser vistas em [VERVLOET - 91].

Neste modelo, ao conceito de classe está associada a idéia de sincronismo. A forma de se comunicar com um objeto destas classes implica em se aplicar a estes um método, usando o clássico mecanismo de "call/return". Objetos de

classes convencionais neste modelo são, pois, entidades passivas, pertencentes ao processo que as instanciam.

Xclasses, ao contrário, envolvem a idéia de processos que, uma vez instanciados, têm vida própria. Objetos de xclasses podem usar tanto a comunicação síncrona descrita acima, como a assíncrona através de ***mensagens***. Assim sendo, xclasses devem também definir o conjunto de mensagens que podem ser recebidas e os procedimentos que as tratam. A estes procedimentos em particular chamamos de ***tratadores de eventos***.

Um conceito intrinsicamente ligado a MOO's é o ***encapsulamento.*** Encapsular significa "esconder" a descrição interna e o funcionamento de uma abstração, deixando visível apenas o mínimo indispensável para se estabelecer a comunicação. Objetos assim construídos podem ser usados como "caixas_pretas". Apesar de ser reconhecido como positivo, este aspecto ainda suscita divergências entre vários autores. Alguns modelos propostos são absolutamente fechados, como em POOL2 [AMERICA - 89], enquanto que outros já são bem abertos, como em [BUHR - 88].

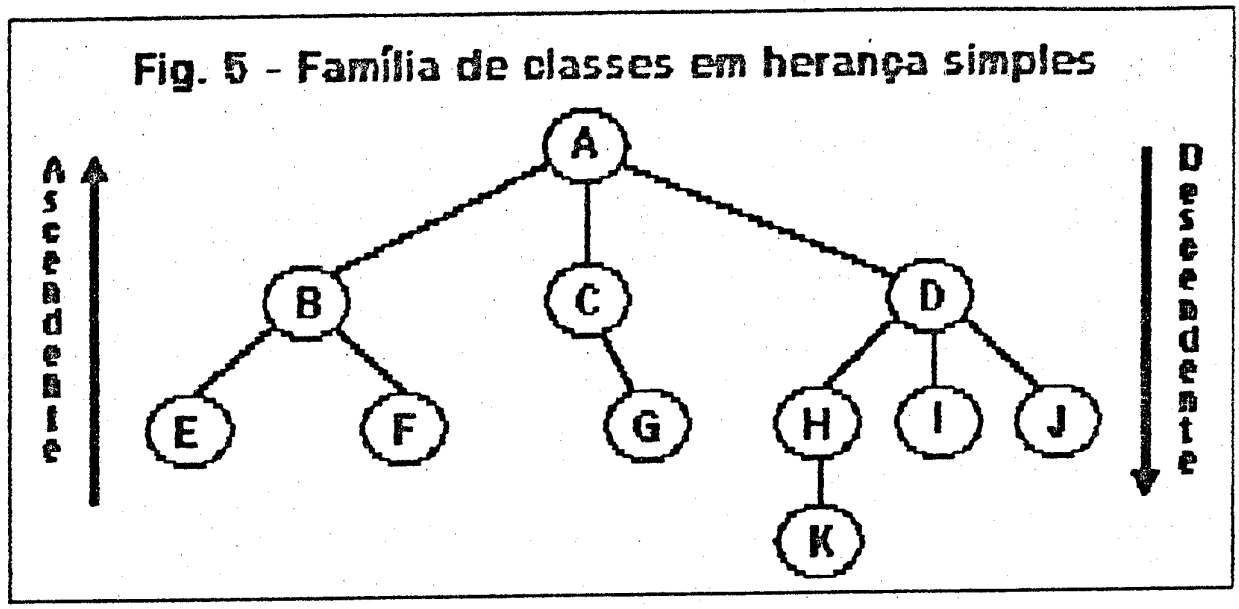
No modelo por nós adotado, a visibilidade das "entranhas" dos objetos é decidida pelo usuário, contemplando com isso as diversas correntes. Da mesma forma, parte do comportamento dos objetos pode também ser definida como de uso exclusivo da classe.

3.2 - Herança.

Outro conceito fortemente ligado a MOO's é a ***reusabilidade***. A idéia sugere que se tenha um mecanismo para reaproveitar software melhor do que o tradicional "corte e colagem". A herança é um mecanismo que permite, ao se projetar uma classe, "importar" as definições de outra classe definindo apenas o que deve ser acrescentado. Uma classe assim construída é vista como uma "especialização" ou subclasse da outra. Herança nos traz as seguintes relações:

- superclasse
- subclasse
- ascendente
- descendente

☐ ***Definição.***



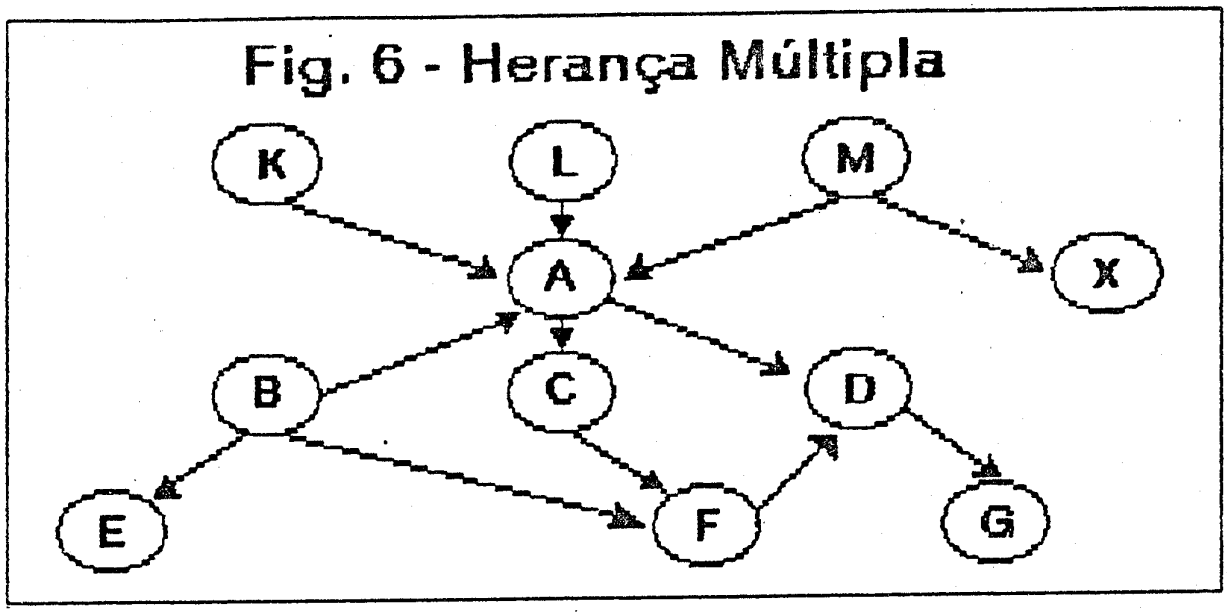
Sejam A e B duas classes onde Ra e Rb são suas representações e Ca e Cb seus conjuntos de métodos respectivamente.

