


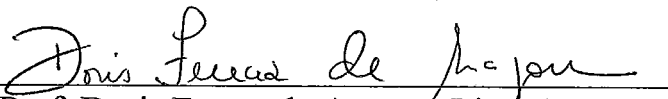
# Modelagem do Processo de Desenvolvimento de Software

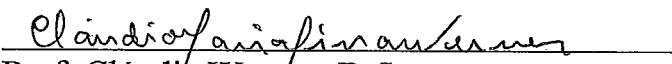
Jobson Luiz Massollar da Silva

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

  
Prof. Ana Regina Cavalcanti da Rocha, D.Sc.  
(Presidente)

  
Prof. Doris Ferraz de Aragon, Livre Docente

  
Prof. Cláudia Werner, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 1993

Silva, Jobson Luiz Massollar da

Modelagem do Processo de Desenvolvimento de Software (Rio de Janeiro) 1993.

viii, 86p., 29,7 cm (COPPE/UFRJ, M. Sc. Engenharia de Sistemas e Computação, 1993).

Tese - Universidade Federal do Rio de Janeiro, COPPE

1 - modelagem de processos 2 - ciclo de vida 3 - metamodelo

I. COPPE/UFRJ II. Título (série)

A minha mãe e a Jane

## **Agradecimentos**

Ao Senhor acima de tudo.

A Prof. Ana Regina por confiar na minha capacidade e pela orientação e apoio durante todo esse período na COPPE.

A Prof. Doris Aragon pelo incentivo ao meu ingresso na pós-graduação e por todos esses anos no ILTC.

A Prof. Cláudia Werner pelas sugestões e pela revisão cuidadosa do texto da tese.

Aos colegas da COPPE pelo interesse e pelo companheirismo e, especialmente, a Vera, por ter me cedido a sala de estudos onde tive o sossego e o conforto necessários para escrever essa tese.

A Cláudia e Ana Paula da secretaria e ao pessoal do laboratório, sempre prestativos.

Aos amigos do ILTC Maurício, Cláudia, Emília, Magali e tantos outros que, ao longo desses anos, sempre proporcionaram um excelente ambiente de trabalho e uma convivência harmoniosa.

Aos meus pais que sempre fizeram de tudo para me oferecer o melhor.

A Jane pelo carinho e amor, sempre.

Resumo da tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M. Sc.)

## **MODELAGEM DO PROCESSO DE DESENVOLVIMENTO DE SOFTWARE**

Jobson Luiz Massollar da Silva

Dezembro de 1992

Orientadora: Ana Regina Cavalcanti da Rocha

Programa: Engenharia de Sistemas e Computação

Este trabalho apresenta uma linguagem gráfica para construção de modelos do processo de desenvolvimento de software. Como ponto de partida para o projeto da linguagem foi feito um estudo de vários metamodelos, procurando determinar suas principais características e como cada um deles abordava a modelagem. A partir daí, definiu-se uma linguagem de modelagem que segue a perspectiva comportamental e que dá ênfase a quatro aspectos do processo de software: o tratamento de iterações, a modelagem de pontos de tomada de decisão, a representação da dinâmica do processo e a representação do paralelismo. É apresentada, ainda, uma ferramenta gráfica para edição de modelos escritos na linguagem proposta. A utilização dessa linguagem se dá na fase de "Definição do ciclo de vida" da ferramenta "XAMÃ: Planejador de ADS", que faz parte do projeto TABA.

Abstract of thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

## **SOFTWARE DEVELOPMENT PROCESS MODELING**

Jobson Luiz Massollar da Silva

December, 1993

Thesis Supervisors: Ana Regina Cavalcanti da Rocha  
Department: Systems and Computer Engineering

This work presents a graphic language for software process modeling. The language design began with a study of several metamodels, their main characteristics and the approach used in each of them. From this point we defined a modeling language that implements the behavioral approach. This language emphasizes four aspects of process modeling: iteration modeling, decision making modeling, representation of process dynamics and paralelism. It also presents a graphic tool to edit the models written in the proposed language. This language is to be used in the "Life Cicle Definition" phase of Xamã, a tool for SDE planning that is part of TABA project.

# ÍNDICE

<b>I</b>	<b>Introdução</b>	<b>1</b>
I.1	Objetivo do trabalho .....	1
I.2	Conteúdo do trabalho .....	3
<b>II</b>	<b>Modelagem de Processos</b>	<b>5</b>
II.1	Definição .....	5
II.2	Objetivos da modelagem .....	6
II.3	Aspectos da modelagem .....	8
II.3.1	Iteração .....	8
II.3.2	Dependência entre objetos .....	9
II.3.3	Representação do conhecimento e tomada de decisão .....	9
II.3.4	Granularidade .....	10
II.3.5	Formalização do processo .....	11
II.4	Conclusão .....	13
<b>III</b>	<b>Metamodelos do Processo</b>	<b>14</b>
III.1	Definição .....	14
III.2	Perspectivas de representação do processo .....	14
III.3	Aspectos da criação de modelos de software .....	16
III.4	Exemplos de metamodelos do processo de software .....	19
III.4.1	Integração de processos em ambientes CASE .....	19

III.4.2	Visual Process Language .....	23
III.4.3	STATEMATE .....	29
III.4.4	DesignNet .....	33
III.4.5	Desenvolvimentos de software como objetos computáveis .....	36
III.5	Conclusão .....	46
<b>IV</b>	<b>A Linguagem de Processos</b> .....	<b>47</b>
IV.1	Introdução .....	47
IV.2	Definição da linguagem de modelagem .....	49
IV.2.1	Elementos da linguagem .....	50
IV.2.2	Modelagem das iterações .....	55
IV.2.3	Operador de tomada de decisão .....	57
IV.2.4	Objetos do tipo GRUPO .....	58
IV.2.5	Objetos virtuais e instanciação .....	58
IV.3	Paradigma de execução .....	60
IV.4	Exemplos de utilização da linguagem .....	62
IV.4.1	Modelagem do ciclo de vida cascata .....	63
IV.4.2	Modelagem do ciclo de vida espiral .....	69
IV.5	A ferramenta EPROS .....	71
IV.6	Conclusão .....	76
<b>V</b>	<b>Conclusão</b> .....	<b>77</b>
	<b>Apêndice A</b> .....	<b>81</b>
	<b>Bibliografia</b> .....	<b>83</b>

# CAPÍTULO I

## INTRODUÇÃO

### I.1 Objetivo do Trabalho

O desenvolvimento de sistemas de software, principalmente de larga escala, é uma atividade que requer um planejamento cuidadoso de como atingir os objetivos desejados bem como um gerenciamento eficiente dos recursos utilizados. Soma-se a isso uma grande quantidade de conhecimento que é produzida e consumida durante o processo de desenvolvimento e tem-se um ambiente altamente dinâmico e complexo, onde a constante realimentação e alteração do próprio processo de desenvolvimento são a tônica.

Dentro desse contexto, a utilização de um ambiente automatizado dotado de ferramentas capazes de auxiliar de forma efetiva o desenvolvimento do software é de importância vital para o sucesso do empreendimento.

Na década de 80 começaram a surgir os primeiros Ambientes de Desenvolvimento de Software (ADS) capazes de dar suporte tanto às atividades de mais baixo nível do ciclo de vida (ligadas à programação propriamente dita) quanto às atividades de mais alto nível (ligadas à especificação, projeto, etc). Porém, a natureza e as características de cada sistema de software determinam, de certa forma, o perfil mais apropriado para o ADS a ser utilizado no desenvolvimento daquele sistema. A

necessidade de se configurar o ADS de acordo com o software que se pretende desenvolver leva a três características essenciais dos ambientes de desenvolvimento do futuro:

- Integrado - o ADS deve integrar os diversos elementos do processo de desenvolvimento em uma visão coerente.
- Extensível - o ADS deve possibilitar a incorporação de novos recursos (ferramentas, metodologias, etc) de forma fácil.
- Adaptável - o ADS deve se adaptar ao desenvolvimento de cada sistema de software.

Todas essas características sugerem um ADS altamente dinâmico e configurável. No entanto, tais capacidades ainda não são suficientes diante da demanda por sistemas cada vez mais complexos e robustos. Pesquisas mais recentes apontam na direção de ADS capazes de guiar e coordenar o processo de desenvolvimento de um sistema de software [SCAC92, HUMP89].

Para que um objetivo dessa natureza seja atingido, o ADS deve fornecer mecanismos para a geração de um modelo do processo de desenvolvimento de software, no qual possam ser especificadas todas as atividades ligadas ao desenvolvimento de um software, tendo associado a cada uma delas componentes como: recursos necessários, métodos empregados, ferramentas utilizadas, produtos gerados, restrições de ativação, consequências de execução, etc. Tal modelo seria usado, basicamente, para:

- Executar tarefas passíveis de total automatização.
- Coordenar e auxiliar a execução de tarefas mais complexas.
- Assegurar informações gerenciais atualizadas.
- Integrar as ferramentas e os objetos do processo de desenvolvimento.

O elemento central do mecanismo para definição de modelos é o metamodelo, ou seja, um modelo genérico no qual podem ser construídos modelos específicos de desenvolvimento.

