NOTAS DE AULA

Engenharia de Requisitos
Notas de Aula, Parte I

Julio Cesar Sampaio do Prado Leite
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

1994

1 Introdução

Um fato cada vez mais marcante é a constante dependência do mundo moderno dos sistemas de software. Os sistemas computacionais estão espalhados por todos os cantos, desde dos sistemas de frenagem dos carros até o controle dos fornos de microondas. A integração do mundo pelas telecomunicações com velocidades cada vez mais rápidas não pode prescindir dos sistemas computacionais de apoio, e cada vez mais telecomunicações confunde-se com a informática, na medida em que a maior parte das redes de são digitais e controladas por software.

A industria da informação emprega hoje, segundo dados de Bell [Bell 82], mais de 45 % da mão de obra americana (Figura 1). A indústria de software em 1990 era da ordem de 94 bilhões de dólares nos Estados Unidos e por volta de 46 bilhões de dólares na Europa. Hoje as grandes corporações sabem da importância estratégica de suas políticas em relação a tecnologias de informação e o papel fundamental do software como parte dessas tecnologias. Programas nacionais como o Software Engineering Institute e supra nacionais como o projeto ESPRIT tem procurado estudar em mais detalhe o processo de produção de software e transferir esta tecnologia para as organizações.

O produto software passa a ter cada vez mais valor, principalmente quando comeca a ser uma prática comum a proteção do software por instrumentos legais como a patente. A proteção do software por um mecanismo como a patente, além de demonstrar o valor que a sociedade está atribuindo ao software, suscinta também uma série de discussões sobre a pertinência desse tipo de mecanismo para proteção da propriedade intelectual.

Apesar dessa crescente participação no mundo moderno, o software como produto ainda utiliza processos de produção bastante insatisfatórios. São vários os episódios em que erros em software trouxeram prejuízos não só financeiros, como também de vidas humanas [Levenson 93]. Recentemente, a revista do grupo de interesse em Engenharia de Software da Association for Computing Machinery (ACM), SIGSOFT, vem listando em todas as suas edições casos de desastres em sistemas baseados em computador, sendo que muitos deles ocasionados por erros no software. Esses casos vão desde de casos de contas erradas, como por exemplo o caso de um casal de Tampa , Florida, do qual foi cobrado US$ 4,062,599.37 de conta de luz quando o certo era US$ 146.76 [Neumann 92], até o caso do Banco da America que gastou 23 milhões de

dólares em 5 anos do desenvolvimento do sistema MasterNet. O MasterNet ainda consumiu mais 60 milhões, mas mesmo assim não funcionou, tendo-se que retornar ao sistema antigo [Neumann 93]. Esses erros podem, portanto, impactar seriamente a qualidade do software. É função da Engenharia de Software evitar que estes erros ocorram, produzindo software mais robustos e provendo processos de produção mais confiáveis.

A Engenharia de Software como disciplina é relativamente uma área nova, tendo surgido organizadamente a partir de 1968 numa conferência patrocinada pela OTAN[1]. Apesar de consideráveis avanços, a Engenharia de Software ainda esta aquém das necessidades de qualidade hoje demandadas por sistemas cada vez mais complexos. Neste contexto existem consideráveis esforços de pesquisa e desenvolvimento com objetivo de aperfeiçoar o processo de produção de software, quer através de estudos teóricos, como de estudos aplicados.

O processo de produção de software tem seu início quando o que se quer é definido. Esta definição portanto é o ponto de partida do processo, ou seja a construção do software só pode ser iniciada quando se tem bem estabelecido o que se quer produzir. Quando os sistemas que requerem o apoio de software são sistemas complexos, esta definição do que se quer não é trivial. Em função disso muitos software, vide os exemplos acima, não se comportam como seria desejável. Fazer uma definição que cubra todas as necessidades de um sistema complexo é tarefa penosa. Desempenhar essa tarefa sem métodos, técnicas e ferramentas adequadas, torna a tarefa ainda mais penosa.

Neste escopo situa-se a Engenharia de Requisitos, enquanto uma área de pesquisa, que procura atacar um ponto fundamental no processo de produção que é a definição do que se quer produzir. Cabe a Engenharia de Requisitos como sub-área da Engenharia de Software propor métodos, técnicas e ferramentas que facilitem o trabalho de definição do que se quer de um software. Portanto a Engenharia de Requisitos tem uma interação muito forte com aqueles que demandam um software, quer estes sejam o mercado ou clientes de um aplicativo que será especialmente construído.

Sendo a Engenharia de Requisitos parte integrante do processo de produção de software, é importante abordamos uma série de conceitos ligados à Engenharia de Software para que possamos melhor entender a importância do processo de definição do software. Dessa maneira a Seção 2 revê alguns conceitos importantes sobre sistemas, a Seção 3 relata sobre o conceito de ciclo de vida e a Seção 4 expande este conceito e trata do processo de produção de software. A Seção 5 descreve um modelo (Sistema de Desenvolvimento de Software) que identifica as principais partes de uma organização que tem por objetivo desenvolver software. Finalizamos com um resumo do que foi apresentado.

2 Alguns Conceitos Importantes

Um dos conceitos mais importantes para a engenharia de software é aquele em que software é encarado como um produto [Gunther 78]. Software como um produto acarreta uma visão mais industrial do processo de criação de software. A visão como um produto também auxilia a correta identificação dos aspectos que são software e dos aspectos que pertencem ao universo no qual o software vai atuar. Conforme afirmamos anteriormente o produto software passa a ser cada vez mais um componente comum em uma série de outros produtos, desde de carros, fornos de micro-ondas, elevadores, telefones até sistemas de informação organizacionais. De um produto exige-se preço e qualidade, o que muitas vezes é negociado. Portanto como produto,

[1] Organização do Tratado do Atlântico Norte

software tem que ter o nível de qualidade exigido e procurar ser desenvolvido no menor custo possível. Esta é exatamente a função da engenharia, procurar sistemas de melhor qualidade dentro de um custo compatível com essa qualidade, otimizando a redução de custos.

Outro conceito fundamental para que se possa trabalhar com sistemas complexos é a Teoria Geral dos Sistemas (TGS). Essa teoria desenvolvida em meados do século vinte nasceu em função da crescente necessidade de tratar de ciências distintas de uma maneira interdisciplinar. Ludwig von Bertallanffy, um biólogo, objetivou ao propor a TGS produzir uma arcabouço teórico no qual diferentes conhecimentos poderiam ser integrados. A noção de sistemas e subsistemas pode ser considerada, hoje, de certa maneira como senso comum. No entanto, através de um melhor conhecimento de seus pontos fundamentais e das características básicas de um sistema podemos nos utilizar melhor desse ferramental indispensável para entendimento e modelagem de sistemas complexos. Abaixo detalhamos então esses aspectos.

Características Básicas de um Sistema:

- Propósito: Todo sistema tem que ter um objetivo.
- Globalismo: Qualquer mudança feita em qualquer parte do sistema, por menor que ela seja, altera o sistema como um todo.
- Entropia: Com o decorrer do tempo um sistema tende a degenerar. Essa degeneração ocorre quando não há suficente troca de informação do sistema com o meio ambiente.
- Homeostasia: Uma característica que aponta para a adaptabilidade dos sistemas. Todo sistema tende a se adequar ao seu meio externo.

Pontos Fundamentais da TGS :

- Objetos e Relações: Os objetos são os componentes básicos do sistema e de que maneira se relacionam.
- Limites: Talvez um dos pontos mais difícies de serem definidos, isto é qual a fronteira de um sistema? Como delimitar o que está dentro ou fora do sistema.
- Pontos de Vista: Todo sistema pode ser entendido ou observado de diferentes ângulos ou pontos de vista. A TGS considera que um sistema pode ser influenciado por pontos de vista.
- Nível de Abordagem: Todo sistema tem um nível de detalhe. O importante é assegurar que o nível de detalhe utilizado é condizente com o propósito do sistema.
- Hierarquia: A pedra fundamental da TGS na luta com a complexidade. A idéia de dividir um problema grande (sistema) em problemas menores (subsistemas) é intríseca a idéia de sistemas.
- Meio Ambiente: Todo sistema é um subsistema de um sistema maior. Este sistema se comunica com seu macrosistema através de entradas e saídas. É através das saídas que o sistema altera o estado do seu macrosistema.

É importante observar que tanto a visão de produto como a visão de sistemas nos ajuda a perceber que a definição de um produto de software exige a existência de um contexto onde ele será aplicado. Em alguns casos esse contexto é determinado por circunstâncias específicas com um mercado consumidor restrito e que demanda várias exigências (sob medida), em outros casos esse contexto é determinado por condições de mercado, na qual clientes individuais não têm como impor exigências. Lubars [Lubars 93] observou que este tipo de software têm um processo de definição diverso daqueles em que o mercado é restrito. Nos desenvolvimentos voltados para o mercado a liberdade dos arquitetos do software é muito grande, o contrário do que acontece quando o software é desenvolvido sob medida. Os sistemas de mercado mais restrito são também comumente conhecidos como software aplicativos, quer seja aplicado no tratamento de informações de uma organização, quer seja no controle de um artefato de engenharia.

Software aplicativos normalmente são definidos quer como função de um trabalho anterior de Sistemas de Informação ou de um trabalho de Engenharia de Sistemas. Ou seja, caso estejamos definindo um software contábil o faremos segundo a arquitetura do Sistema de Informação Contábil da organização. Caso estejamos definindo um software ABS para um automóvel estaremos utilizando a arquitetura do sistema de frenagem produzida pelos engenheiros de sistemas. É importante deixar claro que Engenharia de Software e Sistemas de Informação são especialidades diferentes [Leite 91a] (Figura 2) e que, a falta de cuidado em atacar problemas de informação organizacionais sem entender as atribuições dessas especialidades distintas pode acarretar em sérios problemas para a definição de software eficazes.

A grande diferença entre um sistema de informação e um software é que um sistema de informação não é somente o tratamento e processamento de informações, isto é um sistema de informação tem uma integração com a organização que vai além dos fluxos de informação, como por exemplo a criação ou modificação de processos da própria organização. Em vista disso , um sistema de informação tem em sua arquitetura aspectos não manipuláveis por um sistema de computação. Por outro lado, um software de apoio a um sistema de informação tem aspectos técnicos que longinquamente tem semelhança com a organização, tem aspectos de estreita ligação com o sistema computacional, além de tratar, é óbvio, somente dos aspectos computáveis.

Por outro lado um recente grupo de trabalho da IEEE Computer Society [IEEE 93] chegou a conclusão da necessidade de criação de uma nova disciplina chamada de *Systems Engineering of Computer-Based Systems* com o objetivo de resolver problemas de integração entre software, hardware e comunicações em sistemas complexos.

3 Ciclo de Vida

Desde dos primórdios a Engenharia de Software tem tido uma preocupação com o processo de produção que deveria apoiar. Muitas dessas discussões centraram-se, no início, no que se tornou popularmente conhecido como *ciclo de vida* do software. Este ciclo de vida, dependendo do autor, tinha mais ou menos fases. É importante salitentar, que quando utilizando a idéia biológica de ciclo de vida, tem-se como parte fundamental a fase *manutenção*, que é a maneira pela qual os sistemas de software combatem a entropia. Outrossim é também comum que muitas vezes o processo de produção seja descrito sem a preocupação explícita com a manutenção, nestes casos não é necessária a inclusão da fase manutenção.