- Se B é subclasse de A, SUB (B, A), então A é superclasse de B, SUPER (A, B).
- SUB (A,B) implica em REPb e Cb conterem propriamente REPa e Ca respectivamente.
- Diz-se que uma classe B descende de A, DESCENDENTE (B, A) se SUB (B, A) ou se existe uma classe X tal que SUB (B, X) e DESCENDENTE (X, A).
- A relação DESCENDENTE (B, A) implica em A ser ascendente de B, ASCENDENTE (A, B).
- Damos o nome de família de classes de A à união de A com o conjunto de suas descendentes. Nesta família, A é vista como a classe raiz.

Até aqui falamos apenas de herança de uma única classe. A este tipo de herança chamamos herança simples. Quando construímos uma classe herdando de mais de uma, temos o que chamamos de herança múltipla. Os conceitos até aqui apresentados de super e subclasses, ascendência, descendência e família também se aplicam.

☐ *Ao definirmos TOOL em sua primeira versão, retiramos temporariamente de nosso modelo este aspecto. Esta decisão se deveu a uma série de razões, especialmente a da eficiência. Como se pode observar nas figuras 5 e 6, ao passarmos de herança simples para múltipla, mudamos de árvores para grafos acíclicos dirigidos (DAG's).*

Consequentemente a complexidade dos algoritmos para navegação e verificação de integridade aumenta consideravelmente, podendo impactar fortemente no tempo de compilação.



Uma das consequências da herança é que um objeto de uma subclasse não perde as características de suas classes ascendentes, podendo ser eventualmente visto como objeto de qualquer delas.

☐ *Como exemplo podemos citar as classes Rádio e Rádio-de-Pilha. Rádio-de-Pilha é uma especialização de Rádio. O fato de um objeto ser um Rádio-de-Pilha não o faz deixar de ser um Rádio.*

3.3 - Composição.

Outra forma de se construir classes reusando definições existentes é através da composição. Este mecanismo consiste em definir, na representação de uma classe, objetos de outra.

Composição difere fundamentalmente de herança múltipla em seu significado. Enquanto um carro anfíbio pode ser visto como uma herança de carro e barco, um trem deve ser visto como uma composição de locomotiva e vagões. Uma maneira prática de se decidir entre estes dois mecanismos é fazer a descrição do objeto através dos verbos "ser" e "ter". Se um objeto "é" várias coisas ao mesmo tempo dizemos que ele é produto de herança

múltipla, se possui ou "tem" vários componentes ele é uma composição destes.

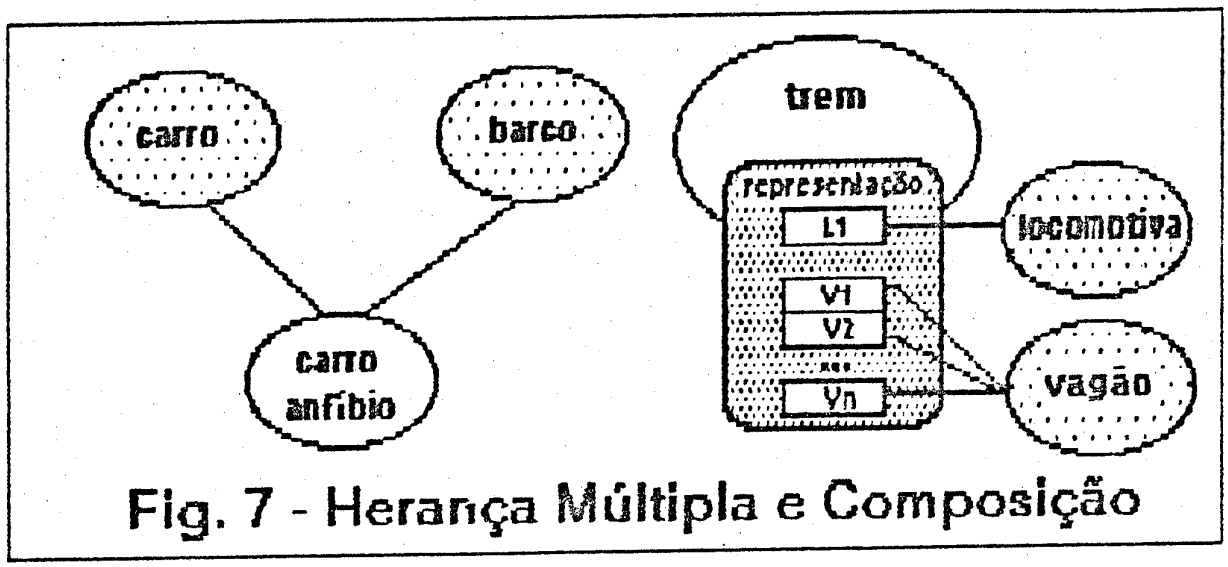


Fig. 7 - Herança Múltipla e Composição

3.4 - Mensagens, Métodos e Tratadores de Eventos.