Este novo enfoque mostra uma evolução nos ADS [SCAC92]: partindo inicialmente de um ambiente orientado à integração de ferramentas (chamada e controle implícito de ferramentas), passando pelos ambientes orientados à integração de objetos (geração, acesso e controle dos objetos do processo de desenvolvimento) e chegando aos ambientes orientados à integração de processos (utilização de um modelo explícito do processo de desenvolvimento de software).

Este trabalho tem a finalidade de definir um metamodelo que será utilizado na fase de "Definição do ciclo de vida" da ferramenta "XAMÃ: Planejador de ADS" [AGUI92] que faz parte do projeto TABA [ROCH90, ROCH91]. Este projeto visa a construção de uma estação de trabalho que auxilie no planejamento e construção do ADS mais adequado ao desenvolvimento de um produto específico, bem como na implementação das ferramentas necessárias a este ambiente. Também é apresentada, neste trabalho, uma ferramenta que permite a construção de modelos de processo de desenvolvimento utilizando a linguagem de modelagem proposta.

## **I.2 Conteúdo do Trabalho**

O capítulo II apresenta a definição e os objetivos da modelagem de processo de software, além de abordar alguns aspectos da modelagem de processos.

No capítulo III é feito um estudo dos metamodelos, enfocando questões como perspectivas do processo, grau de formalismo e criação de modelos. Também são apresentados alguns exemplos de metamodelos.

No capítulo IV é apresentada a linguagem de modelagem proposta, abordando sua perspectiva e suas características de representação. Neste mesmo capítulo são dados exemplos comentados da utilização da linguagem, procurando cobrir os aspectos mais importantes do processo e as soluções de modelagem oferecidas pela linguagem. A ferramenta para edição de processos de software segundo a linguagem proposta também é descrita nesse capítulo.

Finalmente, o capítulo V apresenta as conclusões e destaca as direções que dão continuidade à pesquisa apresentada neste trabalho.

# CAPÍTULO II

## MODELAGEM DE PROCESSOS

### II.1 Definição

Um processo é uma coleção de atividades relacionadas que têm lugar durante o desenvolvimento de um produto. Quando se abstrai o processo real de desenvolvimento a ponto de vê-lo como um padrão temos um modelo desse processo de desenvolvimento. O propósito central de um modelo é reduzir a complexidade de um fenômeno e a interação com o mesmo pela eliminação de detalhes que não influenciam seu comportamento [CURT92].

Caracteriza-se, então, um modelo de processo de software como uma representação suficientemente geral para modelar vários processos específicos de desenvolvimento de software, e suficientemente específica para permitir o raciocínio sobre esses modelos. O modelo do processo de software é um elemento que permite organizar e descrever como o ciclo de vida, os métodos, as ferramentas, os produtos e os desenvolvedores se relacionam [DOWN87, SCAC92].

A seguir são apresentados conceitos de alguns termos utilizados na modelagem de processos e que também serão utilizados neste trabalho [HUMP92, DOWN91].

- **Atividade** - uma ação atômica sem nenhuma estrutura externa visível. Também denominada elemento ou passo do processo.
- **Tarefa** - uma ação decomposta em sub-tarefas ou atividades. Muitas vezes o termo *processo* é utilizado como sinônimo de tarefa.
- **Agente** - um ator (ser humano ou máquina) que executa uma atividade.
- **Papel** - um conjunto coerente de atividades atribuídas a um ator.
- **Artefato** -um produto criado ou modificado pela execução de uma atividade. Também denominado objeto de software.

## II.2 Objetivos da Modelagem

A utilização cada vez mais crescente da computação em diversas áreas da atuação humana vem estimulando o crescimento da demanda de sistemas de software cada vez mais robustos e confiáveis, isto sem falar na complexidade das aplicações que estão surgindo.

Dentro desse panorama, os ciclos de vida tradicionais têm se mostrado incapazes de atender às necessidades das equipes de desenvolvimento de software. Isto acontece porque as descrições geradas por esses ciclos de vida não correspondem ao processo real de desenvolvimento ou manutenção. Elas apresentam, geralmente, planos de ação em alto nível e não contêm as informações necessárias ao desenvolvimento eficiente de um projeto [CURT92]. Essa falta de correspondência se dá por fatores como:

- Descrições do processo em alto nível, que não têm uma relação mais estreita com as atividades reais.
- Descrições ambíguas, imprecisas ou incompreensíveis.

- Falha na atualização da documentação durante o desenvolvimento.

Por exemplo, um ciclo de vida tradicional pode mostrar as revisões que devem ser executadas ao final de cada fase, mas não expõe as incontáveis revisões dentro dessas fases a fim de gerar um produto de alta qualidade. Além disso, muitos aspectos da modelagem que serão vistos abaixo, são ignorados nesses modelos.

As pesquisas na área de modelagem de processos de software estão se consolidando em torno de uma série de objetivos. Curtis [CURT92] agrupa esses objetivos em 5 categorias:

**a) Facilitar a compreensão e o entendimento humano**

- Representar o processo de forma compreensível ao grupo de trabalho.
- Formalizar o processo para facilitar o trabalho em grupo.
- Prover informações suficientes ao trabalho de um indivíduo ou do grupo.
- Permitir a comunicação efetiva entre os membros da equipe.

**b) Dar suporte a evolução do processo**

- Identificar componentes necessários ao desenvolvimento de um software.
- Reutilizar processos bem definidos.
- Comparar alternativas de desenvolvimento para o processo.
- Estimar o impacto de modificações potenciais.
- Facilitar a incorporação de novas tecnologias ao processo
- Dar suporte ao gerenciamento das modificações.

**c) Dar suporte ao gerenciamento do processo**

- Desenvolver um modelo do processo de software específico para o projeto.
- Dar suporte ao desenvolvimento de planos para o projeto.
- Monitorar, gerenciar e coordenar o processo de desenvolvimento.

- Fornecer uma base para medições do processo.

#### **d) Dar suporte à execução automática**

- Automatizar partes do processo.
- Dar suporte ao trabalho cooperativo.
- Colecionar dados que reflitam a experiência com o processo.
- Aplicar regras que garantam a integridade do processo.

#### **e) Dar auxílio à execução do processo**

- Definir um ambiente de trabalho.
- Dar sugestões e material de referência a fim de facilitar o trabalho humano.
- Armazenar representações reutilizáveis do processo em um repositório de dados.

## **II.3 Aspectos da Modelagem**

A modelagem do processo de software vem ao encontro da necessidade de se especificar de forma clara e detalhada as particularidades do processo de desenvolvimento de um software específico. Serão vistos agora alguns aspectos da modelagem de processos.

### **II.3.1 Iteração**

Em geral, o processo de software está longe de ser uma simples sequência bem comportada de atividades executadas de uma forma linear: por exemplo, partindo dos requisitos, passando pelo projeto e codificação e chegando ao produto final. Existem, normalmente desvios desse percurso que representam a reativação de fases anteriores do processo devido à necessidade de mudanças, compulsórias ou desejadas. Esses desvios são denominados iterações, repetições,

backtracking, entre outros. Neste trabalho será utilizado o termo iteração, que foi adotado em [DOWN87].

A iteração cumpre papel vital no processo de desenvolvimento do software no momento em que ela está intimamente relacionada às modificações durante esse processo. É importante notar que essas iterações não ocorrem somente devido a erros na codificação ou projeto ou qualquer outra fase do desenvolvimento; elas representam, muitas vezes, a própria natureza evolutiva de um software.

A modelagem das iterações passa obrigatoriamente pela questão da dependência entre as atividades e objetos que compõem o processo de software, assim como pelas estruturas de controle que definem como os efeitos provenientes de uma iteração serão aplicados.

### **II.3.2 Dependência entre Objetos**

A ocorrência de uma iteração geralmente afeta um subconjunto de objetos e atividades que representam o estado atual do processo, tornando necessárias mudanças nesses elementos. Essas mudanças ocorrem, inicialmente, nos elementos diretamente afetados, e se propagam de acordo com as dependências entre os vários objetos. A modelagem de tais dependências deve permitir uma visão clara dos objetos direta ou indiretamente afetados por uma iteração.

### **II.3.3 Representação do Conhecimento e Tomada de Decisão**

O desenvolvimento de um produto de software é uma atividade que consome e produz uma grande quantidade de conhecimento, conhecimento este que raramente é registrado explicitamente [BARS87]. Por exemplo, o conhecimento sobre o domínio das aplicações se reflete nos documentos de especificação e

requisitos, porém o raciocínio que está por trás desses documentos são raramente armazenados. Da mesma forma, os documentos de projeto e implementação não explicitam as motivações ou imposições das decisões tomadas.

O gerenciamento eficiente das modificações que ocorrem no desenvolvimento de um software (modificações no software propriamente dito ou no seu processo de desenvolvimento) passa, obrigatoriamente, pelo acesso ao conhecimento embutido nas várias decisões tomadas durante o processo de software (suas motivações ou imposições). Da mesma forma, somente com o conhecimento sobre os vários elementos do processo (por exemplo, seu domínio, sua origem e seu papel dentro do processo) é possível propagar as modificações de maneira eficiente [PERR89]. Além disso, o conhecimento armazenado, contendo todo o histórico do desenvolvimento de um software, é primordial para o entendimento e aprimoramento do processo de software, podendo servir de base à reutilização de modelos.

### **II.3.4 Granularidade**

Um modelo de processo pode conter descrições em vários níveis de abstração, e existe uma relação direta entre esses níveis e a ambiguidade dessas descrições: quanto maior o nível de abstração na descrição de uma atividade, maior o potencial de ambiguidades que ela carrega.