A manutenção pode ser dividida [Leite 91b] em:

- corretiva (correção de erros oriundos do processo de construção),
- evolutiva (novas funcionalidades exigidas pelo ambiente externo),
- preventiva (mudanças visando a facilidades de mudanças futuras),
- adaptativa (mudanças necessárias em função das mudanças das plataformas de suporte, ou de hardware ou de software) e
- de suporte (visando principalmente monitoramento de performace do software).

Em seguida apresentamos algumas propostas de fases do processo de produção. A Figura 3 mostra um ciclo de vida típico. A Figura 4 mostra um processo de produção, onde a experimentação é o ponto principal. Esta estratégia foi proposta por Barry Boehm em 1988 [Boehm 88] como resposta as estratégias de prototipação (vide Figura 5). É importante salientar, que no que diz respeito a prototipação, deve-se fazer uma distinção entre a prototipação utilizada para o aperfeicoamento da arquitetura e aquela em que o protótipo torna-se o produto, o que em geral ocasiona sérios problemas de manutenibilidade. Na Figura 6 observamos uma visão da escola formal [Maibaum 87] sobre as fases do processo de produção. Segundo Maibaum e Turski, a especificação é a representação chave do processo de produção de programas. Na Figura 7 apresentamos o que consideramos ser uma forma canônica para a produção de software.

Uma outra visão do processo de construção de software é aquela representada pela Figura 8. Neste gráfico encontramos o processo de desenvolvimento sendo explicado segundo dois eixos. Um eixo diz respeito ao nível de abstração e o outro eixo diz respeito ao nível de formalidade da linguagem de representação utilizada. É interessante observar que, segundo essa figura, o processo *ideal* envolveria uma formalização do problema ainda em alto nível de abstração e que o processo normalmente seguido tem dificuldades principalmente porque utiliza descrições informais para descrever detalhes (baixo nível de abstração) do problema. É importante notar que em níveis altos de abstração têm-se ao mesmo tempo aspectos algoritmicos e outros não algoritmicos. Deve-e observar que a Figura 8 supõe que o processo possa produzir um ponto final (Meta) equivalente ao ponto inicial que é obrigatoriamente informal.

4 O Processo de Produção de Software

Apesar da existência de várias propostas como as vistas acima, na realidade ainda pouco se sabe sobre o processo de produção de software e muito do que se sabe ainda não é utilizado na prática de produção de software. Em função desse estado de coisas é que os Estados Unidos, o Japão[2] e a Europa[3] têm procurado organizar de alguma maneira esforços nacionais e supra-nacionais destinados a pesquisa e a transfência de tecnologia em Engenharia de Software. Nos Estados Unidos em particular o Software Engineering Institute, patrocinado pelo Departamento de Defesa, tem procurado desenvolver e transferir tecnologias de software para a indústria.

Dentro do espírito de aprimorar a capacidade americana de produção de software, a SEI desenvolveu um modelo chamado de Capability Maturity Model [Paulk 93]. Este modelo

[2]No Japão prossegue o projeto de quinta geração, hoje com mais enfâse em engenharia de software.

[3]Os projetos ESPRIT de integração Européia tem vários projetos de pesquisa e desenvolvimento na área de Engenharia de Software.

procura medir o estágio de maturidade de uma organizacao na produção de software. Essas estágios são de número de 5 e estão descritos na Figura 9. a classificação de uma organização produtora de software em um desses níveis é feita com base em um extenso questionário com perguntas sobre o uso de práticas de Engenharia de Software.

Numa amostragem feita pelo Software Engineering Institute onde este modelo foi empregado, concluiu-se que pouquissímas instituições tem um nível de maturidade entre 4 e 5. Portanto o estado da prática (o que o mercado utiliza) da Engenharia de Software nos Estados Unidos no fim da década de 80 estava aquém do estado da arte (o conhecimento que já se dispõe sobre o problema). Este estado de coisas é função não só da falta de métodos e técnicas como também de sua correta divulgação. Esta situação é basicamente a mesma na maior parte do mundo[4].

Apesar da pesquisas realizadas pelo SEI indicarem uma situação de uso muito aquém das possibilidades, visto as tecnologias disponíveis, é importante mostrar outros dados. Em pesquisa realizada em diferentes setores de produção de software [Moad 90] mostrou-se que a produção de software para Sistemas de Informação tem tido um grande aumento de produtividade (Figura 10). Este aumento de produtividade, decorrente das facilidades principalmente das linguagens de manipulação de banco de dados, não necessariamente implica numa organização do setor de produção de software condizente com altos níveis de maturidade à la SEI.

Uma das lições que se tem aprendido no decorrer do desenvolvimento de software, e hoje cada vez mais aceita, é o fato de que o processo de produção de software é um processo fortemente dependente de um fator social. Desenvolve-se software num contexto social, quer seja aquele em que clientes são os principais atores, como também dentro das próprias equipes de desenvolvimento. Este visão mais abrangente permite enxergar que o processo de produção de software não é simplesmente um processo técnico.

Levando-se em consideração o aspecto técnico, é importante observar os grandes avanços alcançados pela engenharia de software ao longo dos últimos anos. É importante ressaltar que hoje já se dispõe de bons programas tradutores e bons ambientes de apoio a esses tradutores[5]. É importante também ressaltar a disponibilidades de métodos e ferramentas de modelagem que tornam possível trabalhar na construção de software a nível de especificação [Harel, 92][6]. Não obstante esses truísmos muito ainda tem que ser pesquisado na direção de ambientes de desenvolvimento de software que procurem sanar vários dos problemas ainda existentes principalmente na integração entre linguagens de representação distintas e na interface com o usuário, isso sem falar dos problemas realmente difíceis, isto é referentes a disponibilização de *inteligência* nesses ambientes. Quando se fala na programação dos processos de software não se pode esquecer da advertência que fez Hebert Simon sobre a Engenharia de Software querer reinventar a Inteligência Artificial, isto é se quizermos automatizar partes do processo de software, então não podemos nos furtar a examinar os resultados (positivos e negativos) obtidos pela Inteligência Artificial.

Em um artigo escrito em meados de 1980, Fred Brooks responsável pelo desenvolvimento do IBM OS-360[7] e autor do livro `The Mythical Man Month`, escreveu o seguinte [Brooks 87]:

[4]Alguns estudos mostram que por fatores culturais, o Japão tem mais facilidade na sistematização dos processos de produção de software, o que não implica necessariamente em estar num nível de maturidade alto.

[5]Só como exemplo veja os ambientes de programação hoje disponíveis em plataformas PC (Turbo Pascal, Microsoft C,...) onde facilidades de edição e depuração estão incluídos

[6]Harel neste artigo chama essa modelagem de "vanila approach", referindo-se ao que é básico em termos de modelagem para especificações.

[7]O sistema operacional mais utilizado na década de 70

> The hardest part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other par is more difficult to rectify later.

É importante salientar que esse tipo de problema, considerado por Brooks como o problema essencial é aquele para o qual não existem soluções mágicas ("silver bullet") e que propostas recentes como por exemplo orientação a objetos lidam apenas com aspectos não essenciais (acidentais).

No início dos anos 90, David Harel emininete cientista de computação na área de algorítmos e autor do livro `Algorithmics: The Spirit of Computing`, escreveu um artigo que pretendia mostrar que a visão de Brooks era na verdade um tanto pessimista em função de alguns avanços importantes na área de modelagem e análise de modelos [Harel 92]. O argumento de Harel é que se 40 anos antes trabalhassemos com a hipótese de que o principal era conceber um algorítmo, como teria se alcançado progresso se não havia meio de expressar esse algorítmo? Portanto ela considera que os aspectos representacionais (acidentais) não devem de todo ser desconsiderados porque eles são decisivos em termos de produtividade, haja visto os progressos feitos até hoje, por exemplo: linguagens de especificação executáveis, tipos abstratos de dados, linguagens lógicas, editores gráficos, gerentes de configuração, entre outros. Ou seja os problemas de essência permanecem, mas hoje temos ferramentas muito mais poderosas. Harel acredita que com a disponibilidade de modelos que tratam dados, funcionalidade e controle e com facilidades de análise desses modelos estamos no caminho para um incremento substancial de produtividade na produção de software.

Um dos aspectos não tratados em nenhum desses importantes artigos é o fato de que ganhos consideráveis de produtividade podem ser alcançados simplesmente por políticas de gerência na produção de software. Como importante fundamento para esta observação está a constatação da eficácia de projetos de software que se utilizam da técnica de Inspeções. Inspeções é uma técnica proposta por Fagan [Fagan 86] que procura sistematizar um processo de revisão do produto software procurando atingir altos indíces de qualidade através da detecção e correção de erros. Acreditamos que a grande vantagem dessa proposta é o reconhecimento de que o produto software é fundamentalmente um conjunto de descrições produzidas por seres humanos e que um processo de garantia de qualidade depende também desses seres humanos.

Outro bom exemplo de que técnicas de gerência são eficazes na produção de software é o resultado alcançado por fábricas de software japonesas. Resultados de indíces de produtividade muito superiores do que as similares americanas indicam que as práticas gerenciais empregadas no Japão têm tido maior sucesso que as práticas adotadas em outras partes do mundo.

A produção de software vista sob o ângulo da engenharia invoca, como já dissemos anteriormente, o binômio, custo - qualidade. Produzir software de qualidade é uma meta básica da Engenharia de Software, que disponibiliza métodos, técnicas e ferramentas para este fim. Muito tem-se escrito sobre qualidade e seus vários adjetivos, no entanto o fundamental é que o software seja eficaz e dentro dos padrões (eficiência) exigidos pelo contexto onde o software irá operar. Por exemplo, Freeman [Freeman 87] faz uma distinção entre qualidade básica e qualidade extra. Em qualidade básica ele lista: funcionalidade, confiabilidade, facilidade de uso, economia e segurança de uso. Em qualidade extra ele lista: flexibilidade, facilidade de reparo, adaptabilidade, facilidade de entendimento, boa documentação e facilidade de adicionar melhorias. Este tipo de classificação nos fornece uma ídeia sobre o que estamos falando, mas é

importante ressaltar que a priorização dessas "qualidades" depende de cada caso e do custo associado a antigir cada uma dessas "qualidades". É importante ressaltar, no entanto, que cada vez mais a sociedade pressiona o setor de software para que a característica qualidade seja preponderante, normas internacionais como a ISO 9000 são o reflexo dessa pressão.

É importante salientar que a escolha de quais métodos serão utilizados no processo de produção é uma escolha gerencial ligada a qualidade. É também importante não esquecer que a produção de software é um processo que envolve como peça fundamental seres humanos. As tecnologias de produção de software, nas quais estão incluídas as tecnologias de gerência, tem por objetivo automatizar ao máximo a produção de software. Esta automatização proporcionará um aumento da produtividade da mão de obra empregrada, reduzindo portanto a demanda de técnicos.

A seguir, mostraremos uma visão global das partes de um sistema responsável pela produção de software. Este modelo procura identificar as principais partes de uma organização produtora de software, bem como suas principais atividades e relacionamentos.