Mensagens, como já foi dito, são a forma tradicional em MOO's de se estabelecer comunicação entre objetos. Atuam como "avisos" sinalizando a ocorrência de um evento. Por sua própria natureza, são compostas de duas partes. Uma identifica a mensagem propriamente dita, a outra, opcional, "carrega" informações adicionais.

> ☐ *Em ambiente Windows (Microsoft Corporation, ver [PETZOLD - 88]) apertar o botão do "mouse" sobre um objeto gráfico qualquer faz com que o sistema envie ao processo que possui o objeto uma mensagem, informando que o mouse foi "clicado" e a posição em que se encontrava no momento do "clic".*

Associados às mensagens estão os procedimentos que devem ser acionados para tratá-las. Estes procedimentos, mais conhecidos como ***tratadores de eventos*** ou ***"event handlers"*** são os responsáveis pelas ações que se espera de um objeto quando a ele enviamos uma mensagem.

Ao enviar uma mensagem, um processo pode querer fazê-lo de duas formas: síncrona e assincronamente. Ao analisarmos eventos no mundo real, verificamos que alguns devem ser tratados antes do prosseguimento de uma

determinada tarefa, outros podem ser apenas sinalizados para tratamento posterior. No modelo adotado em TOOL, mapeamos eventos síncronos às chamadas de métodos. Eventos assíncronos são mapeados em mensagens, colocadas na fila de mensagens do sistema, que oportunamente serão recebidas e tratadas por seus destinatários.

MOO's possuem um mecanismo que permite modelar parcialmente os objetos de uma classe. Consiste em se criar uma definição onde apenas parte do comportamento dos objetos está definida. Isto se faz através da definição de apenas um subconjunto (que pode ser vazio) dos métodos e tratadores de eventos, adiando explicitamente a definição dos demais. Este adiamento obriga as subclasses desta a definí-los ou novamente adiá-los para sua descendência. A estes métodos chamamos de adiados ou virtuais.

☐ *Podemos citar, como exemplo, uma classe* ***veículo*** *que, por estar num nível de abstração ainda muito genérico, não possui meios de definir o método* ***mover*** *adequadamente, adiando a definição deste para sua descendência. Assim, as classes* ***avião*** *e* ***navio****, especializações de* ***veículo****, devem definir seu próprio método* ***mover****.*

3.5 - Polimorfismo.

A palavra polimorfismo se origina do grego *polis* (muitos) e *morphos* (forma). Ser polimórfico significa então, ter várias formas.

Esta propriedade permite a um objeto "mudar de classe" após sua criação. Sua abrangência pode ter ou não restrições. Esta abrangência, como o encapsulamento, é também objeto de divergência entre os autores. Não é nossa intenção neste trabalho advogar a causa deste ou daquele modelo de polimorfismo. As dificuldades adicionais causadas por este mecanismo em suas formas mais livres podem ser vistas em [VERVLOET - 91].

No modelo adotado em TOOL, objetos polimórficos podem apenas "transitar" pelas classes componentes da família de sua classe de definição e é este o modelo que vamos usar.

☐ *Um bom exemplo de aplicação de polimorfismo está presente em alguns jogos feitos para computador, onde o participante começa conduzindo um boneco que, à medida que cumpre determinadas tarefas, vai adquirindo poderes especiais e pode até mudar sua forma gráfica.*

*Esta "aquisição" de poderes especiais pode ser naturalmente representada pelo trânsito de um objeto polimórfico da classe **boneco** para a classe **guerreiro**, que é, no caso, uma especialização de **boneco**.*

JSD, por não tratar suas entidades como hierarquia, não possui representação especifica para este aspecto. Para modelar entidades que, no mundo real, mudam efetivamente de forma, apresenta duas alternativas.

- Ter apenas uma entidade no modelo contendo em seu vetor de estados as informações sobre a situação corrente da instância. Isto significa que esta entidade deve englobar a representação de toda uma família de classes de MOO's. A estrutura obtida no *"Entity-Structure step"* deve espelhar todas as sequências possíveis de transições como pode ser visto na fig. 8.1.
- Definir mais de uma entidade no modelo e tratar as várias formas como intâncias de entidades diferentes. Isto pode trazer alguma dificuldade se um dos requisitos do sistema for produzir informações sobre esta entidade ao longo de suas diferentes mudanças de forma (fig 8.2).