Entretanto, o modelo não deve impor nenhum grau particular de abstração. Quando as atividades são representadas em um alto grau de abstração, as pessoas envolvidas na sua execução devem ter o conhecimento necessário para eliminar as imprecisões que porventura existam. Por outro lado, se essas pessoas não possuem um conhecimento suficiente sobre o sistema a ser desenvolvido ou sobre o próprio processo de desenvolvimento, torna-se necessária uma descrição mais detalhada.

Geralmente, as descrições em mais alto nível (também denominadas roteiros do processo) são utilizadas em atividades a serem executadas por seres humanos, e as de mais baixo nível (também denominadas programas de processo) em atividades a serem automatizadas. Essa flexibilidade na formalização do comportamento é extremamente importante no momento em que tanto permite a implementação de um apoio automático à execução do processo quanto permite a modelagem de regras de decisão complexas. Entretanto, é importante ressaltar que mesmo as atividades de mais alto nível devem ser descritas em um nível de abstração bem menor do que aquele utilizado pelos ciclos de vida tradicionais.

As descrições mais precisas do processo de desenvolvimento têm sido modeladas utilizando-se uma abordagem baseada em linguagem de programação, enquanto que as descrições mais abstratas são modeladas através de técnicas utilizadas em linguagem de projeto, principalmente as linguagens com forte apelo visual.

A capacidade de realizar o refinamento de atividades (dentro de uma abordagem top-down) em vários níveis de abstração é essencial, pois além de fornecer uma metodologia bastante natural para a construção do modelo também possibilita uma visualização bastante simplificada do processo, já que cada nível possui um domínio restrito de atuação, mantendo-se afastados detalhes que não são relevantes naquele ponto.

### **II.3.5 Formalização do Processo**

Os modelos que estão surgindo têm buscado descrever o processo de software com uma abordagem cada vez mais formal. Muitas linhas de pesquisa levantam a importância de representações executáveis do processo de software e esse conceito tem sido empregado em muitos modelos. Um modelo de processo é

executável quando algum mecanismo o utiliza no monitoramento do desenvolvimento de um software (por exemplo, um interpretador da linguagem de modelagem). Osterweil [OSTE87] afirma que o desenvolvimento de modelos de processo é análogo ao desenvolvimento de software e que, assim sendo, as linguagens de programação convencionais poderiam ser utilizadas na criação de tais modelos, bastando apenas adaptar as primitivas dessas linguagens para tal fim. Porém, como o próprio Osterweil conclui, é preciso escrever programas de processo a fim de que se possa encontrar as características necessárias as linguagens de processo, mas, por outro lado, é preciso ter uma linguagem com as características apropriadas para escrever programas de processo. Muitas linguagens tradicionais foram utilizadas como base para modelagem de processos e várias linguagens de modelagem surgiram desses experimentos nos últimos anos.

Entretanto, a abordagem proposta por Osterweil ganhou veementes críticas [LEHM87, CURT87, LEHM89]. A principal crítica era como tais linguagens representariam os aspectos heurísticos e criativos existentes no processo de desenvolvimento de um sistema de software? Sem essa representação os programas de processo seriam modelos como quaisquer outros e essa abordagem seria essencialmente equivalente, dentro do contexto da modelagem de processos, ao uso de programação declarativa em contraste com outros estilos, tais como funcional, imperativo, etc. Todos esses fatores tornariam a programação de processos uma abordagem de pouco espectro: apenas os processos bem formulados, cujas estratégias e algoritmos estivessem bem definidos poderiam tirar proveito dela.

Apesar das controvérsias e de haver casos onde a formalização da descrição do processo não se aplica, os modelos que utilizam essa abordagem oferecem a oportunidade de raciocínio sobre o processo de desenvolvimento e servem de base para o desenvolvimento de ADS automatizados [WILL88].

## II.4 Conclusão

Como foi visto, os objetivos da modelagem de processo de software são bastante abrangentes e têm uma ligação direta e estreita com muitas outras áreas de pesquisa na engenharia de software. Da mesma forma, o modelo do processo não é um elemento que existe isoladamente, mas está inserido de forma marcante no contexto do desenvolvimento de um software e pode ser visto como um dos elementos centrais dos ADS do futuro. A modelagem do processo de software, como é vista hoje pelos pesquisadores, tenta, basicamente, eliminar o caráter estático das descrições tradicionais de ciclo de vida e imprimir uma nova dinâmica aos modelos de processo, reformulando e redimensionando o seu papel no desenvolvimento do software.

# **CAPÍTULO III**

## **METAMODELOS DO PROCESSO**

### **III.1 Definição**

Um metamodelo pode ser visto como uma linguagem adequada à descrição de modelos de processo [WILL88], por isso, muitas vezes é também chamado de linguagem de modelagem de processos. A partir das primitivas dessa linguagem é que são construídos modelos específicos de software.

### **III.2 Perspectivas de Representação do Processo**

Um modelo de software pode ser visto como a união de várias perspectivas do processo de desenvolvimento, onde cada uma delas revela um perfil do processo. Pode-se ter inúmeras perspectivas (também denominadas visões ou abordagens) do processo de desenvolvimento:

- Como o produto final está sendo desenvolvido, ou seja, quais as atividades do processo e suas relações.
- Que artefatos estão sendo gerados durante esse processo e as suas relações.
- Quem é o responsável pela execução de determinada atividade.

- Quando e onde será executada uma atividade qualquer.
- Por que o desenvolvimento se dá de uma forma e não de outra, isto é, quais as razões que levaram a uma determinada decisão.

Note que os itens acima têm um caráter interrogatório, revelando uma série de questões que comumente são levantadas durante o desenvolvimento de um sistema. As linguagens de modelagem geralmente apresentam uma ou mais abordagens relacionadas às questões acima. Dentre essas destacam-se quatro:

### **Abordagem Funcional**

Na abordagem funcional são levadas em consideração as atividades que estão sendo executadas e os fluxos de informação relevantes a essas atividades.

### **Abordagem Comportamental**

Nessa abordagem são representadas as sequências em que as atividades são executadas, assim como o comportamento da execução diante de iterações, tomadas de decisão, critérios de entrada e saída, etc.

### **Abordagem Organizacional**

Aqui são representados quando e por quem as atividades serão executadas.

### **Abordagem Informacional**

Representa os elementos de informação (dados, objetos, produtos) produzidos e consumidos pelo processo de desenvolvimento, levando em consideração as estruturas desses elementos e suas relações.

Embora pareçam independentes, pode-se induzir que a união dessas quatro perspectivas nos fornece um modelo completo, consistente e integrado do

processo. Porém, essa afirmação ainda necessita de confirmação experimental [CURT92].

As várias perspectivas do processo de software são construídas a partir das primitivas do metamodelo. Porém, para que se possa analisar as propriedades de um modelo, é preciso que o mesmo tenha uma coerência nas perspectivas que apresenta, ou seja, as primitivas devem ter um caráter semântico e sintático bem definido para que durante a construção do modelo não se intercale características de diversas abordagens em uma única representação (o que certamente a tornará confusa). O ideal, então, é que o metamodelo possibilite a construção das várias perspectivas de uma forma independente, porém consistente e integrada.

### **III.3 Aspectos da Criação de Modelos de Software**

Com o surgimento dos metamodelos e de um grande número de pesquisas nessa área, algumas questões surgiram e merecem destaque:

- Como os modelos de processo devem ser desenvolvidos ?
- Quais são os componentes necessários à descrição de um processo e como eles se relacionam ?
- O que é necessário para a execução de um modelo ?

Essas questões ressaltam dois enfoques dentro do estudo da modelagem de processos:

- O desenvolvimento e o gerenciamento de modelos de processo de alta qualidade.
- Mecanismos necessários para descrever e implementar modelos.

Existe uma analogia entre o primeiro enfoque e a engenharia de software e entre o segundo enfoque e o projeto e implementação de linguagens de programação.

Dentro do segundo enfoque parece haver um consenso sobre a necessidade de uma linguagem de modelagem de processos que dê suporte a múltiplos paradigmas, já que não existe nenhum capaz de descrever, sozinho, as diversas características do processo de software. Porém, Osterweil vai ainda mais longe e destaca que nenhum paradigma existente serve como base sólida para codificação de processos de software [OSTE89]. Ele declara que a modelagem do enorme grau de dinamismo existente no processo de desenvolvimento ainda não é possível e que esta é uma questão central que ainda deve ser pesquisada. Balzer também aborda este ponto quando critica a afirmação de que programas de processo são similares a programas de sistemas [BALZ89]. Sem dúvida, parece claro que um processo real de software só pode ser efetivamente modelado se novos tipos de objetos puderem ser criados, se os tipos existentes puderem ser modificados, se novos sub-processos puderem ser criados e integrados e se os processos existentes puderem ser alterados, tudo isso dinamicamente.

Dentro do primeiro enfoque parece haver uma reedição dos problemas enfrentados no desenvolvimento de sistemas: a construção de um modelo pode ser tão complexa quanto a construção do próprio produto de software. Seguindo a mesma linha de raciocínio, o ADS deve prover mecanismos que facilitem a construção de modelos, assim como é feito para a construção do produto de software. O aparecimento desse nível de abstração caracteriza o surgimento de uma nova classe de ambientes, orientados em função de *como* os desenvolvedores gostariam de construir e manter os produtos de software. Porém, por mais apoio que o ADS dê ao desenvolvimento do modelo, este sempre sofrerá alterações, pois a dinâmica do processo de software e as iterações ocorridas nesse nível afetam diretamente o meta-nível de desenvolvimento do modelo.