5 Sistema de Desenvolvimento de Software

Um maneira eficaz de melhor enterdermos o processo de produção de software é se pudermos descreve-lo como um sistema. Dessa maneira propomos o Sistema de Desenvolvimento de Software (SDS). O SDS é um sistema composto de 5 subsistemas que retratam as principais partes de uma organização que pretende desenvolver software. Este modelo sistêmico teve uma grande influência da filosofia de sistemas e principalmente do livro de Peter A. Freeman, *Software Perspectives: The System is the Message* [Freeman 87] onde o tema é bastante bem explorado.

Na Figura 11 descrevemos o SDS através de um datagrama SADT [Ross 77]. O SADT é um método de descrição de modelos baseado na decomposição e no relacionamento das partes. O SADT utiliza-se de caixas e setas, sendo as setas classificadas como: entrada, controle, saída e mecanismo. De tal maneira que, não só, descreve-se entradas e saídas como também o fluxo de controle e os mecanismos utilizados pelas caixas. Um diagrama SADT pode descrever um sistema sob a perspectiva de atividades ou sob a perspectiva de coisas. Na Figura 11 usamos a perspectiva de coisas (datagrama). Neste modelo, as caixas correspondem a substantivos (objetos), enquanto as ações são descritas pelos fluxos de entrada e fluxos de saída. É importnante salientar que nesse modelo as entradas produzem os objetos e as saídas utilizam-se dos objetos das caixas para produzir ou influenciar outros objetos.

As entradas do SDS são: `estabelecer os objetivos` e `manter o estado da arte` e a saída é `produzir software`. Aqui, é importante perceber que a interface com o mundo externo, é feita através dos objetivos do Sistema de Software segundo a organização que hospeda esse sistema e através de uma absorção de conhecimento do mercado. Aqui vale notar que na prática é extremamente difícil se encontrar esse esquema em que os objetivos são pre-estabelecidos, muitas vezes os proprios engenheiros de software participam dessa tarefa que idealmente não lhes pertenceria. A outra entrada procurar acompanhar as mudanças de mercado e avaliar a possível adoção de novas tecnologias.

A seguir descreveremos suscintamente os Subsistemas do SDS:

- GERÊNCIA: Este subsistema é responsável pelo estabelecimento das políticas de planejamento e controle a serem utilizadas na produção de software.

- PESSOAL: Este subsistema lida com a peça chave do sistema de produção, as pessoas. Neste subsistema os assuntos relativos a motivação, poltica de pessoal, avaliação, treinamento e outros aspectos relacionados a pessoal são tratados, sem esquecer que a produção de software é um trabalho cooperativo.

- MÉTODOS: Este subsistema trata de materia fundamental para o estabelecimento do processo de produção. São os métodos os reponsáveis pelas regras e pelas técnicas a serem utilizadas em um projeto de software.

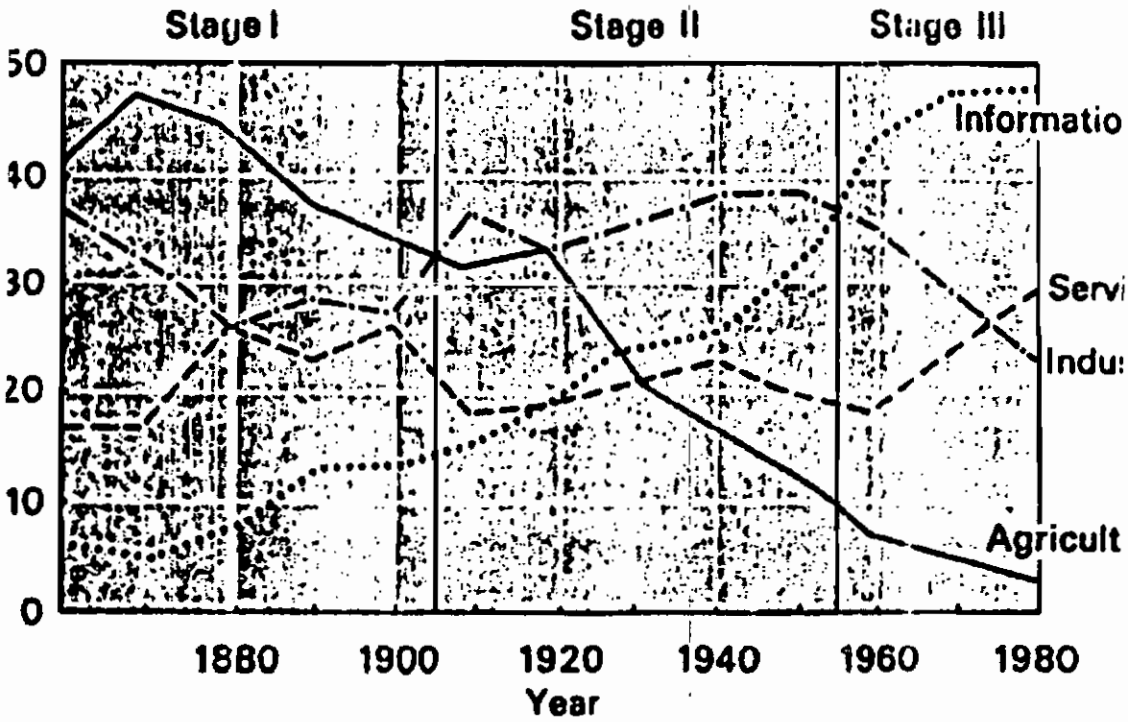
- FERRAMENTAS: Este subsistema é encarregado das ferramentas que dão suporte aos métodos e procuram aumentar a produtividade do sistema de produção. É importante saber que uma das atividades desse subsistema é a de seleção de ferramentas.

- INFORMAÇÃO: Este subsistema é responsável por prover informações sobre novas tecnologias, dessa maneira evitando uma degeneração do SDS. Este subsistema é também responsável pela retroalimentação da GERÊNCIA e pelo arquivamento dos produtos produzidos.

O SDS é um modelo com um ponto de vista gerencial sobre a produção de software e tem portanto o objetivo de facilitar a identificação dos aspectos mais relevantes para a criação e operação de uma estrutura de produção de software.

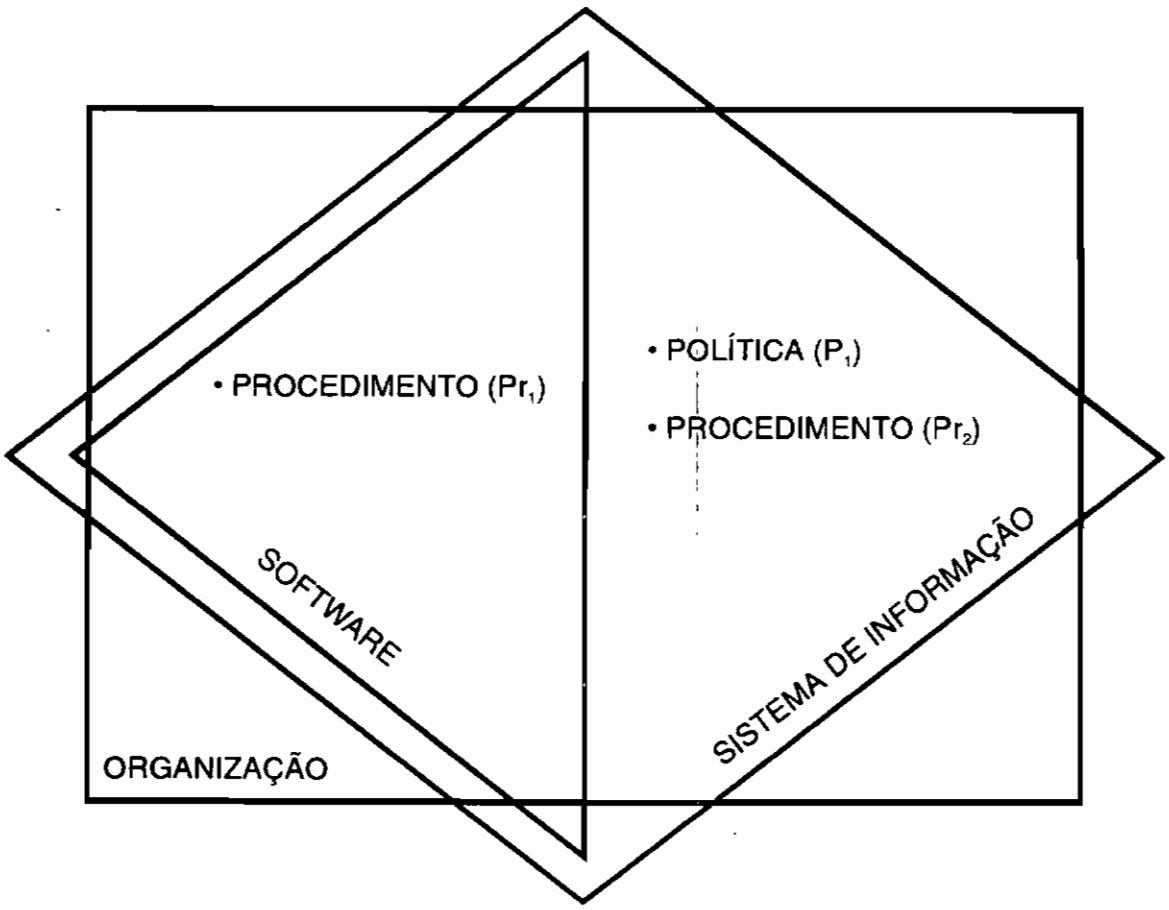
6 Resumo

O entendimento de Engenharia de Software como a disciplina que procura tornar mais eficaz o produto software produzido e mais eficiente o processo utilizado é fundamental para entender o papel da Engenharia de Requisitos. Entender que métodos são peças fundamentais na produção de software é também de muita importância, haja visto o crescente desencanto de várias organizações com ferramentas mais complexas de produção de software (ferramentas CASE). Sem métodos, ferramentas não têm utilidade. É importante também relembrar as dificuldades intrísecas na produção de software, como bem lembrou Brooks, mas também é importante saber que no início da década de 90 já dispunhamos de importantes modelos para representar informações mais abstratas, conforme lembra Harel. Sobre essa observação fizemos notar a importância do aspecto gerencial e mostramos uma proposta sistêmica com o objetivo de se melhor compreender o desenvolvimento de software (SDS). Cabe a Engenharia de Requisitos produzir descrições que representem o mais fialmente possível o esperado do software. Essas descrições se adendidas confirmarão a eficácia do software. Na parte seguinte procuraremos contextualizar ainda mais a Engenharia de Requisitos.



Daniel Bell *The Information Society* em "The Microeletronics Revolution" Forester (ed.) MIT Press 1982.