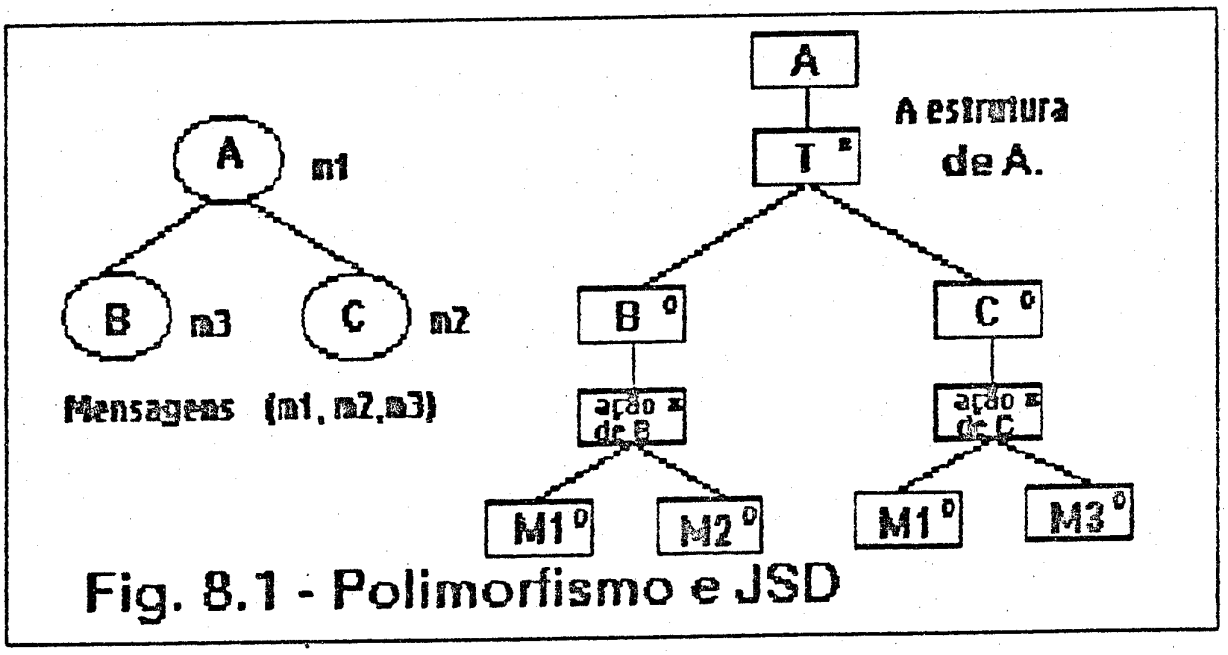


Fig. 8.1 - Polimorfismo e JSD

As figuras 8.1 e 8.2 mostram as duas alternativas acima citadas. No mundo real e em MOO's temos a classe A da qual descendem as classes B e C. Uma instância de A que pode, ao longo do ciclo de vida do sistema, se transformar em instância de B ou C, deve ser representada em JSD de uma das duas formas apresentadas.

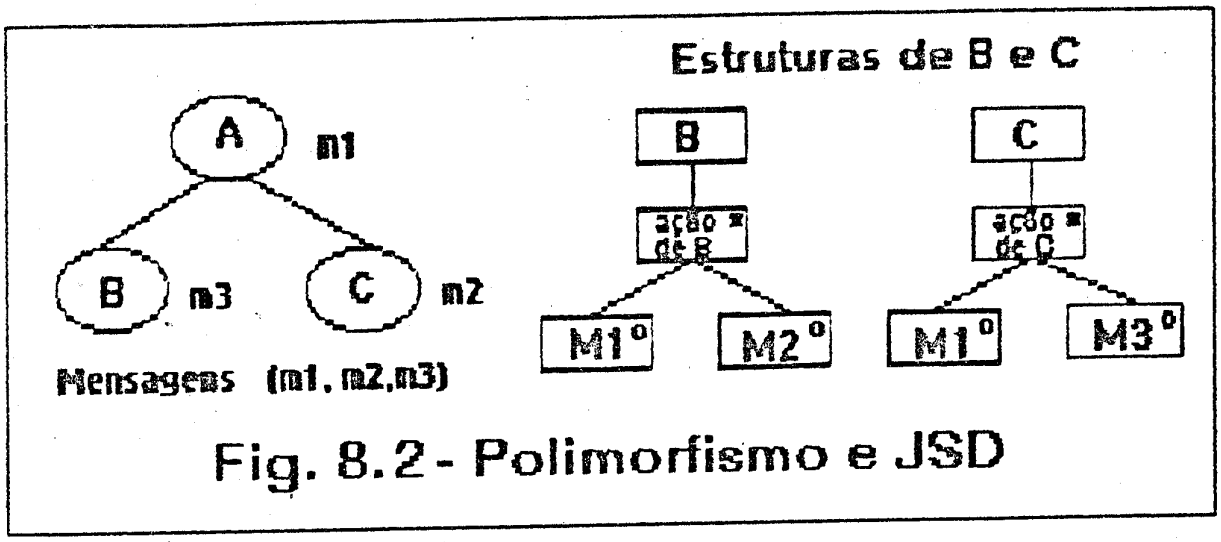


Fig. 8.2 - Polimorfismo e JSD

4.0 - Mapeando JSD em MOO.

Ao longo deste trabalho fizemos uma apresentação destes dois modelos e, sempre que oportuno, chamamos a atenção do leitor para pontos de flagrante semelhança entre conceitos e componentes. Nosso objetivo neste capítulo é mostrar como passar do modelo JSD para MOO.

Como foi visto, cada passo de JSD produz suas saídas. Estas, agora, são as entradas de nosso modelo. Vários exemplos são apresentados ao longo deste capítulo. Estes exemplos são, em sua maioria, bastante semelhantes aos usados por Jackson em [JACKSON - 83], atendendo a dois propósitos:

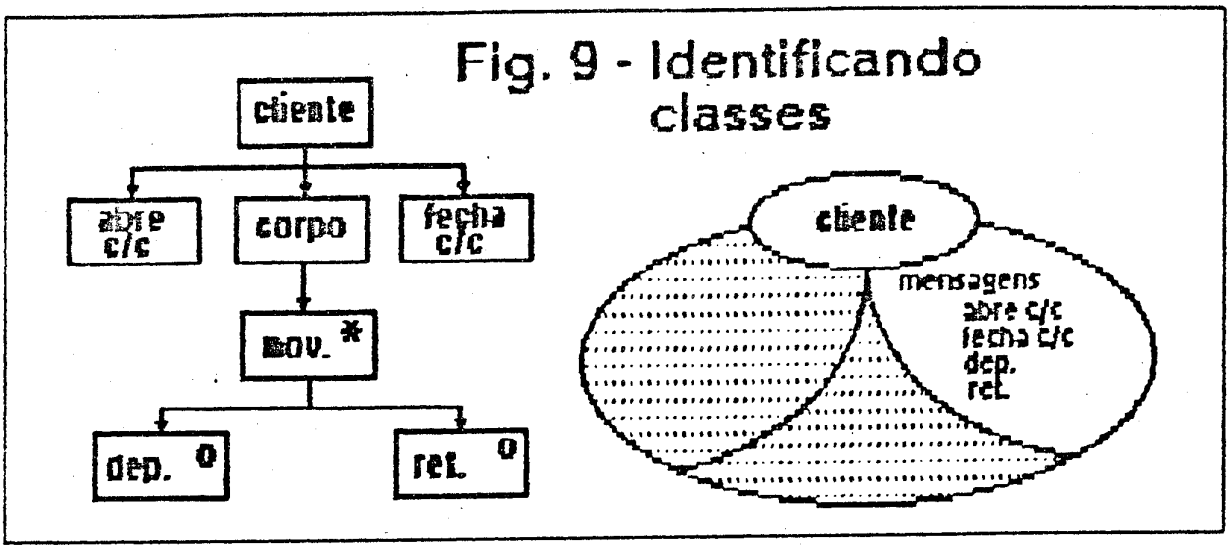
- Garantir que o modelo JSD usado como base está livre de "imperfeições".
- Simplificar consideravelmente nossa tarefa.