Nos estudos apresentados em [CURT87] são destacados o aprendizado, a comunicação técnica e as negociações e iterações com o usuário como fatores de alta relevância para o sucesso do desenvolvimento de grandes sistemas. Deve-se notar que esses fatores geram iterações dentro do processo de software, e que as consequências dessas iterações não se refletem somente no produto final, mas também no próprio processo de desenvolvimento. O ambiente deve, então, permitir a reestruturação do modelo, quando isso for necessário, baseado nos conhecimentos adquiridos nessas iterações. Nesse ponto, pode-se visualizar um ciclo evolutivo do modelo, que se estende por todo o processo de desenvolvimento do software. Surgem, ainda, duas questões importantes dentro desse ciclo evolutivo. Primeiro, é necessário que se faça uma análise do impacto das iterações sobre os produtos a serem gerados durante o andamento do processo e sobre o próprio processo. Essa análise vai permitir a detecção dos pontos críticos do sistema (pontos nos quais as iterações provocariam alterações mais significativas). Segundo, somente com a manutenção do conhecimento sobre as várias decisões dentro do processo é possível propagar as modificações de maneira correta.

Dentro desse quadro, onde a evolução é a tônica, o ADS não pode agir como uma entidade estática, mas sim permitir operações de reconfiguração e reestruturação do modelo de software e o gerenciamento de instâncias, customizações e da própria evolução do modelo. A utilidade do modelo e do próprio ADS depende diretamente da sua capacidade de auto-conservação.

Madhavji [MADH91] apresenta um ADS que procura atender a essas características, adotando mecanismos para construção, manutenção e simulação de descrições do processo, formando um meta-processo altamente incremental. [MADH92] define a modelagem das modificações para o ambiente acima.

## III.4 Exemplos de Metamodelos do Processo de Software

A seguir serão apresentados alguns metamodelos encontrados na literatura. Os exemplos abaixo não pretendem cobrir todos os enfoques utilizados na definição de metamodelos, porém eles apresentam características que demonstram bem a diversidade das soluções propostas. É importante notar que não será feita nenhuma avaliação ou comparação desses metamodelos (mesmo porque, ainda não existe nenhuma base de comparação plenamente aceita que diga que um metamodelo é melhor ou pior que outro), mas apenas a exposição das principais características de cada um, buscando situá-los dentro das abordagens apresentadas na seção III.1.

### III.4.1 Integração de Processo em Ambientes CASE

No trabalho apresentado por Scacchi e Mi [MI90, SACC92] é proposta uma arquitetura de ADS orientados a processos, onde os dois elementos centrais do ambiente são o metamodelo e o *process driver*. O metamodelo é visto como um repositório de informações sobre o estado dos processos e das atividades durante o desenvolvimento de um sistema. Ele especifica uma hierarquia de atividades e os requisitos de recursos de cada uma. No topo da hierarquia encontram-se as tarefas, representadas por um retângulo, que são decompostas em sub-tarefas ou ações. No nível mais baixo situam-se as ações, representadas por círculos, que denotam a chamada a uma ferramenta ou uma simples transformação de artefatos.

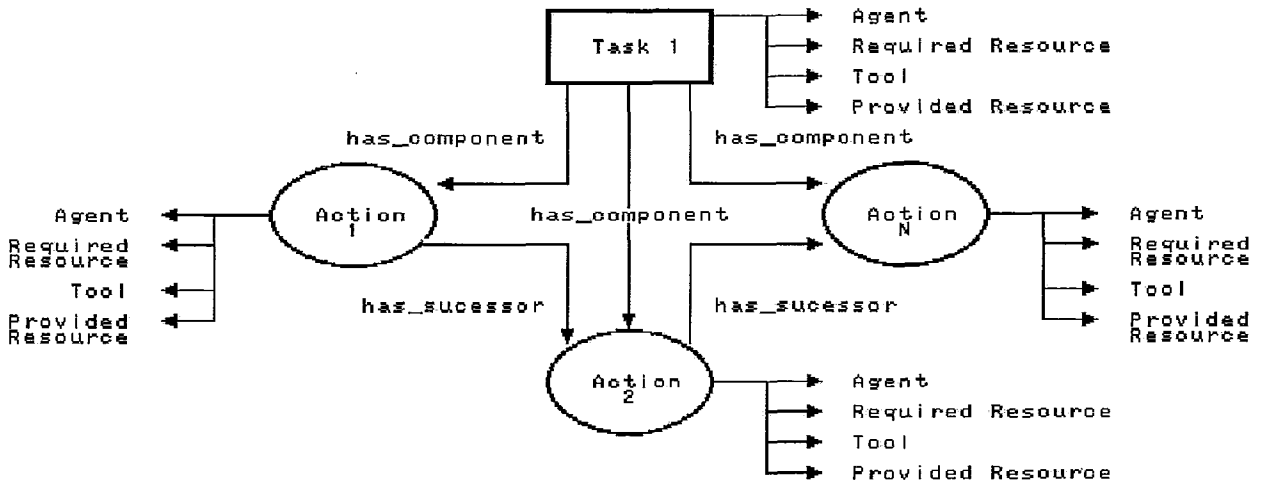


Figura 1: Estrutura de um modelo do processo

Cada nível da hierarquia especifica uma ordem parcial de execução das sub-tarefas. A ordem de execução define vários tipos de relações de precedência entre sub-tarefas: sequencial, paralela, iterativa e condicional. São também detalhadas quatro categorias de requisitos necessários à execução de cada sub-tarefa (figura 1):

- Os vários papéis que os desenvolvedores assumem durante a execução de sub-tarefas.
- Os artefatos necessários ("*required resources*") e criados ou alterados ("*provided resources*") durante a execução de uma sub-tarefa.
- As ferramentas utilizadas.
- As informações sobre escalonamento de sub-tarefas e sua duração esperada.

O *process driver* interpreta e executa o metamodelo de acordo com a hierarquia de atividades. Ele inicializa o metamodelo, executa as ações totalmente automatizadas, gerencia a ordem de execução e as restrições que devem ser atendidas para a execução de cada sub-tarefa, recebe as entradas dos usuários, atualiza e propaga

o estado das sub-tarefas e dispara outras sub-tarefas.

<b>Valor</b>	<b>Definição para Ações</b>	<b>Definição para Tarefas</b>
None	Inicialização	Sub-tarefas têm estado None
Allocated	Ferramentas e recursos alocados	Sub-tarefas têm estado Allocated
Ready	Alocado e com predecessores Done	Algumas sub-tarefas prontas, mas nenhuma ativa
Active	Execução em andamento	Algumas sub-tarefas ativas
Stopped	Execução para voluntariamente	Algumas sub-tarefas paradas, mas nenhuma ativa, quebrada ou pronta
Broken	Alguns recursos não disponíveis ou impossível de continuar	Algumas sub-tarefas quebradas mas nenhuma ativa ou pronta
Done	Execução terminada com sucesso	Sub-tarefas têm estado Done
Not Chosen	Não selecionada para execução	Não disponível

Tabela 1: Valores de estado e suas definições

Para cada sub-tarefa ou ação da hierarquia existe uma variável cujo valor representa o estado atual do desenvolvimento daquele elemento. O valor dessa variável é atualizado pelo *process driver* baseado no modelo e nas interações com o usuário. Essa atualização representa o progresso da sub-tarefa dentro do grafo de transição de estado. A tabela 1 mostra os valores assumidos pela variável de estado e a sua definição tanto para ações quanto para tarefas. Enquanto o modelo prescreve a ordem de execução das sub-tarefas, o histórico dos valores de estado descrevem a ordem real de execução.

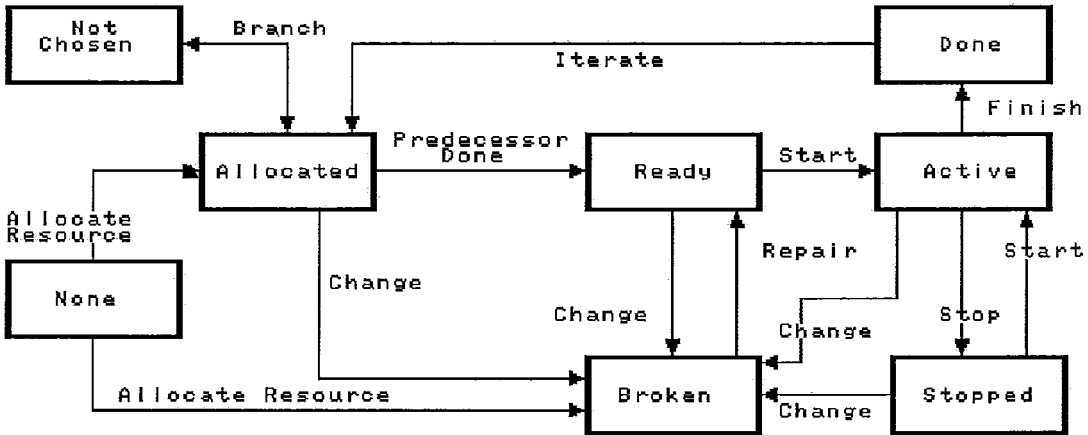


Figura 2: Grafo de Transição de Estados

A figura 2 apresenta o grafo de transição de estado. Nesse grafo os retângulos representam os valores de estado e as arestas representam os tipos de transição que levam de um estado a outro.