Figure 1: A Sociedade da Informação



P_1 → faturas acima de R$ 1000,00 têm que ser analisadas antes da remessa

Pr_2 → o departamento de contas a receber remete fatura > R$ 1000,00 a auditoria que verifica visualmente e remete para expedição

Pr_1 → IF TOTAL-FATURA ≥ R$ 1000 THEN MOVE FATURA TO FATURA-1000

Figure 2: Sistemas de Informação X Engenharia de Software

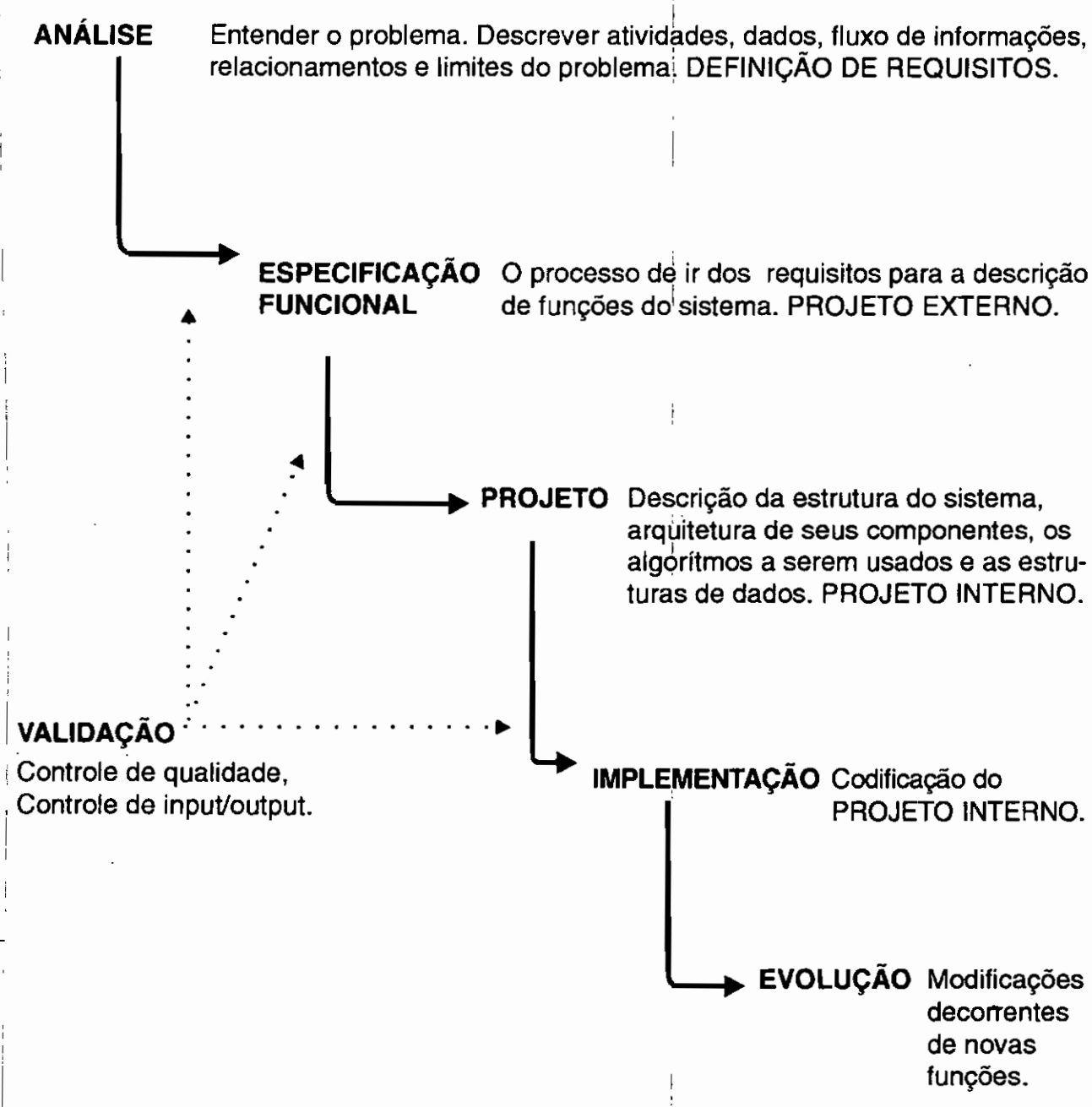


Figure 3: Ciclo de Vida

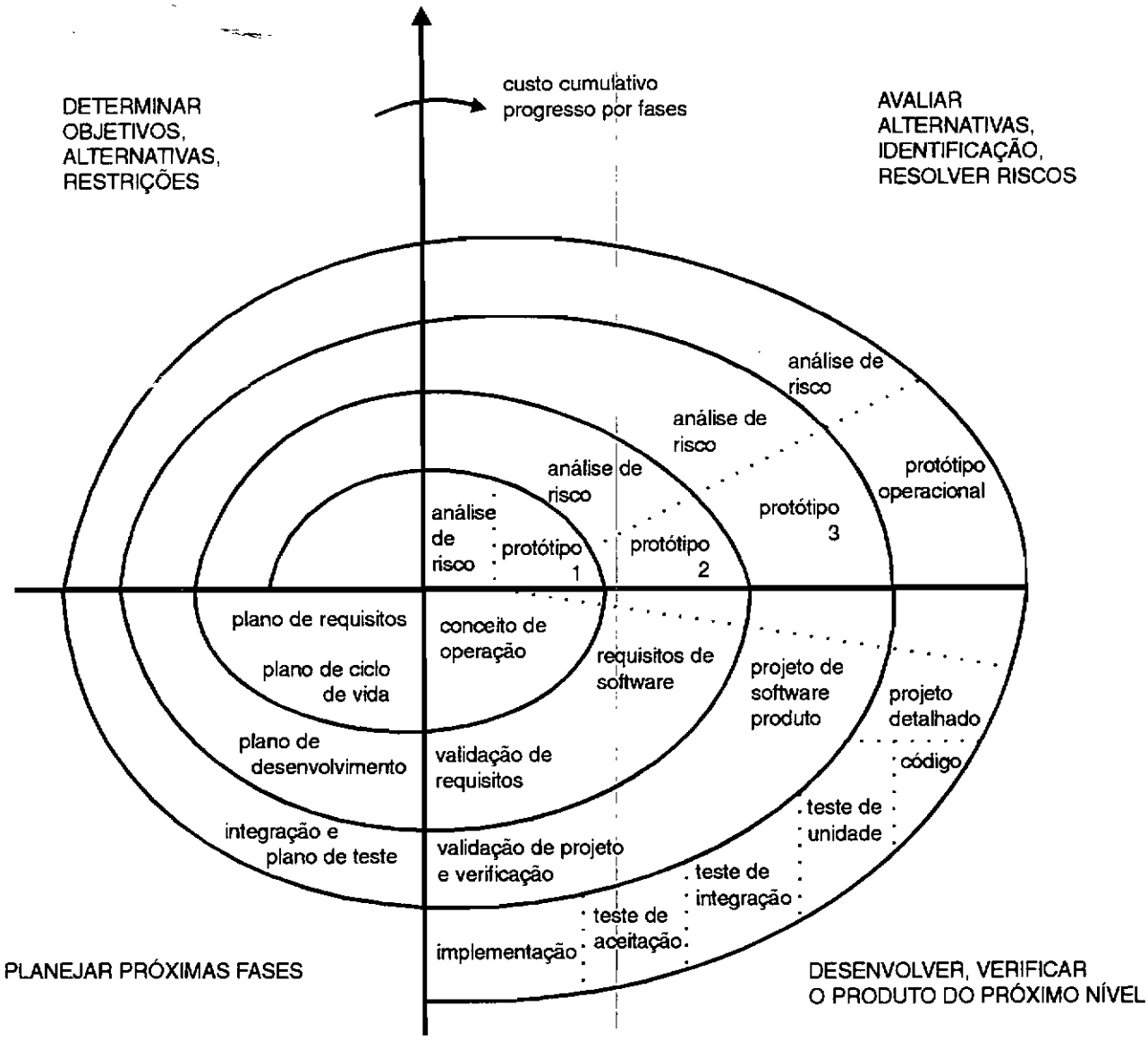


Figure 4: O Modelo Espiral

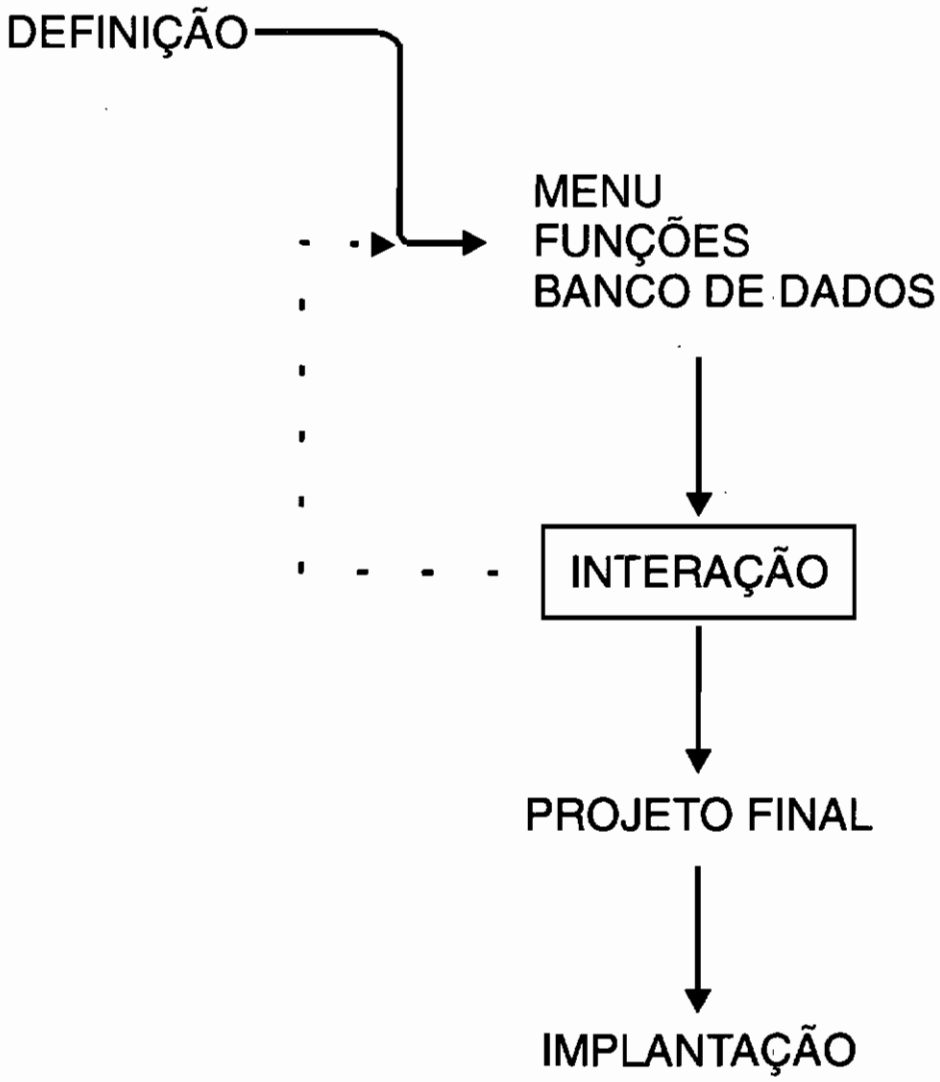


Figure 5: Prototipação

Figure 6: A Visão da Escola Formal

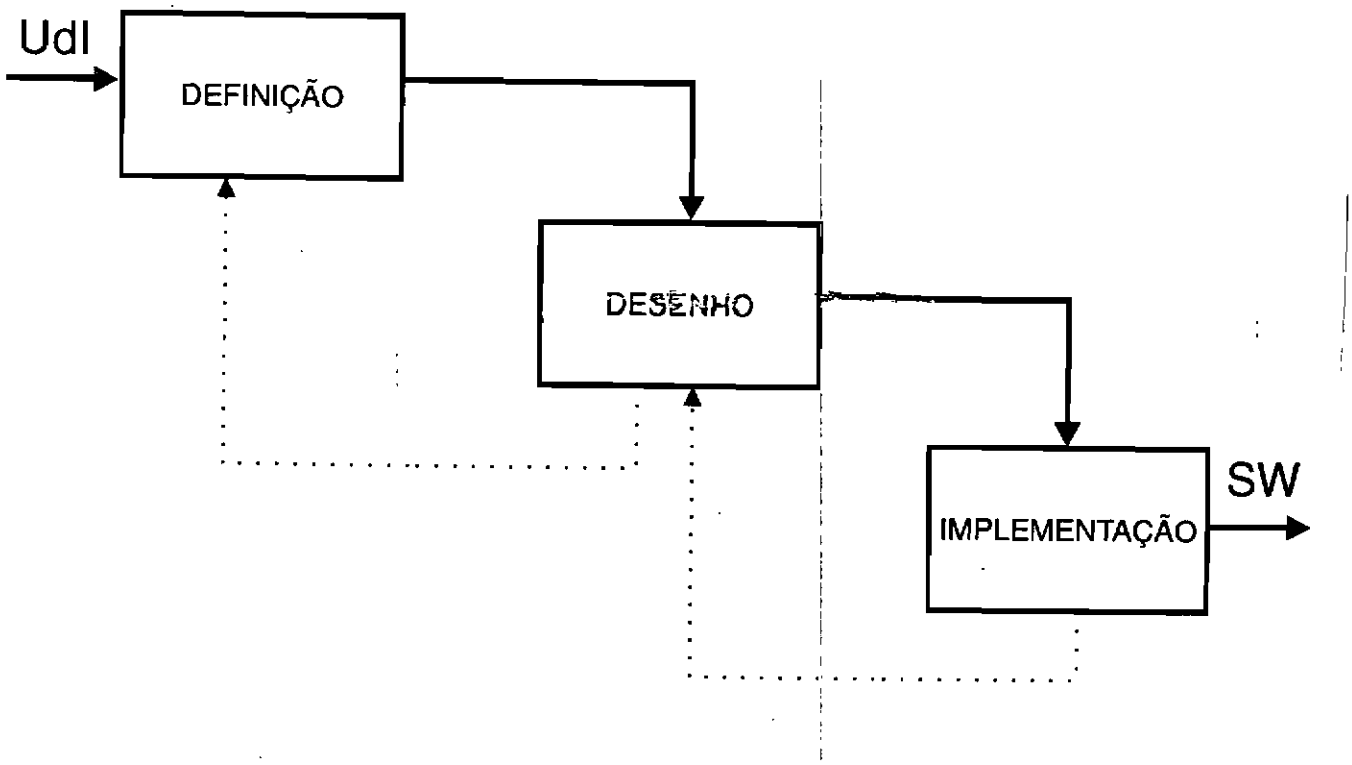


Figure 7: Produção de Software: Forma Canônica

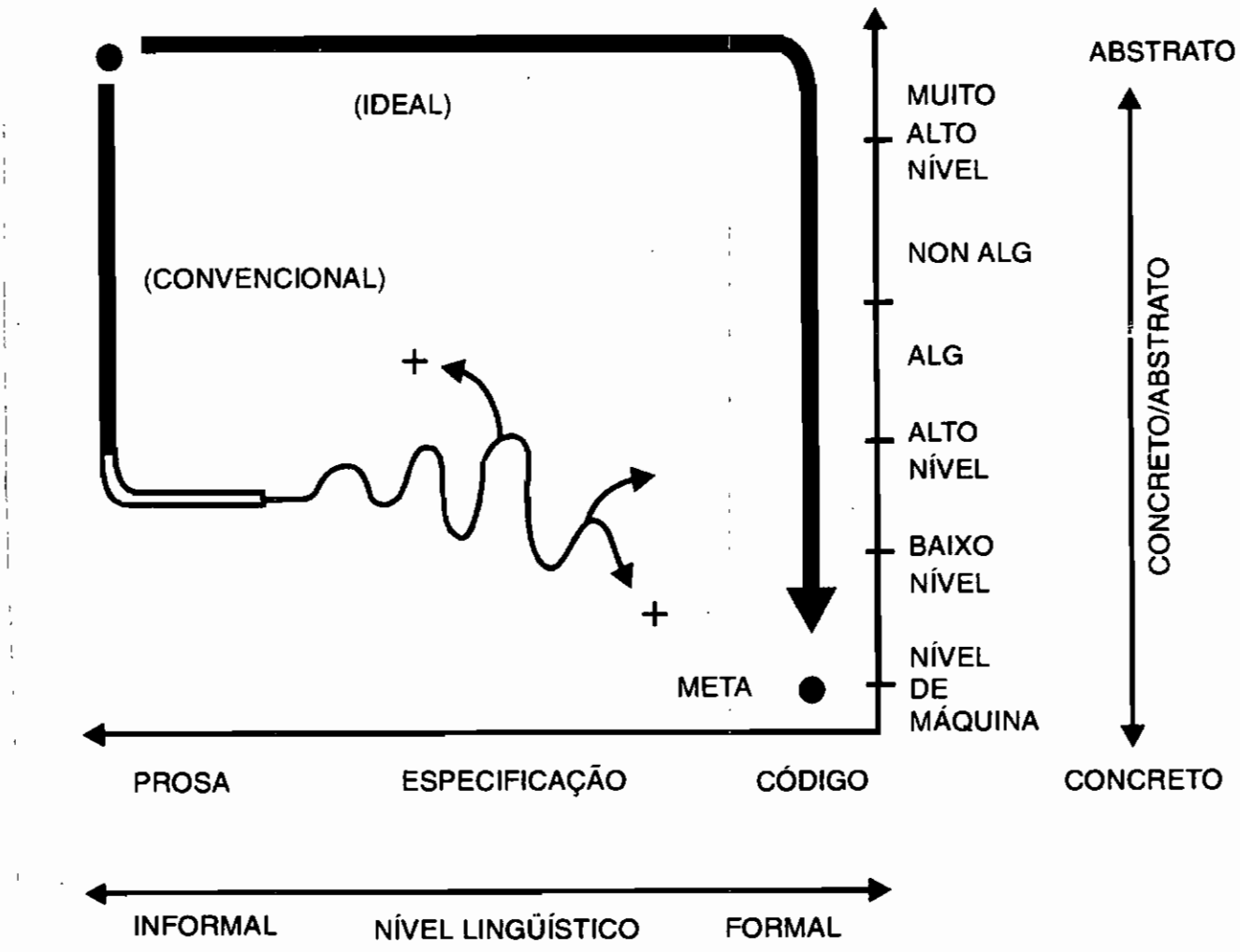


Figure 8: Abstração X Formalismo

NÍVEL 1:

- processos e controles mal definidos
- a organização não aplica gerência de E. S. ao processo
- a organização não utiliza tecnologia moderna

NÍVEL 2:

- a organização tem padrões para:
  estimativa de custos, mudanças em requisitos, mudanças em código e revisões

NÍVEL 3:

- melhorias incluem: revisão de código e de desenho, programas de treinamento em revisões, mais atenção a "E. S.".

NÍVEL 4:

- decisões baseadas em dados quantitativos, análise de dados, ferramentas para gerência e controle do processo de desenho, ferramentas para aquisição de dados. Previsão de falhas.

NÍVEL 5:

- alto grau de controle sobre o processo
- atenção para melhorias (otimização)
- prevenção de problemas