4.1 - Identificando as classes.

Esta seção trata principalmente da identificação das classes que vão compor nosso MOO. Como em JSD, as entidades e a maioria das funções estão ligadas à idéia de processos, o natural é associá-las às nossas classes estentidas (*xclasses*) e isto é precisamente o que estaremos fazendo. Entretanto, para não estar sempre repetindo que as classes das quais estamos tratando são estendidas, usamos apenas *classes*, fazendo ressalva sempre que se tratar de classes normais ou passivas (3.1). Estamos também relacionando as ações e os "data streams" das comunicações entre processos (2.5 - Initial Model step) às mensagens de MOO.

Iniciamos a construção do MOO, definindo uma classe para cada entidade identificada nos dois primeiros passos de JSD. A estas classes são então associadas as mensagens (ações) conforme descritas no Entity-Structure step.

☐ *Observem que em JSD as ações identificadas no passo 1 representam sempre mensagens recebidas pela entidade, mapeando eventos ocorridos no mundo real que devem ser tratados.*



4.1.1 - Adicionando mensagens às classes.

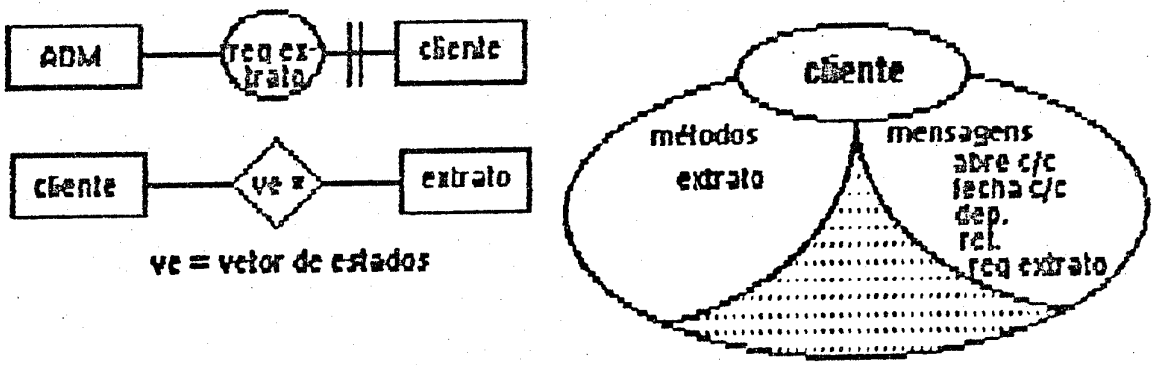
Para completar o conjunto das mensagens que devem ser tratadas pelas classes, é preciso inspecionar os SSD resultantes do Initial Model step. Para cada conexão via data stream entre dois processos, onde o receptor é uma entidade, deve-se adicionar, caso não esteja presente, este "data stream" à lista de mensagens da classe correspondente. Não é demais lembrar que, ao completarmos a lista das mensagens a serem tratadas pelas classes, estamos também identificando os tratadores de eventos que estas devem ter.

☐ *Imagine que no mundo real da figura 9 não existe o evento "pedir extrato". A emissão deste se faz através de uma função embutida acionada toda vez que um dos tratadores dos eventos depósito ou retirada constata que o contador de lançamentos chegou a 10. Entretanto existe um processo representando a administração do banco (Função) que pode eventualmente fazer este pedido através de uma mensagem ("data stream"). Assim, deve haver no SSD esta comunicação representada, a qual deve ser incluída na lista de mensagens da classe* ***cliente****. Neste caso temos um evento que pode ser tratado síncrona ou assincronamente.*

4.1.2 - Identificando os métodos.

Métodos em nosso MOO são procedimentos que devem ser executados sincronamente pela classe. Identificamos nestes as funções *embutidas* do Function step, pelas caracteristicas de sincronismo que possuem. Assim sendo, adicionamos à classe um método para cada função embutida da sua entidade correspondente.

Fig. 10 - Adicionando mensagens e métodos.