Uma vez que os gerentes tenham alocado os recursos necessários para o início do projeto, o *process driver* é acionado e este envia o sinal *start* às sub-tarefas e ações que estão em *ready* e o valor do estado desses elementos é alterado para *active*. Conforme as ações vão sendo executadas, seus estados podem retornar *done*, *stopped* ou *broken*. Este último geralmente leva a uma iteração no processo para recuperação da atividade. Quando as ações atingem o estado *done*, o *process driver* atualiza o modelo e coloca novas ações no estado *ready*. Esse mecanismo prossegue até que a tarefa de mais alto nível atinja o estado *done*, caracterizando o fim do desenvolvimento.

Durante a execução os gerentes monitoram o desenvolvimento do software observando as mudanças no estado das ações, além de poderem alterar o modelo ou os recursos alocados a qualquer momento.

Três interfaces de trabalho distintas estão relacionadas a este metamodelo. A primeira é a interface do desenvolvedor, que fornece o espaço de trabalho necessário à execução das ações previstas no modelo. O disparo de uma ação inicializa a área de trabalho, colocando a disposição dos usuários os recursos e as ferramentas previstas para aquela ação. A segunda interface é a do gerente do projeto, que fornece ferramentas para monitorar e controlar o modelo. Nessa interface é inicializado o modelo, assegurando que os recursos mínimos necessários para o início do mesmo estão alocados. A terceira interface é a do gerente do modelo, que fornece ferramentas para criação, prototipação, análise e simulação de modelos de processo de software.

### III.4.2 Visual Process Language (VPL)

A linguagem de modelagem VPL [SHEP92] é uma linguagem formal projetada para representar graficamente o processo de software e permitir a execução automática de atividades. As descrições formais são vitais para o suporte à execução automática, embora muitos aspectos do processo de software sejam difíceis ou impossíveis de serem formalizados. A VPL está baseada na abordagem funcional, embora implemente também alguns aspectos da abordagem organizacional.

Um modelo do processo escrito em VPL forma um grafo direcionado que satisfaz as seguintes restrições:

- O grafo é totalmente conexo.
- Cada nó tem que estar associado a um dos nove tipos embutidos na linguagem.
- Cada nó de entrada está conectado a uma, e somente uma, saída de outro nó.
- Não existe nenhum caminho entre a saída de um nó e a entrada deste mesmo nó, exceto através de um nó *branch*.

- Existe exatamente um nó de entrada e um nó de saída no grafo.

Associado a cada grafo VPL existem duas tabelas: uma de objetos e outra de papéis. Juntos, esses três elementos formam um programa VPL.

Um objeto em um programa VPL é qualquer artefato associado a uma atividade, e a execução do programa corresponde ao fluxo desses objetos pelo próprio programa.

A forma com que o ADS utiliza o programa VPL para direcionar e auxiliar as ações do usuário depende fortemente do papel desse usuário. Um papel pode ser visto como um rótulo que está associado a cada usuário do sistema para indicar as suas funções dentro do processo de software.

Existem nove tipos de nós em um programa VPL (figura 3):

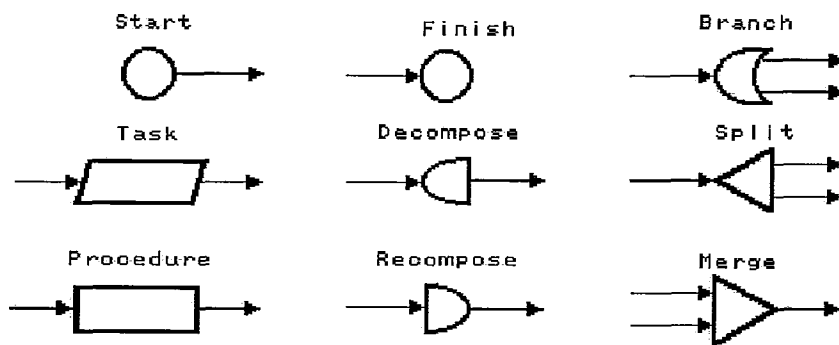


Figura 3: Tipos de nó da VPL

- *Start* - é o ponto de entrada dos objetos em um programa VPL ou procedimento. O início de um programa corresponde ao ponto em que os objetos são criados. O

início de um procedimento é o ponto em que os objetos entram nesse procedimento.

- *Finish* - é o ponto de saída de um programa ou procedimento. Quando os objetos chegam ao nó *finish* de um programa são desativados e armazenados, e quando chegam ao nó *finish* de um procedimento deixam o mesmo pelas arestas de saída.
- *Procedure* - serve apenas como uma abreviação conveniente para um grupo de nós que têm uma funcionalidade bem definida. O grupo de nós representados pelo nó *procedure* têm as mesmas características de um programa VPL.
- *Task* - apresenta uma ação a ser executada por uma ferramenta, ou por um usuário e uma ferramenta, sobre um objeto. O nó *task* difere do nó *procedure* porque ele é atômico e envolve um único usuário. Porém as tarefas podem requerer o envolvimento de um grupo de pessoas, com uma delas representando o usuário do nó. Um nó *task* pode também servir de interface entre uma atividade off-line (por exemplo, uma reunião) e o ambiente de desenvolvimento. Nesse caso o tempo de duração da reunião ou um registro das decisões tomadas poderiam ser objetos fornecidos ao sistema.
- *Decompose* - decompõe um objeto em uma família de objetos, cujos membros possuem apenas parte da informação do objeto original. Os objetos resultantes de um nó *decompose* seguem em paralelo.
- *Recompose* - acumula objetos nas arestas de entrada até que uma família esteja completa. Neste momento os objetos são sintetizados e surgem como um único objeto na aresta de saída.
- *Split* - cria um objeto duplicata do objeto de entrada. Este novo objeto segue por

Objetos são criados pelos nós *start*, *decompose* ou *split* e o programa pode conter um número qualquer de objetos distribuídos pelo grafo. No desenvolvimento do software, o resultado de uma tarefa é normalmente usado como ponto de entrada para a próxima tarefa. Essa característica é representada pelo mecanismo de avanço. Quando os objetos entram em um nó, as ações associadas com esse nó são executadas e o objeto, ou objetos, passam para o próximo nó. A ação de um nó pode mudar o estado de um objeto ou até mesmo o número de objetos. Nesse caso, essas mudanças são armazenadas em um banco de dados e as mensagens apropriadas são automaticamente enviadas aos supervisores.

O mecanismo de avanço provê apenas o controle linear dos objetos através do grafo. O controle não-linear é conseguido através da desativação e da reativação. A desativação de um objeto é a remoção desse objeto da tabela de objetos e o seu armazenamento em um arquivo, onde ele fica congelado. A reativação é o mecanismo oposto da desativação e ocorre quando uma versão mais antiga de um objeto que foi desativado é extraída do arquivo e recolocada na tabela de objetos. É importante notar que esses dois mecanismos não são explicitamente representados em um programa VPL, pois, de acordo com a visão de seus criadores, seria impossível determinar todas as ocasiões em que elas seriam necessárias. A solução foi implementar esses mecanismos como parte do interpretador VPL. Dessa forma, a desativação e reativação têm que ser explicitamente chamadas pelo usuário.

Quando ocorre uma iteração, o movimento dos objetos segue o sentido contrário das arestas do grafo. O nó para o qual o objeto retorna é especificado pelo usuário e a iteração é conseguida pela desativação de um objeto e pela reativação de versões mais antigas desse mesmo objeto. O mecanismo de backtracking, juntamente com o armazenamento do histórico do objeto, fornece meios eficientes para modelagem de abordagens que utilizam prototipação.

todas as arestas de saída do nó, de forma que processos diferentes possam ser executados sobre as cópias.

- *Merge* - age como um ponto de sincronismo para os processos concorrentes originados de um nó *split*. Quando uma família completa de objetos atinge um nó *merge*, alguns deles podem ser sintetizados para formar um objeto composto ou um deles pode ser escolhido como o melhor ou mais apropriado para continuar o processo, sendo o restante desativado.
- *Branch* - direciona o objeto na sua entrada para uma das arestas de saída. A decisão de qual aresta deve ser usada é tomada a partir de expressões que utilizam variáveis que representam o estado do objeto e a partir de interações com o usuário.

Todos os objetos ativos são armazenados em um arquivo chamado tabela de objetos como um conjunto de valores que correspondem ao nome do objeto e as suas variáveis de estado. O tipo do objeto é determinado pelo nó em que reside, não sendo um atributo próprio do objeto. Isso ocorre porque a informação associada a um objeto muda de nó para nó.

Todas as pessoas dentro de um projeto estão associadas a papéis, que são parte do modelo do processo. O papel que um indivíduo assume em determinado momento determina a lista de ações que ele pode executar.

O paradigma de execução da VPL combina características de orientação a objeto, redes Petri e diagramas de fluxo. A execução de um programa VPL é alcançada através de quatro mecanismos que agem sobre os objetos: criação, avanço (fluxo dos objetos através do grafo), desativação e reativação.