Figure 9: Capability Maturity Model

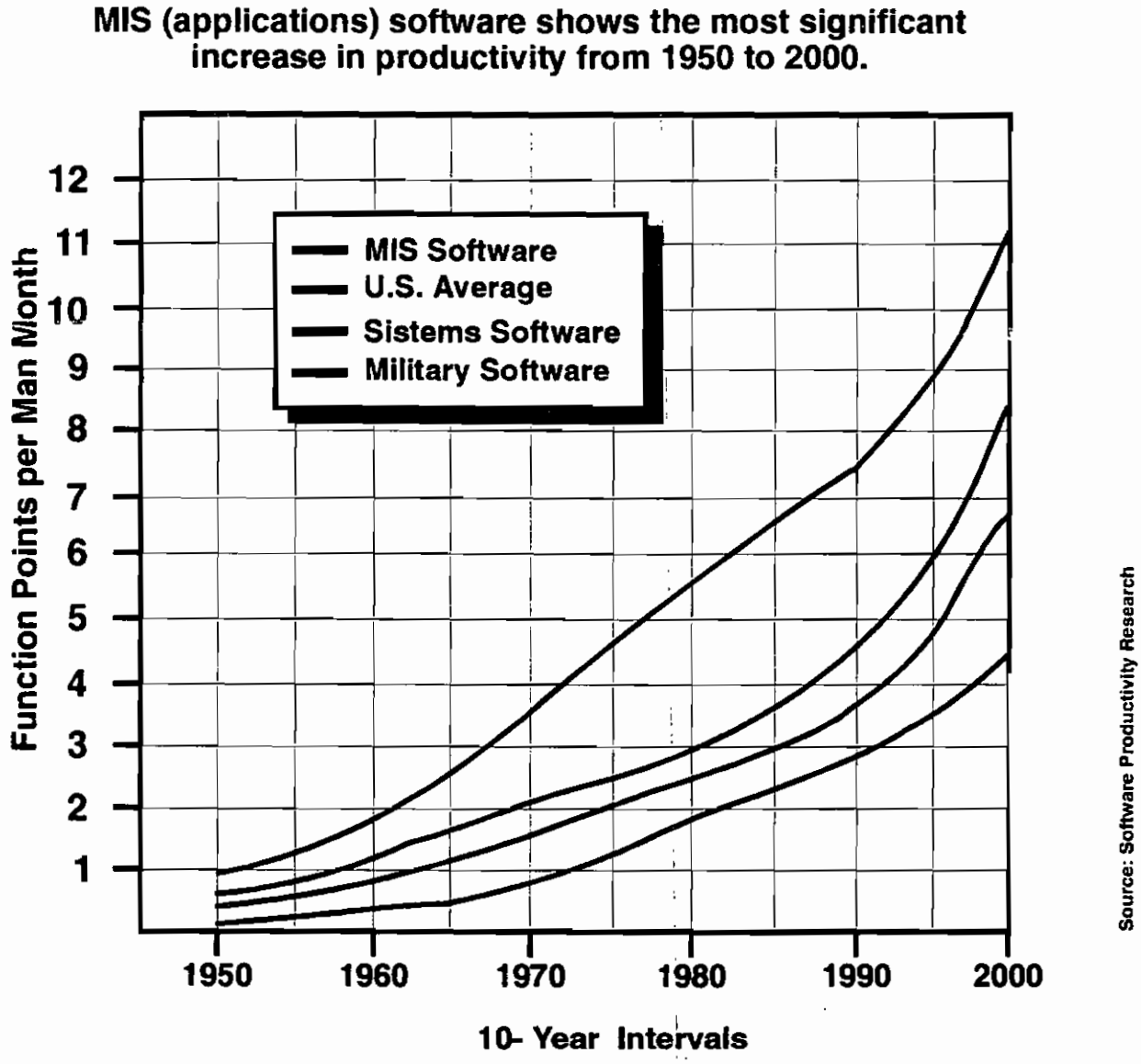


Figure 10: Produtividade de Software

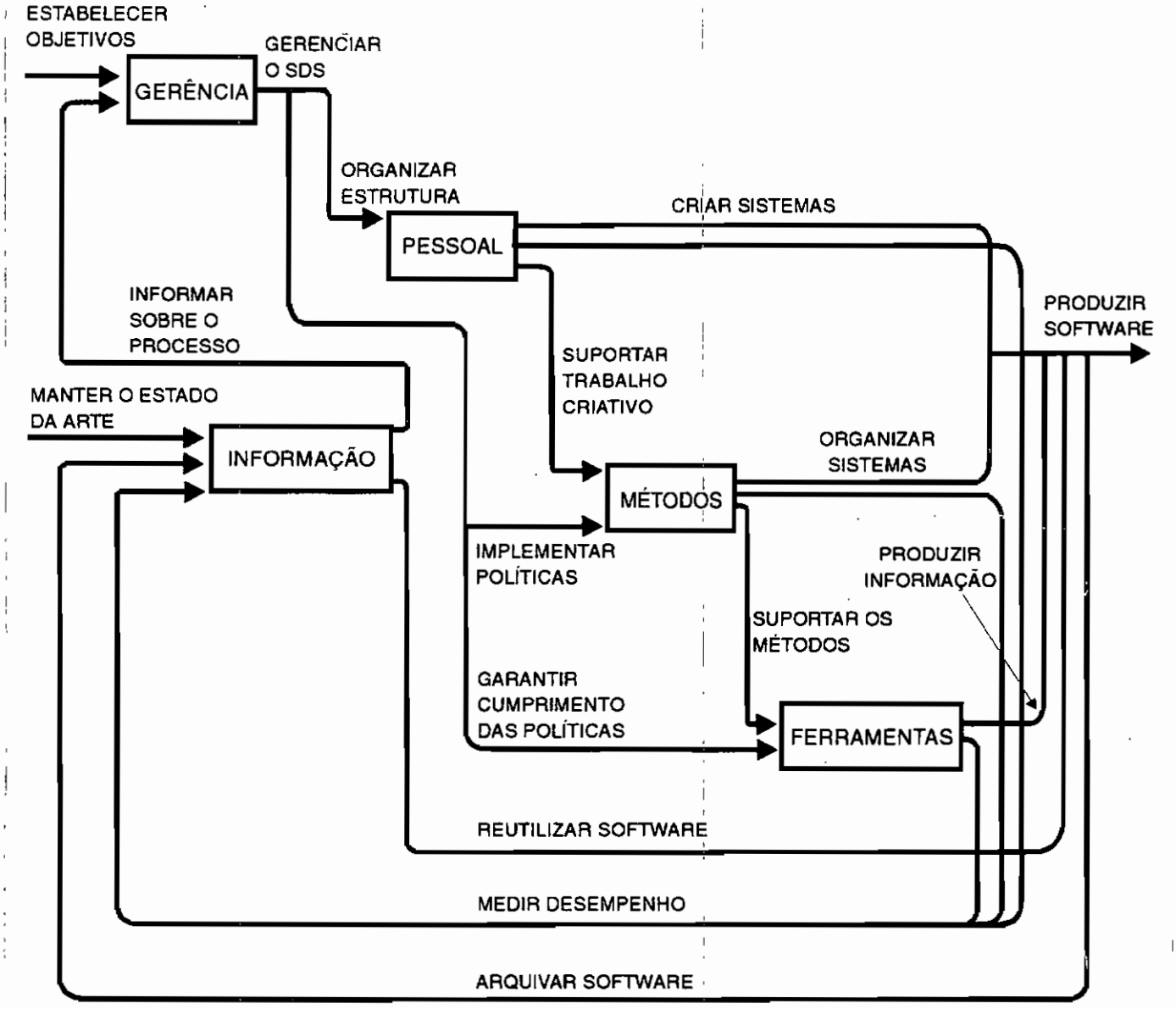


Figure 11: O Sistema de Desenvolvimento de Software

References

[Bell 82] Bell, D. The Information Society, *The Microelectronics Revolution*, Forester (ed.), MIT Press, 1982.

[Boehm 88] B.W., A Spiral Model of Software Development and Enhancement, *IEEE Computer*, vol. 25, n.5, May, 1988, pp. 61-71.

[Brooks 87] Brooks, F. P. No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Computer*, vol. 20, n. 4, Apr., 1987, pp. 10-19.

[Fagan 86] Fagan, M.E., Advances in Software Inspections, *IEEE Trans. on Software Engineering*, vol. 12, n. 7, Jul., 1986, pp. 744-751.

[Freeman 87] Freeman, P.A., *Software Perspectives: The System is the Message*, Addison-Wesley, Reading, Massachusetts, 1987.

[Gunther 78] Gunther, R., *Management Methodology for Software Product Engineering*, Wiley-Interscience Publication, Jonh Wiley & Sons, New York 1978.

[Harel 92] Harel, D. Bitting the Silver Bullet, *IEEE Computer*, vol. 25, n. 1, Jan. 1992, pp. 8-19.

[IEEE 93] IEEE Task Force on ECBS, Systems Engineering of Computer-Based Systems, *IEEE Computer*, vol. 26, n. 11, Nov., 1993, pp. 64-69.

[Levenson 93] Levenson, N. and Tuner, C. S., An Investigation of the Therac-25 Accidents, *IEEE Computer*, vol. 24, n. 7, Jul., 93, pp. 18-41.

[Leite 91a] Leite, J.C.S.P., Sistemas de Informação e Engenharia de Software, o Elo Gerencial, *Anais do XXIV Congresso Nacional de Informática*, SUCESU-SP, 1991.

[Leite 91a] Leite, J.C.S.P. e Souza, A. L.M.F., Re-Engenharia de Software, um Novo Enfoque para um Velho Problema, *Anais do XXIV Congresso Nacional de Informática*, SUCESU-SP, 1991.

[Lubars 93] Lubars, M., Potts, C. and Richter, C., A Review of the State of the Pratice in Requirements Modeling, *Proceedings of First IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, Jan., 1993, pp. 2-14.

[Maibaum 88] Maibaum, T.S.E. and Turski, W.M., *The Specification of Computer Programs*, Addison - Wesley, Workingham, 1988.

[Moad 90] Moad J., The Software Revolution, *Datamation*, Feb., 15, 1990, pp. 22-30.

[Neumann 92] Neumann, P.G., Aggravation by Computer: Life, Death and Taxes, *Communications of the ACM*, vol. 35, n. 7, Jul., 1992, p. 122.

[Neumann 93] Neumann, P.G., System Development Woes, *Communications of the ACM*, vol. 36, n. 10, Oct., 1993, p. 146.

[Ross 77] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. In *Tutorial on Design Techniques*, Freeman and Wasserman, Eds., IEEE Computer Society Press, Long Beach, CA 1980, pp. 107-125.

Engenharia de Requisitos
Notas de Aula, Parte II

Julio Cesar Sampaio do Prado Leite
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

1994

1 Introdução

A Engenharia de Requisitos é a disciplina que procura sistematizar o processo de definição de requisitos. Essa sistematização é necessária porque a complexidade dos sistemas exige que se preste mais atenção ao correto entendimento do problema antes do compromentimento de uma solução. A Engenharia de Requisitos procura trazer para a fase inicial do processo de produção procedimentos que reflitam a noção de engenharia, portanto diferente do processo artezanal. É sempre bom lembrar, Figura 1, que quanto mais nos afastamos da definição do produto, mais caro será corrigir problemas ou adicionar novas funcionalidades. O software por ser uma descrição da realidade, é portanto uma abstração de fenomenos observáveis. Esta característica explica a dificuldade de vizualização do produto software e ao mesmo tempo dá a falsa impressão de facilidade de alteração. O gráfico da Figura 1 reflete justamente isto, quanto mais detalhadas são as descrições, maior será o custo de modifica-las corretamente.

Na Parte I dessas notas, mencionamos que a DEFINIÇÃO DE REQUISITOS pode acontecer em basicamente 3 `ambientes`:

- em sistemas de informação,
- em sistemas de engenharia e
- em organizações que produzem software para o mercado.

Esses ambientes influenciam a escolha de modelos que serão tomados como base para o desenvolvimento de software. Por modelos, queremos nos referir a uma série de fatores que vão influenciar aspectos relacionados a produção de software. Uma analogia, num elevado nível de abstração, seria considerar modelos como a ideologia básica a ser utilizada. Dependendo da escolha do modelo, vão variar os métodos, técnicas e ferramentas utilizadas durante a definição de requisitos.

É importante ter claro que a definição de requisitos é a tarefa fundamental da Engenharia de Requisitos. Para que esta definição seja a mais eficaz possível, caberá aos engenheiros de software entender o ambiente no qual o produto software irá funcionar e escolher o modelo, ou modelos que melhor se encaixem no ambiente.

Vamos procurar definir de uma maneira abstrata o que entendemos por Engenharia de Requisitos. Nosso entendimento de Engenharia de Requisitos é fortemente baseado em nossos estudos e pesquisas da área e pretende facilitar a estudiosos e profissionais de informática a correta compreensão dessa importante matéria para o desenvolvimento de software. A Seção 2 cuida de apontar alguns aspectos fundamentais de contexto, principalmente exemplificando alguns dos modelos existentes. A Seção 3 discorre sobre algumas das práticas correntes, principalmente no ambiente de Sistemas de Informação. A Seção 4 apresenta a idéia de Universo de Informações, que contextualiza a prática da Engenharia de Requisitos. A Seção 5 define em mais detalhe a Engenharia de Requisitos. Finalizamos com um resumo do que foi apresentado.

2 Aspectos Básicos

Durante muito tempo a literatura em Engenharia de Software e a pesquisa procuravam centrar os esforços na modelagem, isto é procurando desenvolver linguagens e técnicas de representação que ao mesmo tempo que seriam mais abstratas, seriam também mais completas (isto é poderiam representar mais coisas) e principalmente procuravam ser formais. Para que uma linguagem seja formal é indispensável que sua semântica seja bem definida. É justamente isso, prover linguagens de alto poder de abstração com a semântica bem definida, o que poderia se caracterizar como a busca do gral na Engenharia de Software.

Em função disso, muito pouca atenção era dada a tarefa de coletar ou descobrir o que o sistema tinha de fazer, visto que o mais importante era a representação do sistema. A situação mostrada nas Figuras 2 e 3 ilustra este ponto. Um reflexo a confusão na literatura entre o que é requisitos e o que é especificação. Uma maneira de clarear este ponto é utilizarmos a definição do dicionário do Aurélio:

Requisito:

> Condição necessária para a obtenção de certo objetivo, ou para o preenchimento de certo objetivo.

Especificação:

> Descrição minuciosa das características que um material, uma obra, ou um serviço deverão apresentar.