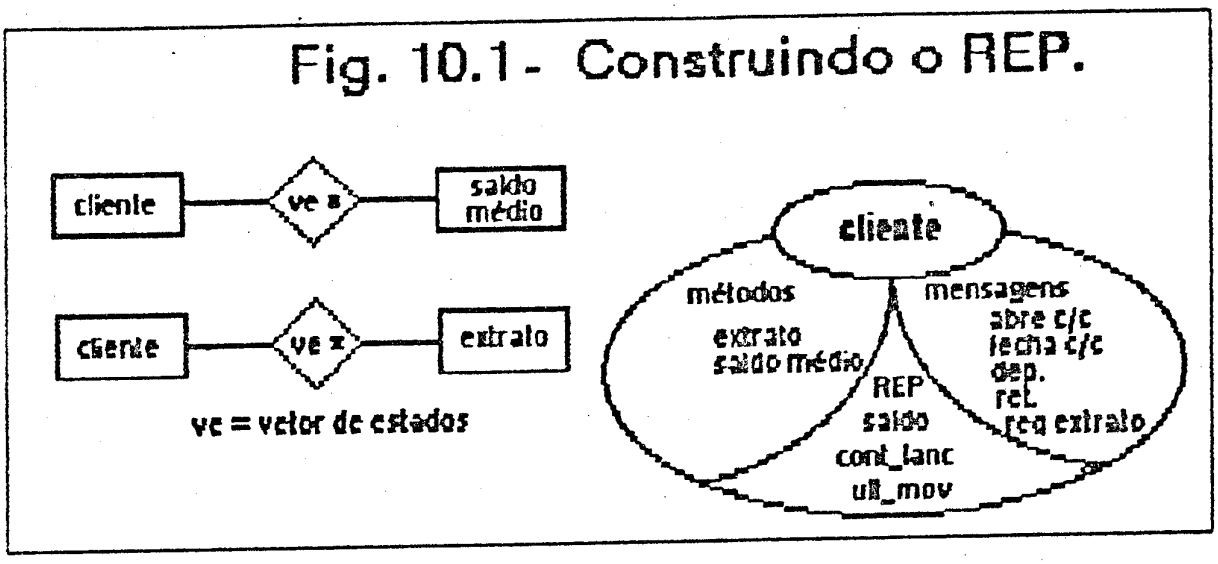


4.1.3 - Construindo a representação das classes.

Ao apresentarmos no cap. 3 os componentes de um MOO, relacionamos os vetores de estado aos REP's das classes. A semelhança é grande, entretanto, como podemos observar, classes são conceitos mais abrangentes que as entidades de JSD, por isso o que vamos propor a seguir é apenas um primeiro passo na direção de construir o REP das classes.

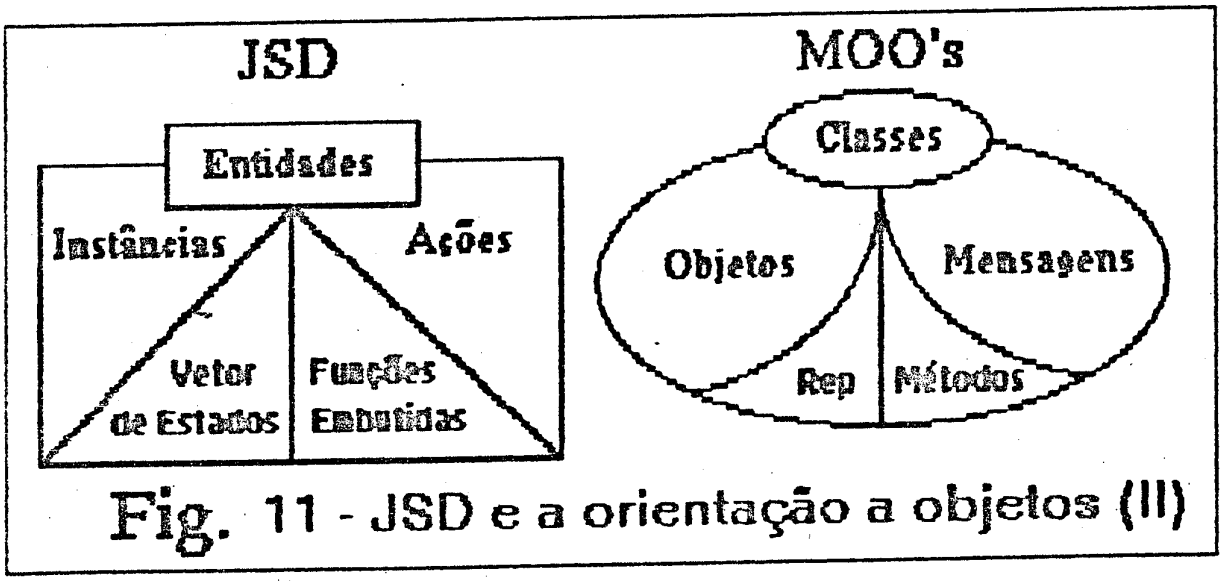
Em JSD os vetores de estado são sempre tratados *"en bloc"*, com total visibilidade por parte dos processos que os inspecionam. Isso vai contra uma das metas pretendidas pela orientação a objetos que é o encapsulamento. À medida em que vamos introduzindo herança, composição e polimorfismo, modificamos os REP's fazendo sugestões de como defini-los mais adequadamente para MOO's.

Para a primeira versão do REP de uma classe, partimos do vetor de estados de sua entidade correspondente. A este acrescentamos os vetores de estado de suas funções embutidas.



4.1.4 - Completando o conjunto de classes.

Para completar as classes do modelo falta-nos apenas inspecionar a saída do Function step. Lembrando que, com exceção das embutidas, as funções de JSD são, na realidade, processos, devemos incluir uma classe para cada uma destas.



4.2 - Introduzindo Herança.

Neste momento já temos um modelo preliminar orientado a objetos equivalente ao modelo base em JSD. Todos os componentes do modelo JSD estão aqui representados sob a forma de classes, métodos, mensagens, tratadores de eventos etc. Entretanto falta-nos explorar um pouco mais a potencialidade de MOO, com respeito aos mecanismos não existentes em JSD.

Ao introduzirmos o conceito de herança estamos promovendo uma mudança substancial no MOO, estabelecendo relações de hierarquia entre classes que não estão representadas em JSD. Como consequência, podemos estar acrescentando classes novas ao modelo, que, como vamos ver adiante, representam melhor o mundo real.

[GOSSAIN - 90] faz uma incursão neste assunto propondo uma maneira de se estabelecer estas relações através da interseção dos conjuntos de componentes entre as diversas classes. Esta é, sem dúvida a maneira mais ampla de se tratar o assunto. Para identificarmos as famílias de classes verificamos quais têm seus conjuntos de métodos, mensagens e dados da representação contidos em outras e podemos estabelecer a relação de subclasse. Alguns ajustes podem ser feitos quanto aos métodos pois o fato do conjunto destes não estar contido no de outra classe não invalida a priori a relação de herança.