Para complementar a apresentação desse metamodelo a figura 4 ilustra o modelo de nível mais alto do processo de manutenção de um sistema de software. Esse modelo vem sendo utilizado na avaliação da VPL e do ADS que a suporta. A figura 5 ilustra o refinamento do nó procedure *generate software change request*. O nó procedure *change software* da figura 4 possui uma sub-procedure chamada *implement* que é mostrada na figura 6.

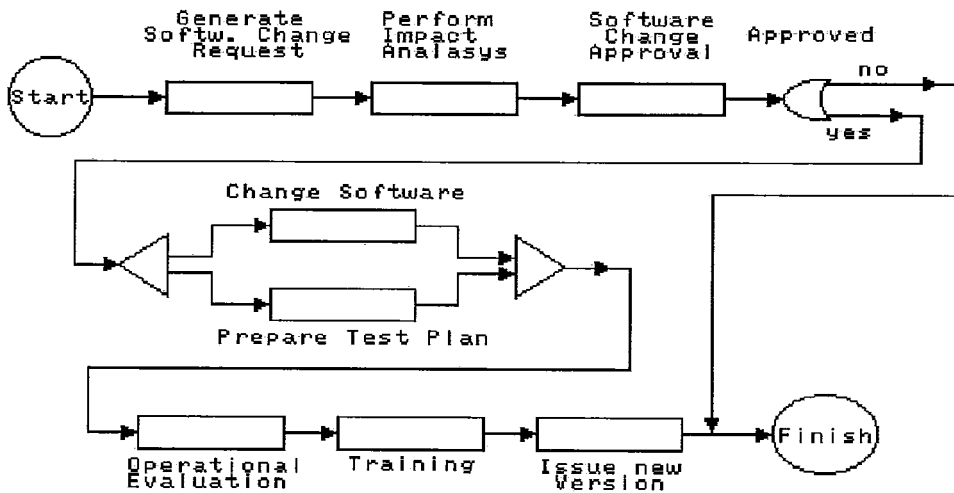


Figura 4: Procedimento principal do processo de manutenção de software

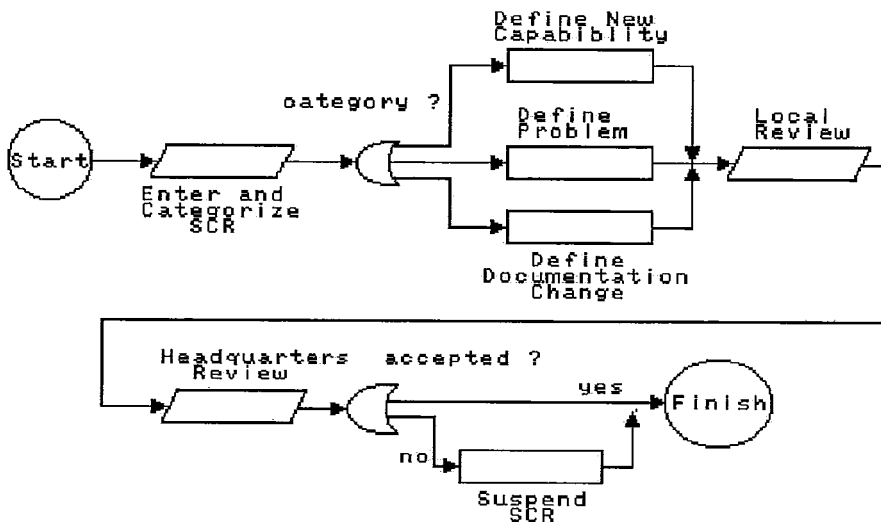


Figura 5: Refinamento do nó procedure Generate SCR

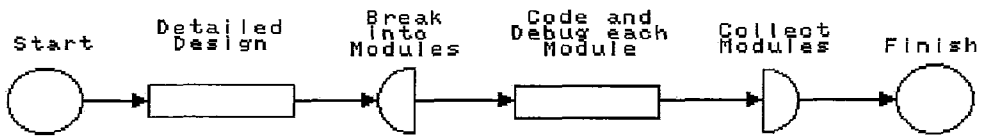


Figura 6: Refinamento do nó procedure Implement

### III.3.2 STATEMATE

A abordagem utilizada pelo STATEMATE [HUMP89] para modelagem de processos de software se concentra em três dos cinco objetivos apresentados na seção II.2: facilitar a comunicação e o entendimento humano, dar suporte à evolução do processo e dar suporte ao gerenciamento do processo.

Para a modelagem são utilizadas três linguagens visuais formais. Tais linguagens dão suporte às três perspectivas implementadas no metamodelo: funcional, comportamental e organizacional. Em cada uma dessas linguagens são utilizadas primitivas adequadas a sua finalidade:

- Diagramas de estado com eventos e disparos (usado na perspectiva comportamental).
- Diagramas de fluxo de dados aprimorados (usado na perspectiva funcional).
- Diagramas de estrutura modificados (usado na perspectiva organizacional).

Por ser um modelo bastante complexo, a exploração de todas as potencialidades do STATEMATE foge das finalidades desta seção. Por isso será exemplificada apenas a perspectiva comportamental do STATEMATE.

Neste metamodelo, a abordagem comportamental segue um modelo denominado modelo de entidades EPM (Entity Process Model). No EPM o processo de software é representado em termos de entidades reais e operações executadas sobre elas. Tais entidades são objetos que existem no mundo real e têm persistência durante todo o processo de software. Os documentos de requisito, o produto final, a documentação do software ou os documentos de projeto são exemplos de entidades manipuladas por este modelo. Ao invés de constituírem meros objetos gerados durante uma ou outra fase do desenvolvimento, as entidades perduram durante todo o processo.

As principais características da EPM podem ser resumidas da seguinte forma:

- Ela manipula entidades persistentes do mundo real.
- Cada entidade possui uma sequência bem definida de estados.
- A transição de um estado a outro resulta de causas bem definidas, embora estas causas possam depender do estado de outras entidades, assim como de condições específicas do processo.

O primeiro passo para a definição do modelo é a identificação das entidades e seus estados. Os requisitos básicos para que um objeto se torne uma entidade são:

- Existir no mundo real e não somente no modelo descrito.
- Ser unicamente identificável.
- Poder ser transformado pelo processo através de um conjunto bem definido de estados.

A especificação de uma entidade necessita, obrigatoriamente, da

definição de *mundo real*. Para propósitos de desenvolvimento de software, *mundo real* pode ser definido como o conjunto de usuários do sistema. Essa definição não inclui, por exemplo, o desenvolvimento propriamente dito e nem as atividades gerenciais. Porém a questão não é qual é a melhor ou mais correta definição, mas sim o que é mais importante em uma situação específica. Por exemplo, uma entidade em potencial é o protótipo produzido para esclarecer algumas questões do sistema. Assumindo que ele é produzido, testado e descartado, ele claramente não é uma entidade, mesmo que tenha contribuído de forma decisiva para o sucesso do projeto. O fato é que não ser uma entidade não significa não ser importante.

Identificadas as entidades e seus estados é necessário definir os disparos que causam a movimentação das entidades entre os seus estados.