É interessante fazer uma comparação entre alguns termos utilizados em Inteligência Artificial, Gerência de Projetos e Engenharia de Software. Em IA utiliza-se os termos: meta, plano e solução. Em Gerência de Projetos utiliza-se: objetivo, plano e resultados. Em ES pode-se utilizar:

- especificação, desenho (design) e código, ou
- requisitos, especificação e implementação.

Observe que temos vários termos que têm mesma semântica, mas na ES temos um termo, especificação, que tem uma semântica complexa, hora serve como meta, hora serve como plano. Peter Freeman apontou este problema de terminologia e considerou que no primeiro

caso o design estava implícito na especificação. De qualquer modo é importante notar que o primeiro termo em qualquer dessas listas, aponta para a questão fundamental segundo Polya: *What is the unknown?*

Outro ponto fundamental sobre esses termos é que o primeiro termo é a base para que se possa saber o grau de sucesso do último elemento. Por exemplo em Gerência de Projetos os resultados uma vez comparados aos objetivos, indicam o grau em que os objetivos foram alcançados.

Requisitos são ligados ao que se pretende do software, enquanto a especificação procura detalhar sob uma ótica computacional essas pretensões dos clientes. Se por um lado, especificação é distinta de requisitos, especificação também é distinta de desenho (arquitetura). Muitas vezes, no entanto, se utiliza a palavra especificação associada a requisitos (especificação de requisitos) ou a desenho (especificação de desenho), dessa maneira tanto os requisitos como a arquitetura ficam implícitos na especificação.

Portanto, uma vez entendido que os requisitos estão intimamente ligados aos objetivos e que a preocupação da Engenharia de Requisitos é justamente fazer com que esses objetivos estajam definidos da melhor maneira, temos que, primeiramente, nos preocupar com aquelas noções que nos permitem contextualizar o processo de definição de requisitos. Essa contextualização é, na verdade, uma tentativa de estabelecer os limites do sistema "definir requisitos". Acreditamos que, para que se possa realmente ser capaz de definir esses limites, é importante considerar alguns pontos ou conceitos listados abaixo.

- **Modelos**: É impossível definir requisitos sem que se utilizem modelos. Esses modelos servem para facilitar a descrição e a compreensão dos requisitos. Podem ser de várias espécies e de distintos níveis de abstração, dependendo do contexto onde é aplicado. O modelo fornece uma visão do mundo que vai nortear grande parte da tarefa de definir requisitos. A semântica que ligamos a palavra modelo é ampla o suficiente para encaixar como modelos, não só paradigmas diferentes, mas também algumas técnicas relevantes. Acreditamos que a escolha de modelos é na verdade uma escolha política, ou seja uma opção por uma filosofia. Idealmente devemos utilizar mais de um modelo, visto que isto pode nos trazer diferentes pontos de vista sobre os requisitos que queremos definir.

- **Falácia da Página Em Branco**: É um pressuposto errado, o de que podemos começar a elaborar um sistema partindo do conhecimento zero. Não só as pessoas que participam do processo têm um conhecimento anterior, como é impossível deixar de conhecer o ambiente no qual o processo de definir requisitos ocorrerá. Ou seja a definição de requisitos não acontece num vácuo.

- **A Definição de um Sistema é Função da Implementação do Macrosistema:** Conforme já visto, seria ideal se todo software já tivesse a implementação do seu macrosistema bem definida. Na realidade isso nem sempre acontece devido a recursividade dos processos de desenvolvimento (Figura 4) do sistema em relação aos seus subsistemas.

- **Um Sistema é Sempre um Subsistema de um Sistema Maior:** Como dissemos acima, não podemos começar do nada. O ponto fundamental (conforme visto na TGS) é o de que um software sempre será parte de algum sistema maior. Este conceito é bastante ligado aos dois últimos. A simbiose entre o desenho do macrosistema e o desenho do software é na, verdade, um processo extremamente complexo, porque a implementação total do macrosistema é função também da implementação do software.

No entanto, o que é importante ter em mente é que ao se definir um software, podemos gerar uma revisão em partes do desenho do macrosistema. Em empresas industriais esse fenômeno tem sido atacado pela idéia de *engenharia concorrente*, isto é em vez de utilizar um processo sequencial na composição dos desenhos, esses passam a englobar não só o macrosistema como também os seus sistemas.

- **Falácia da Completeza** Hoje sabe-se que exigir a completeza de um conjunto de requisitos é no mínimo ingênuo. O processo de definir requisitos é inerentemente incompleto, tendo em vista a grande complexidade do mundo real. É óbvio, no entanto, que não estamos lutando uma guerra perdida. Podemos atuar no sentido de termos requisitos o mais completos possívies. A falácia da completeza está intimamente relacionada com a observação de Fred Brooks (ver Parte I) sobre a bala de prata e a característica de homeostasia dos sistemas, ou seja, a característica evolutiva do software, que procura se adaptar ao seu macrosistema.

- **Aspectos Sociais** A definição de requisitos independentemente do ambiente, sempre depende de um contexto social. Esse contexto é muito importante no caso de software sob medida para sistemas de informação, mas também dever ser considerado quando desenvolvemos software para o mercado. O contexto social pode ser mais amplo, como no caso de sistemas de informação onde clientes têm papel importante, mas sempre deve ser considerado, mesmo que esse se restringa a equipe de desenvolvimento. Justamente por ser a parte do processo de produção de software onde se faz a fronteira com a realidade, a definição de requisitos tem, obrigatoriamente, que levar em consideração os aspectos sociais.

Sob o ponto de vista humano é indispensável ter em mente que os atores desenvolvendo o papel de engenheiros de requisitos têm que ter um correto domínio da tecnologia de Engenharia de Requisitos para que sejam os mais eficazes possíveis. Se por um lado, ainda estamos nos primórdios dessas tecnologias, também não podemos nos fiar na ídeia de que o profissional é um super homem. Essa idéia de super homem foi muito difundida na literatura clássica de análise de sistemas [Daniels, 71] e até em recentes livros de Engenharia de Software [Pressman, 90].

O conhecimento dos métodos, técnicas e ferramentas da Engenharia de Requisitos é fundamental, como também é fundamental o conhecimento de como usa-los da melhor forma. Sob esse aspecto é muito importante uma visão geral sobre modelos. Que tipos de modelos existem, quais seus objetivos, como emprega-los? Na Seção seguinte tocaremos em algumas dessas questões, através da apresentação de diversos modelos.

3 Modelos

Através da apresentação de alguns modelos procuraremos mostrar diferentes maneiras de se entender o contexto da Engenharia de Requisitos. Os modelos de que tratamos aqui são bastante abstratos e pretendem tornar mais claro o papel da tarefa de definir os requisitos de um sistema. É importante deixar claro que essa apresentação não pretende ser completa, apenas listamos algumas propostas para mostrar a diversidade entre elas e alertar para o aspecto de contextualização.

O modelo de Bowman, Davis e Wetherbe [Bowman 81], Figura 5, é um modelo aplicado em sistemas de informação e estabelece uma certa linearidade entre tarefas de PLANEJAMENTO,

ANÁLISE DE REQUISITOS DAS INFORMAÇÕES ORGANIZACIONAIS e ALOCAÇÃO DE RECURSOS. Este modelo contextualiza a Engenharia de Requisitos como dependende do planejamento e como sendo a base para a alocação de recursos. Entenda-se essa alocação de recursos, como os comprometimentos em função das decisões tomadas nos processos de desenho.

Outro modelo de Sistemas de Informação [Rockart 79] é o fator crítico de sucesso ou *critical success factors*. Os fatores críticos centralizam sua atenção naqueles setores da organização que são fundamentais para o sucesso da organização como um todo. Ao definir os fatores críticos podemos então procurar as informações que os apoiam. Neste caso portanto o contexto é definido por aqueles fatores fundamentais na organização, segundo a visão de seus dirigentes.

O modelo de Sá Carvalho [Carvalho 88], Figura 6, é um modelo que procura centrar o processo de entender as necessidades do sistema de informação na análise das ações executadas na organização. Identificando as ações de um sistema poderemos então saber quais as decisões que as originaram. Com esse conjunto definido, podemos então identificar quais as informações necessárias para a tomada dessas decisões. Desta maneira, Sá Carvalho define um modelo que procura as razões das saídas (produtos) de um sistema para que estas possam ser corretamente definidas. É interessante notar, que o método de desenvolvimento de software JSD [Jackson 83] também utiliza `ações` como base para definição de seu modelo, se bem que sem tecer as considerações expostas por Sá Carvalho.

Um modelo de engenharia de sistemas [IEEE, 93], Figura 7, centra sua atenção em três componentes básicos de sistemas baseados em computador: `hardware`, `software` e `comunicações`. A interface e o gerenciamento da informação são também relacionados, mas a enfâse é nos aspectos de integração hardware, software e comunicações. Portanto a definição do software tem que compreender bem a integração desses componentes.

Um outro modelo que podemos utilizar para facilitar o entendimento do contexto de requistos é o chamado gráfico de camadas. Originalmente proposto por Curtis [Curtis 88], o gráfico de camadas tem sido usado por nós para demonstrar a separação entre o sistema de software e o sistema de informação. Na Figura 8 podemos observar que fluxos de informação em um SI não necessariamente têm que passar pelo software. Nesse gráfico também observamos que diferentes atores (pessoas ou orgãos) tem níveis hierárquicos distintos e se comunicam de diferentes maneiras, muitas delas sem interferência do software.

O modelo proposto por Freeman [Freeeman 87], Figura 9, é um modelo sobre reuso. Apesar de ser bastante geral, levanta a questão da reutilização, ou seja de que maneira organizada eu posso/devo utilizar o conhecimento já disponível para aumentar a produtividade do processo de produção de software. Um modelo proposto por Neighbors [Neighbors 84], que procura tornar o mais eficaz possível essa idéia de reuso, é conhecido como o Paradigma Draco. Esse paradigma prega que antes de podermos escrever uma especificação para um determinado software, temos que ter disponível uma linguagem própria para este fim. As Figuras 10 e 11 descrevem as idéias básicas desse modelo. Este modelo encontra-se ainda em fase de pesquisas, mas espera-se que os ganhos de produtividade por ele oferecidos venham de certa forma revolucionar o processo de produção de software.

O modelo da Figura 12 [Penedo 88] é um modelo geral para arquiteturas de ambientes de software, isto é um contexto mais voltado para a produção de software. Esse modelo, ambientes de desenvolvimento de software, é base para softwares CASE, isto é software que procuram automatizar as tarefas de produção de software. A Figura 13 apresenta a arquitetura geral do ambiente Talismãn [Staa 92], um exemplo do uso desse tipo de modelo. O Talismãn é um software CASE desenvolvido na PUC-Rio e utilizado em várias empresas.

O modelo proposto por Balzer, Green e Cheatman [Balzer 83], Figura 14, é um modelo

de desenvolvimento de software baseado em bases de conhecimento. Apesar de não focar na contextualização da definição do software (parte de requisitos informais vindos de algum lugar) procura mostrar uma maneira de minimizar os problemas oriundos da definição através da proposta de fazer a validação e a manutenção ao nível da especificação.