Podemos supor duas classes A e B compostas por seus REP, MET, e MSG onde REP é a representação e MET e MSG são os conjuntos de métodos e mensagens. Se REP(a) e MSG(a) estão contidos em REP(b) e MSG(b) respectivamente, há uma grande chance de B ser subclasse de A. Mesmo quando MET(a) não está propriamente contido em MET(b) devemos verificar se para cada método de A não contido em MET(b) não existe um método em B que deve ser chamado nas mesmas condições. Existindo, consideramos este métodos como equivalentes para efeito de identificação de super/subclasses.

Através desta inspeção podemos identificar classes não representadas em JSD que formam famílias com algumas das entidades do modelo JSD, como pode ser visto na fig. 8.2. Não identifica entretanto entidades que em JSD representam mais de uma entidade do mundo real (fig 8.1). Para este particular sugerimos uma inspeção nas estruturas destas entidades.

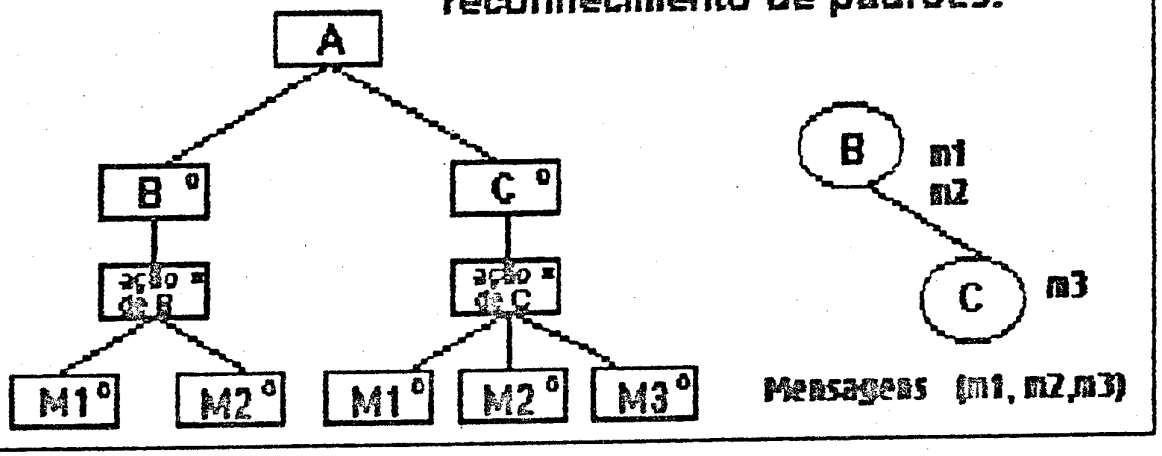
4.2.1 - Desdobrando classes em famílias.

Como mostramos na figura 8.1, JSD pode juntar várias entidades numa só, identificando que tratamento devem ter através de consulta ao seu vetor de estados, Para estas, uma breve inspeção de suas estruturas é suficiente para sua identificação. O padrão é sempre de uma seleção entre vários grupos logo no primeiro nível.

☐ *É o caso de um banco que tem clientes ditos "normais" que apenas depositam e sacam e clientes especiais que, além disto, podem fazer empréstimos, sacando a descoberto, bem como fazer investimentos no mercado financeiro.*

Para estes casos, criamos uma superclasse cujo REP é a interseção dos REP's necessários para cada alternativa. Seus conjuntos de métodos e mensagens são, analogamente, as interseções entre os conjuntos respectivos das alternativas. Podemos agora, excluir dos REP's o estado referente à identificação da alternativa.

Fig. 12 - Obtendo relações de herança através do reconhecimento de padrões.



☐ *Não raro, verificamos que a superclasse assim criada é idêntica a uma das alternativas, como pode ser visto na figura acima.*

4.2.2 - Agrupando classes em famílias.

O próximo passo é descobrir classes sem a correspondente entidade em JSD que são superclasses de classes existentes no modelo. Para este particular, inspecionamos a interseção dos REP's e dos conjuntos de métodos e mensagens usados pelas diversas alternativas. Havendo uma quantidade razoável de elementos comuns (e aqui o processo é empírico, contando com o bom senso do projetista), podemos criar uma família de classes ligadas por herança.

☐ *Cabe aqui dizer que o fato desta transformação se fazer contando com um mínimo de empirismo não invalida o processo. Lembramos que o próprio mapeamento do mundo real proposto pelas metodologias citadas, apesar de todas as regras, leva uma boa dose de empirismo. Isto se deve à ambiguidade existente na forma de modelar sistemas do mundo real, a começar pelo que se chama de "ponto de vista do observador", resultando em várias representações possíveis para um mesmo modelo.*

4.3 - Usando Composição.

JSD para representar entidades que agrupam outras entidades usa a comunicação entre processos. Uma forma que temos em MOO para simplificar esta comunicação, é usar a composição. Se adicionarmos ao REP de uma classe, um objeto de outra, estabelecemos a ligação sem perder o encapsulamento. Ao nosso ver, esta é uma maneira mais elegante de se representar alguns tipos de conexão entre processos.

No exemplo usado em [JACKSON - 83], onde são identificadas as "entidades marsupiais", podemos nitidamente usar composição. O exemplo cita uma entidade ***cliente*** que possui possui várias ***contas*** num banco, concluindo por separar cliente de suas contas, agora vistas como entidades. Em MOO podemos perfeitamente representar a classe cliente como composição de objetos da classe conta.

Concluindo, sempre que se verificar que todas as mensagens recebidas por um processo P, seja ele proveniente de entidade ou função, podem ser

canalizadas através de um único processo Q, podemos acrescentar ao REP de Q uma instância de P no MOO equivalente.

4.4 - Introduzindo o Polimorfismo.

Este mecanismo, como já foi dito, não é tratado especificamente em JSD. Jackson é enfático quando diz que se uma entidade muda de tipo no mundo real, esta deve ser representada por mais de uma no modelo ou, quando possível, por apenas uma que englobe todos os tipos. Ao desmembrarmos classes como mostrado em 4.2.1, estamos introduzindo o polimorfismo em nosso modelo.