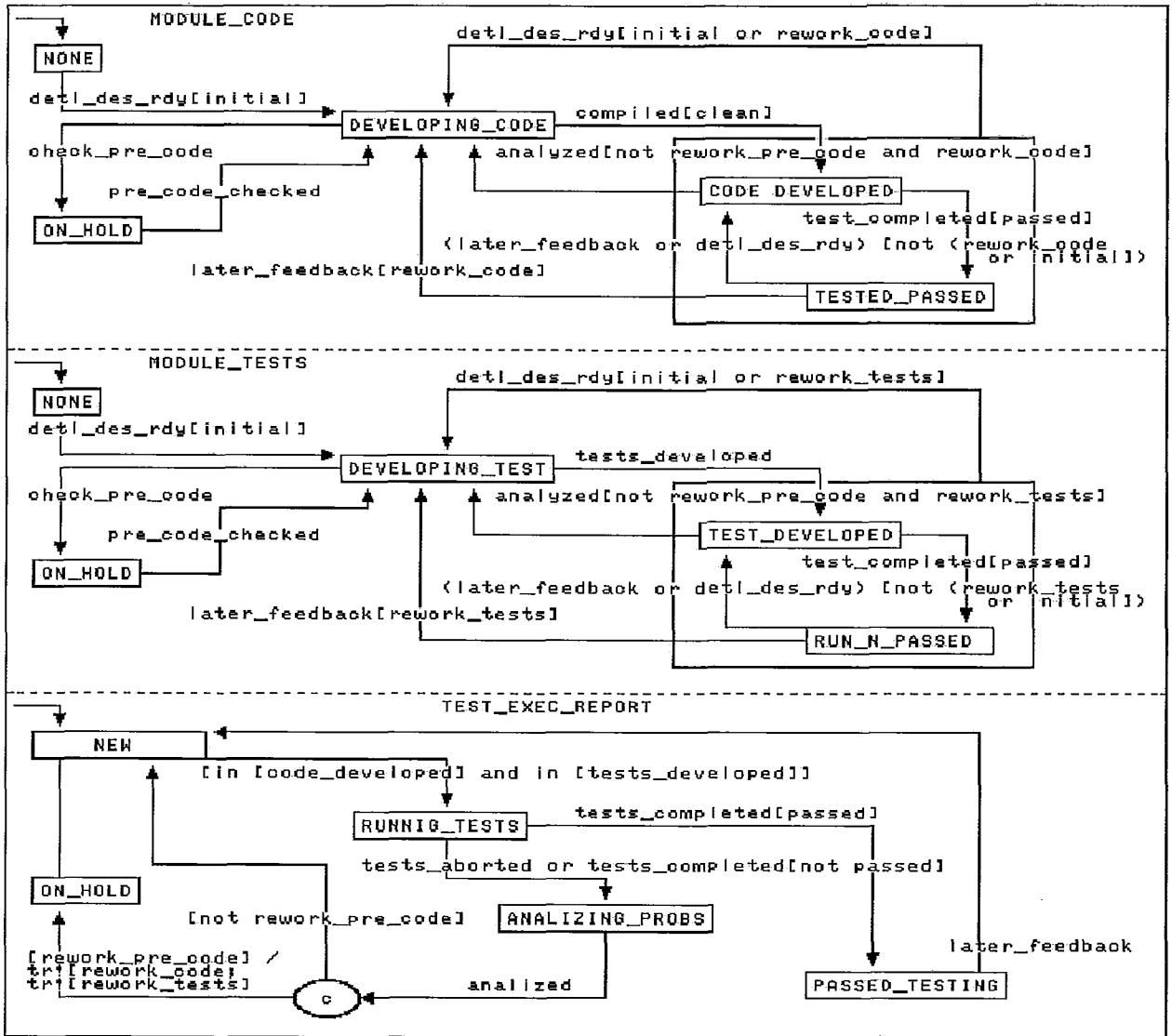


Figura 7: Diagrama STATEMATE

Na figura 7 é apresentado o diagrama de estado de uma pequena fase do processo, mais precisamente desde a fase de planejamento de um módulo (ao final do projeto detalhado) até a fase de testes desse módulo. Dessa forma são focalizadas três entidades:

- Código do módulo
- Unidades de teste
- Execução de testes e análise dos resultados

Os retângulos representam os estados, enquanto as linhas representam as transições. Estados podem ter componentes ortogonais, separados por uma linha pontilhada. Esses componentes representam o paralelismo do processo.

Nesse pequeno exemplo podemos destacar duas características:

a) Observe o estado ON\_HOLD em MODULE\_CODE e em MODULE\_TESTS. Este estado serve como ponto de parada enquanto alguns ajustes estão sendo feitos no processo (por exemplo, uma modificação no projeto). Logo, após a parada, pode-se retornar ao ponto anterior.

b) Observe também o conector condicional (círculo contendo a letra c). Este conector representa uma tomada de decisão sobre qual será o próximo estado. As expressões lógicas que guiam a decisão são montadas a partir de eventos. Além disso a expressão pode conter a ação a ser executada se aquele caminho for escolhido.

### III.3.4 DesignNet

DesignNet [HORO89] é um modelo formal que segue a terminologia usada por grafos AND/OR e redes Petri.

Uma DesignNet consiste de um conjunto de lugares, um conjunto de operadores estruturais e um conjunto de transições. Lugares são tipados e quatro tipos são pré-definidos: atividade, recurso, produto e relatório. Os operadores estruturais AND e OR conectam lugares de mesmo tipo em dois níveis adjacentes, onde os lugares de nível inferior são a decomposição do lugar de nível superior. Os pré-requisitos (produtos e recursos necessários) de uma atividade e os produtos gerados pela mesma são ligados à atividade por transições. Essa ligação define a dependência entre fases do ciclo de vida. O disparo das transições cria novas instâncias dos tokens

da rede e simula a execução do processo de software.

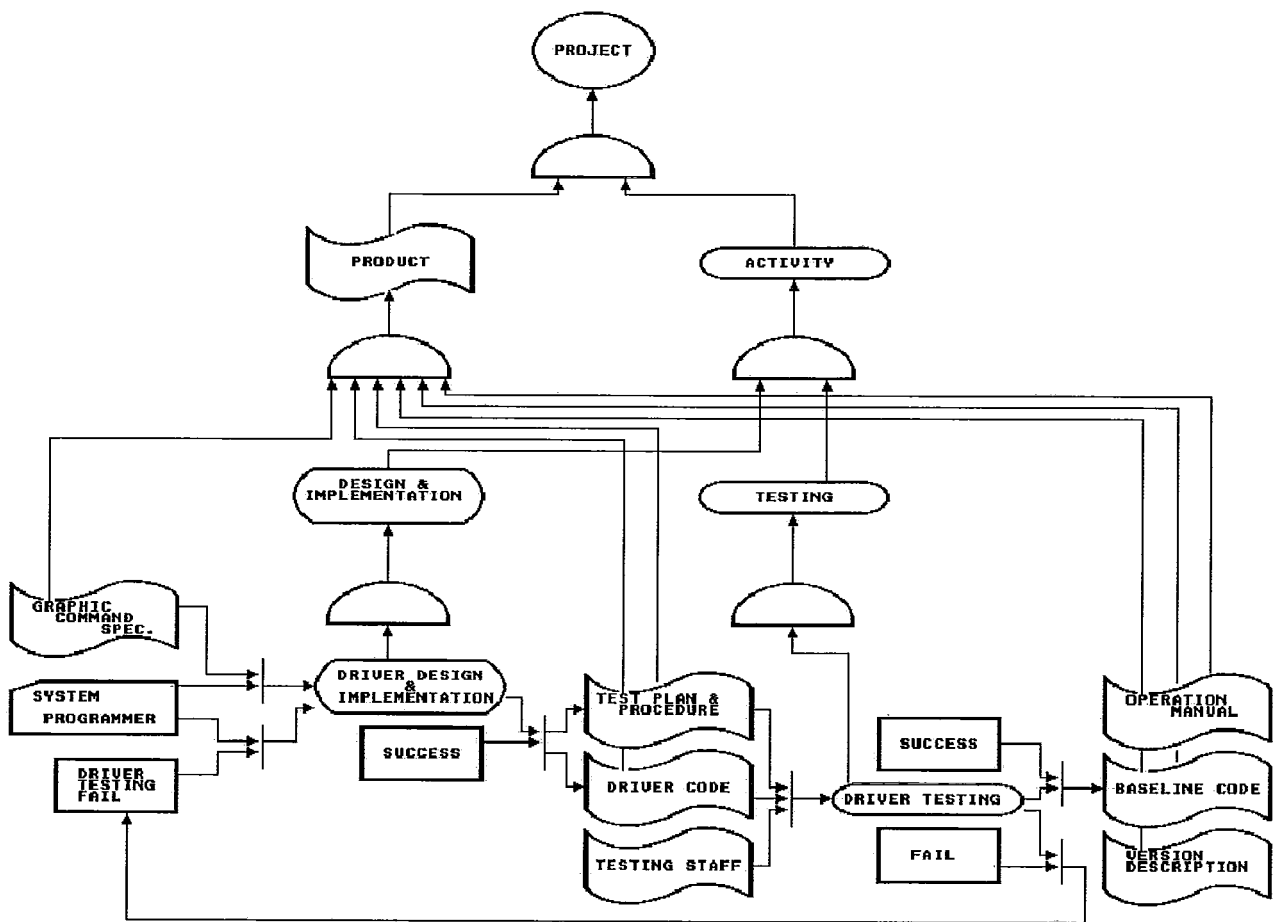


Figura 8: Diagrama DesignNet

Formalmente, uma DesignNet é composta de três componentes básicos (figura 8):

- Um conjunto de lugares  $P$ , onde  $P$  é a reunião de quatro tipos possíveis:  $P_a$  (atividades),  $P_r$  (recursos),  $P_p$  (produtos) e  $P_s$  (relatórios).
- Um conjunto de operadores estruturais  $S$ , onde  $S$  é a união de dois operadores  $S_a$  (AND) e  $S_o$  (OR).
- Um conjunto de transições  $T$ , onde  $T$  é a união de  $T_s$  (transição que inicia uma atividade) e  $T_f$  (transição que finaliza uma atividade), que são procedimentos

executáveis. Na DesignNet uma transição é representada por uma barra.

Os tokens que residem em um lugar possuem mais do que um simples valor booleano. Para cada tipo de lugar, um conjunto de propriedades é definido e qualquer token nesse lugar se torna um objeto composto de um tipo específico.

Há um tipo especial de nó denotado como nó raiz que possui nível zero. O operador AND define os subcomponentes de uma entidade agregada. O operador OR especifica quais alternativas estão disponíveis para o projeto. Lugares com níveis adjacentes são conectados pelas funções Ia, Io, Oa, Oo (Input AND, Input OR, Output AND, Output OR).

Os operadores estruturais permitem que informações de um lugar de nível mais alto sejam obtidas a partir de seus componentes. Eles também permitem que tokens recentemente criados em níveis mais baixos sejam propagados para os níveis superiores.

Os relacionamentos entre lugares de tipos diferentes são definidos pelas conexões das transições. Lugares de mesmo nível são conectados pelas funções Is, If, Os, Of (Input Start, Input Finish, Output Start, Output Finish).

Uma propriedade interessante é a possibilidade da projeção da rede em subconjuntos do seu domínio, ou seja, a visualização de diferentes perspectivas do processo. Removendo as transições e suas conexões Is, If, Os e Of chegamos a quatro partições (atividade, recursos, produtos e relatórios) da descrição original e perdemos a dependência entre tipos diferentes de informação. Removendo S e suas conexões Ia, Io, Oa e Oo o último nível da rede se torna uma rede de atividades (análoga ao diagrama PERT), enquanto a estrutura que mostra a decomposição das atividades é perdida.

A execução de uma DesignNet se dá por disparo de transições. Esse disparo é um processo não-volátil. Um token pode assumir um dos três possíveis estados: ativo, consumido e descartado. O disparo de uma transição pode ser executado se esta estiver habilitada. Isso acontece quando cada um dos lugares de entrada tem pelo menos um token ativo. O disparo da transição não remove os tokens ativos de seus lugares de entrada. Ao invés disso, ele muda o estado dos tokens nos lugares de entrada para consumido, evitando que futuros disparos para os mesmos tokens sejam executados. Os tokens criados nos lugares de saída são colocados em estado ativo. Enquanto são criados os tokens é também checado se existe algum token ativo nos lugares de saída. Se existir então seu estado se torna descartado. Isso garante que existirá, no máximo, um token no estado ativo em qualquer lugar a qualquer momento.

Uma marca  $M$  de uma DesignNet é uma atribuição de tokens aos lugares desta rede. O vetor  $M = (M_1, M_2, \dots, M_n)$  representa o número de tokens em cada lugar  $P_i$ , ou seja, o número de tokens no lugar  $P_i$  é  $M_i$ . Já que o disparo das transições não é volátil, o número de tokens em um lugar  $P_i$  nunca é decrementado. A sequência de marcas  $M^0, M^1, \dots, M^j$  (sendo  $0, 1, 2, \dots, j$  um índice de tempo) junto com as informações armazenadas com os tokens, dá o histórico da execução da DesignNet.

### **III.3.5 Desenvolvimentos de Software como Objetos Computáveis**

Soares propõe uma linguagem de modelagem que utiliza formalismo textual, complementado por uma representação gráfica do processo [SOAR92]. Neste trabalho o processo de software é descrito a partir de instâncias de elementos, onde cada uma dessas instâncias pertence somente a uma dentre quatro classes possíveis:

- Processo
- Atividade
- Componentes de dados
- Recursos

A linguagem proposta define relacionamentos intuitivos envolvendo essas quatro classes:

- Um *processo* é o resultado de cooperação/interação de uma coleção de *atividades*.
- *Atividades* normalmente manipulam *componentes de dados*, mas também podem manipular *recursos*, *atividades* ou, até mesmo, *processos*.

Componentes de dados nada mais são do que objetos de software, tal como documentos, códigos, relatórios, etc. A linguagem define também um modelo de atributos. Cada classe possui um conjunto de atributos que são comuns a todas as instâncias daquela classe. Atributos caracterizam informações, que podem ser de dois tipos: informações sobre uma instância em particular ou informações que representam relacionamentos entre instâncias. Para cada uma das classes são definidos os seguintes atributos:

### 1) Classe Processo

**Name:** identifica univocamente a instância dentro do processo.

**Status:** armazena o estado corrente da instância. Pode assumir os seguintes valores definidos pela linguagem: **not-started**, **running**, **complete-ok**, **complete-fail** ou **aborted**.

**Exp:** expressa a dinâmica de uma instância.

Os valores possíveis para o atributo **Exp** são expressões definidas a partir de um subconjunto da linguagem que segue o paradigma funcional. Os operandos dessas expressões são elementos das classes atividade ou processo e os operadores são os seguintes:

**Combine(elemento, ...)**

Implementa uma estrutura de controle sequencial, que executa os argumentos na ordem inversa em que eles são fornecidos. Dessa forma, a expressão `combine(elemN, ..., elem1)` representa a execução sequencial de `elem1, ..., elemN`.

**Construct(elemento, ...)**

Implementa uma execução independente de seus argumentos, de forma que não há nenhuma regra quanto a precedência de execução dos argumentos. É utilizado na modelagem do paralelismo.

**Condition(elemento, ...)**

Implementa uma estrutura de controle de seleção, simulando uma construção `IF...THEN...ELSE` aninhada. Assim, a expressão `condition(elem1, elem2, elem3, elem4)` é executada da seguinte forma:

```
SE elem1 é verdadeiro
  ENTÃO executa elem2
  SENÃO
    SE elem3 é verdadeiro
      ENTÃO executa elem4
      SENÃO retorna FALSO
    FIM-SE
  FIM-SE
```

Analogamente, a expressão `condition(elem1, elem2, elem3, elem4, elem5)` é executada como:

```
SE elem1 é verdadeiro
  ENTÃO executa elem2
  SENÃO
    SE elem3 é verdadeiro
      ENTÃO executa elem4
      SENÃO executa elem5
    FIM-SE
  FIM-SE
```

### **While(elemento, elemento)**

Implementa uma estrutura de controle de iteração. A expressão `while(elem1, elem2)` é executada como:

```
ENQUANTO elem1 é VERDADEIRO
  FAÇA executa elem2
```

Repare que a dinâmica da classe `processo` descreve a interação/cooperação entre suas atividades ou sub-processos.

## **2) Classe Atividade**

**Name:** identifica univocamente a instância dentro do processo.

**Status:** armazena o estado corrente da instância. Os valores possíveis são os mesmos assumidos pelo atributo `Status` da classe `processo`.

**Behavior:** expressa a dinâmica da instância da classe.

O atributo **Behavior** é formado por uma sequência de 2-tuplas, onde cada uma possui uma pré-condição e uma pós-condição. Uma pré-condição é definida como um conjunto, possivelmente vazio, de diretivas booleanas. Essas diretivas testam valores de atributos e diversas instâncias de elementos do processo. Cada valor de atributo é comparado com um padrão especificado no corpo da diretiva. Dessa forma, podem ser verificadas as ocorrências de relacionamentos entre instâncias e de propriedades das mesmas.

Uma pós-condição é um conjunto, também possivelmente vazio, de diretivas com capacidade para criar ou destruir instâncias, assim como modificar os valores de seus atributos. Podem, ainda, ser verificadas e sinalizadas situações de erro na execução das diretivas de pós-condição.

Quando uma atividade é executada, a sequência de 2-tuplas do atributo **Behavior** é avaliada. O comportamento da atividade é definido pela primeira 2-tupla cuja pré-condição for totalmente satisfeita e cuja pós-condição não sinalizar erro de execução.

As diretivas de pré e pós-condição podem ser vistas como sequências que, utilizando um estilo declarativo, executam inspeções ou atualizações em instâncias de elementos.

As diretivas de pré-condição verificam os valores correntes dos atributos, comparando-os com padrões fornecidos. O operador **check** é utilizado na implementação de diretivas de pré-condição. Sua sintaxe é:

`check(<nome da instância>, <operador booleano>, <padrão>)`

O argumento <operador booleano> pode assumir um dos seguintes valores: **eq** (igual), **ne** (diferente), **isin** (está em), **lt** (menor que), **le** (menor ou igual), **gt** (maior que) ou **ge** (maior ou igual). Neste caso os atributos **lt**, **le**, **gt**, e **ge** só se aplicam aos atributos numéricos.

Por outro lado, as diretivas de pós-condição atualizam valores de atributos e podem, até mesmo, criar ou destruir instâncias. Os operadores utilizados nessas diretivas são:

a) Add(<classe>, <instância>)

Cria uma instância de nome <instância> pertencente a classe <classe>.

b) Del(<instância>)

Destroi a instância de nome <instância>. A classe não precisa ser fornecida, pois os nomes de instância são identificadores únicos.

c) Upd(<instância>, <atrib>, <val>)

Atualiza o atributo <atrib> da instância de nome <instância> para o valor especificado em <val>.

d) Updvar(<instância>, level, [<incr>, <unid>])

Executa uma operação de soma ou subtração sobre o valor corrente do atributo level, que será visto adiante. O valor atual de level é incrementado de <incr> (ou de 1 caso este não seja fornecido) e <unid> especifica a unidade de medida a qual level se refere.

e) Ins(<instância>, <atrib>, <val>)

Insera uma valor <val> na lista de um atributo is-part-of ou has-part-of (que serão vistos adiante) na instância <instância>.

f) Ret(<instância>, ,atrib>, <val>)

Retira o valor <val> da lista de um atributo is-part-of ou has-part-of (que serão vistos adiante) na instância <instância>.

### 3) Classe Componentes de Dados

**Name:** identifica univocamente a instância.

**Status:** armazena o estado corrente da instância. Os valores possíveis são os mesmos assumidos pelo atributo Status da classe processo.

**Is-part-of:** indica quais as instâncias da classe componentes de dados em que a instância que está sendo descrita aparece como membro.

**Has-parts:** indica quais as instâncias da classe componentes de dados que aparecem como membro da instância que está sendo descrita.

Os atributos **Is-part-of** e **Has-parts** permitem caracterizar generalização e especialização, respectivamente, entre duas instâncias, colocando a disposição a descrição em múltiplos níveis de detalhamento.

### 4) Classe Recurso

**Name:** identifica univocamente a instância

**Status:** armazena o estado corrente da instância. Os valores possíveis são os mesmos assumidos pelo atributo Status da classe processo.

**Level:** é fundamentalmente um atributo numérico, normalmente utilizado para expressar propriedades passíveis de quantificação por valores ou escalas. O atributo possui um componente não numérico usado na especificação da unidade de medida associada a parte numérica.























































