☐ *Suponha uma entidade* ***pessoa*** *que possui dois estados:* ***aluno*** *e* ***professor****. Representamos esta entidade em MOO por uma classe* ***pessoa*** *com duas subclasses:* ***aluno*** *e* ***professor****. Ao mudarmos o estado de uma instância de aluno para professor estaremos em MOO mudando a classe do objeto de aluno para professor, o que é perfeitamente válido já que aluno e professor pertencem à família de classes cuja raiz é pessoa.*

5.0 - Conclusões.

Desenvolver uma metodologia para a área de desenvolvimento de sistemas é sempre uma tarefa trabalhosa, complexa e delicada.

No curso deste trabalho, por vezes nos sentimos tentados a esboçar o que poderia servir como base para a construção de uma. Entretanto o esforço para produzí-la consistente, abrangente e que não seja apenas uma "colcha de retalhos" de outras já existentes, só se justifica na medida em que estivermos realmente inovando.

Percebemos que muitas metodologias existentes satisfazem plenamente (ou quase) desde que usadas dentro de seus escopos de aplicação. A única ressalva mais séria que nos permitimos fazer a estas é ao hábito de alguns de seus autores de dizê-las mais abrangentes do que realmente são. O mundo real é muito menos restritivo do que as metodologias conseguem ser, daí concluirmos que, antes de iniciarmos o processo de modelagem, devemos escolher a forma ou a linguagem mais adequada para representá-lo.

Com este trabalho, não pretendemos ter esgotado as mudanças que podemos fazer num MOO, obtido a partir de um modelo JSD, para torná-lo mais fiel ao mundo real, nem os mapeamentos possíveis de JSD para MOO. Nosso objetivo foi mostrar que MOO's são perfeitamente adequados para modelar o mundo procedural, isto é, o conjunto das aplicações representadas adequadamente por modelos procedurais (MP's). Verificamos entretanto que MOO's são mais abrangentes pois mapeiam todos os aspectos que pudemos observar de MP's, o mesmo não acontecendo se invertemos o sentido do mapeamento.

Gostaríamos de acrescentar, a título de finalização, que apesar de ser perfeitamente natural passarmos de um MP para um MOO, se dispomos de um ambiente orientado a objetos, o mais simples é já se iniciar pensando "orientado a objetos". Isto, evidentemente, não invalida o que vimos em JSD que, das metodologias procedurais observadas é a que, na nossa opinião, mais facilmente pode ser adaptada para o mundo dos objetos.

Bibliografia.

☐ *1 - [AMERICA - 89] Pierre America*

Issues in the Design of a Parallel Object-Oriented Language.

Formal Aspects of Computing (1989) 1:366-411

BCS

☐ *2 - [BUHR - 88] P. A. Buhr & C. R. Zarkne*

Nesting in an Object Oriented Language is NOT for the Birds.

ECOOP 88 - pp 128-145

☐ *3 - [CARVALHO - 91] Sergio E. R. Carvalho, Antonio Gil C. Ayres, Otavio P. Coelho, Nelson Gorini, Michael Gouker, Werther J. Verdoet & SPA Sistemas, Planejamento e Análise SA*

The Programming Language TOOL

Em confecção. (Editado pela SPA Sistemas, Planejamento e Análise SA)

☐ *4 - [CHAMPEAUX - 90] Dennis de Champeaux, Larry Constantine, Ivar Jacobsen, Stephen Mellor, Paul Ward & Edward Yourdon*

Panel: Structured Analysis and Object-Oriented Analysis

ECOOP/OOPSLA '90 Proceedings (Oct 90) pp 135-139

☐ *5 - [CHEN - 76] Peter Pin-Shan Chen*

The Entity-Relationship Model - Toward a Unified View of Data

ACM Transactions on Database Systems, Vol I, March 1976, pp:9-36

☐ *6 - [COELHO - 91] Otavio Pecego Coelho*

COOL: The Communicating Objects Oriented Language

Tese de doutorado PUC-RJ, ainda em confecção

☐ *7 - [DE MARCO - 79] Tom DeMarco*

Structured Analysis and Systems Specification

Prentice Hall

☐ *8 - [GANE - 77] Chris Gane & Trish Sarson*

Structured Systems Analysis: Tools & Techniques

IST Databooks

☐ *9 - [GOSSAIN - 90] Sanjiv Gossain & Bruce Anderson*

An Iterative Model for Reusable Object-Oriented Software

ECOOP/OOPSLA '90 Proceedings (Oct 90) pp 12-27

☐ *10 - [JACKSON - 75] Michael A. Jackson*

Principles of Program Design

Academic Press

☐ *11 - [JACKSON - 83] Michael A. Jackson*

System Development

Prentice-Hall

☐ *12 - [MEYER - 89] Bertrand Meyer*

From Structured Programming to Structured Design: The Road to Eiffel

Structured Programming (1989) 1:19-39

☐ *13 - [PETZOLD - 88] Charles Petzold*

Programming Windows

Microsoft Press

☐ *14 - [RUMBAUGH - 91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy & William Lorensen*

Object-Oriented Modelling and Design.

Prentice Hall

☐ *15 - [SOARES - 88] Jefferson F. Soares, Roberto Ierusalimschy & Theonilla E. C. Pessoa*

Um Ambiente para a Transformação Semi-Automática de Diagramas de Fluxos de Dados em Diagramas de Objetos.

Monografias em Ciência da Computação 6/88 PUC-RJ

☐ *16 - [VERVLOET - 91] Werther J. Vervloet*

As Tabelas de Símbolos e as Linguagens Orientadas a Objetos

Monografias em Ciência da Computação 6/91 PUC-RJ

☐ *17 - [YOURDON - 78] Edward Yourdon & Larry Constantine*

Structured Design

Yourdon Press

☐ *18 - [ZAVE - 89] Pamela Zave*

A Compositional Approach to Multi-Paradigm Programming

IEEE Software Sep 89 pp 15-25