Os três últimos modelos estão mais proxímos da produção de software e ajudam a entender o contexto da definição de requisitos no que diz respeito a sua ligação com o processo de desenho (design) do software. No que diz respeito às práticas comuns, isto é, as práticas de produção de software mais utilizadas nas empresas, vale a pena ressaltar três métodos de desenho lógico bastante populares.

- Análise Estruturada: que emprega as técnicas de diagrama de fluxo de dados, dicionário de dados, especificação de processos e diagramas de acesso de dados. O modelo base é composto de entradas, saídas, processos e buffers. A maior enfâse é no sistema e na sua decomposição.

- Análise Essencial: que emprega as técnicas de lista de eventos, diagrama de fluxo de dados, dicionário de dados, mini especificações e modelo de dados. Sua enfâse é no sistema e seu modelo base é o de sistemas reativos planejados.

- Análise de Dados: muito utilizado para a especificação de sistemas baseados em banco de dados. Fundamentalmente utiliza o modelo de entidades e relacionamentos (MER). Um MER é composto de entidades, seus relacionamentos e os atributos das entidades. O modelo basico é o próprio MER, que tem uma enfâse na fronteira entre problema e sistema, mas só cuida da parte estática.

Essa visão geral de modelos tem como propósito básico mostrar a complexidade e diversidade de propostas ou ideologias sobre o contexto do software e sobre sua produção. Os modelos apresentados são exemplos dessa diversidade, muitos outros existem, mas o que é importante é entendermos que a Engenharia de Requisitos deve procurar, antes de mais nada, escolher ou acolher o modelo ou modelos mais apropriados para o caso em questão. Na Seção seguinte definimos o termo Universo de Informações (UdI), que pretende, fundamentalmente, delimitar os contornos do macrosistema em que a definição de software ocorrerá. É no UdI que esses modelos são definidos e utilizados.

4 Universo de Informações

Os modelos apresentados anteriormente ajudam não só na definição do Universo de Informações, como também serão determinados por este mesmo universo. Vale lembrar que, o UdI é extremamente ligado aos conceitos vistos na Seção 2: completeza, página em branco, macrosistema, e a recursividade do processo de desenho. Conforme vimos acima é muito importante que antes de melhor definirmos a Engenharia de Requisitos procuremos entender o seu contexto, ou seja onde as tarefas dessa *engenharia* ocorrem e de que maneira poderemos nos situar, não só em relação ao processo de produção de software, mas principalmente em relação ao ambiente em que esse software vai trabalhar.

É fundamental que saibamos dizer algo sobre: o contexto em que estamos, os limites do nosso desenvolvimento e os objetivos do produto que estamos a desenvolver. Para que isso possa ser viável, é necessário que tenhamos definido um UNIVERSO DE INFORMAÇÕES.

Universo de Informações é o contexto geral no qual o software deverá ser desenvolvido e operado. O UdI inclui todas as fontes de informação e todas as pessoas relacionadas ao software. Essas pessoas são também conhecidas como os atores desse universo. O UdI é a realidade circunstanciada pelo conjunto de objetivos definidos pelos que demandam o software.

É importante saber que o UdI sempre existe. O UdI independe do modelo que estamos utilizando. Mesmo que o macrosistema não esteja bem definido, sempre podemos e devemos estabelecer os limites de nossa atuação. Por outro lado, é preciso sempre motivar as pessoas relacionadas ao software para a necessidade da definição do UdI. Quanto mais bem delineado um UdI, maiores são as chances de um software bem definido.

5 Engenharia de Requisitos

Através do estudo da literatura, dos vários modelos propostos e dos constantes problemas decorrentes da má definição do software, podemos afirmar a importância do estudo e do uso de técnicas que nos ajudem a produzir requisitos confiáveis. Dessa maneira surge a Engenharia de Requisitos, que procura sistematizar o processo da definição de requisitos.

A seguir daremos nossa definição para Engenharia de Requisitos, Figura 15, seguida de um modelo SADT [Ross 77], Figura 16, da tarefa de DEFINIR REQUISITOS. Neste modelo SADT devemos observar que a seleção do pessoal e dos métodos é função do SDS utilizado na organização que se propõe definir o software (vide o datagrama SADT do SDS na Parte I). O SADT aqui utilizado é o *actigrama*. No presente diagrama, as caixas têm verbos ou atividades como título, e a seta de controle passa a ser mais importante que a seta de entrada, visto que é o controle que determina as condições básicas de cada caixa no actigrama.

Portanto uma vez definido, os atores e os métodos, o processo de ELICITAR, MODELAR e ANALISAR se inicia. Este processo interage até que, por consenso dos participantes, chega-se a conclusão de que os requisitos já estão definidos. Vale observar que tanto o modelo como os requisitos são a saída do processo geral. Os requisitos são representados em linguagem natural e o modelo procura espelhar o mais corretamente possível sua semântica.

Na Figura 17 mostramos que o processo de definir requisitos **evolui**, ou seja, através dessa figura reconhecemos que o UdI muda e principalmente que o modelo resultante do processo de definição de requisitos é incorporado ao UdI, modificando-o. Esta realidade é muitas vezes desconsiderada através do que vulgarmente se procurou determinar como um congelamento dos requisitos. Ao mesmo tempo que a constante evolução pode gerar custos altos no desenvolvimento, é importante estar atento para que o congelamento não produza efeitos contrários, isto é aumentando consideravelmente o custo do produto final. O problema da evolução dos requisitos ou do rastreamento dos requisitos ainda é um tema sobre o qual pouco sabemos e que necessita mais pesquisa e desenvolvimento.

É importante deixar claro que, ao apresentar a tarefa de definir requisitos da maneira descrita acima, estamos fundamentalmente procurando melhor entender o processo como um todo. No entanto, estas divisões são por diversas vezes difícies de se distinguir na prática, visto que o enlace entre elicitação, modelagem e análise é muito forte. Portanto temos que entender que na realidade esta visão "limpa" não ocorre. A mesclagem das subtarefas da definição de requisitos é um dado do problema, mas a sua separação ajuda sobremaneira o correto entendimento do processo como um todo.

6 Resumo

A Engenharia de Requisitos tem como tarefa fundamental definir os requisitos de um software. Vimos que, para que esta tarefa seja adequadamente feita é necessario o correto entendimento do aspecto de contexto, ou seja onde realizar essa tarefa, quais os recursos, quais os limites. Foram mostrados alguns modelos que pretedem abordar de diferentes maneiras a interação do software (e portanto de sua definição) com o mundo que o cerca. Definimos aqui o objetivo da Engenharia de Requisitos e apresentamos um modelo SADT que procura delinear os relacionamentos entre as várias tarefas da definição de requisitos.

Página em Branco

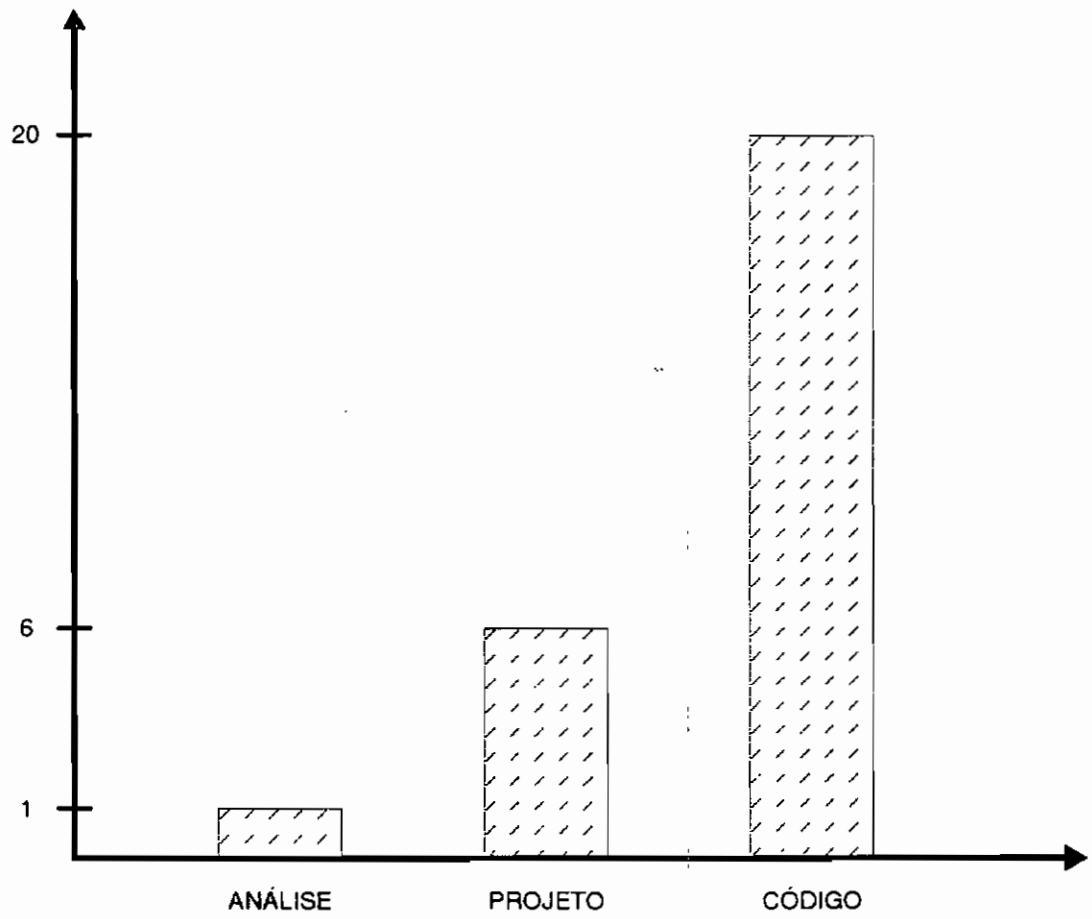


Figure 1: Custos de Correção de Erros X Etapas do Processo

Era uma vez...

O sistema que queremos
deve fazer isto, isto...,
mas nesse caso... e deve,
também, fazer isto.

Sim, sim. Estou
anotando.

Conversei com os usuários e
basicamente este é o sistema
que teremos que desenvolver.
Aqui estão minhas anotações.

Sim chefe.

Ótimo, começaremos a
análise imediatamente.

Fique tranqüilo,
usaremos o nosso
sistema CASE.

Figure 2: Um Cenário Comum (Parte I)

Quatro meses depois...

Srs. Usuários, após o emprego das mais modernas técnicas de especificação, produzimos este documento que descreve minuciosamente o sistema.

Ótimo. Bem, hum...
é um documento com 300 páginas e todos estes gráficos, tabelas. Enfim, vamos analisá-lo e voltamos a falar.

Depois de um mês e meio...

Sr. analista, nosso pessoal analisou com cuidado o documento. Tivemos muita dificuldade em entender tantas bolas e setas, máquinas de estado, e dicionário de dados. Enfim custou-nos muito. Além disso, tivemos muitas dúvidas. Mas o que percebemos é que NÃO FOMOS CORRETAMENTE ENTENDIDOS!

Como não? Tudo que aí está, foi fruto de nossas conversas e de conversas entre seu pessoal e meus analistas! REALMENTE VOCÊS NÃO SABEM O QUE QUEREM!

Figure 3: Um Cenário Comum

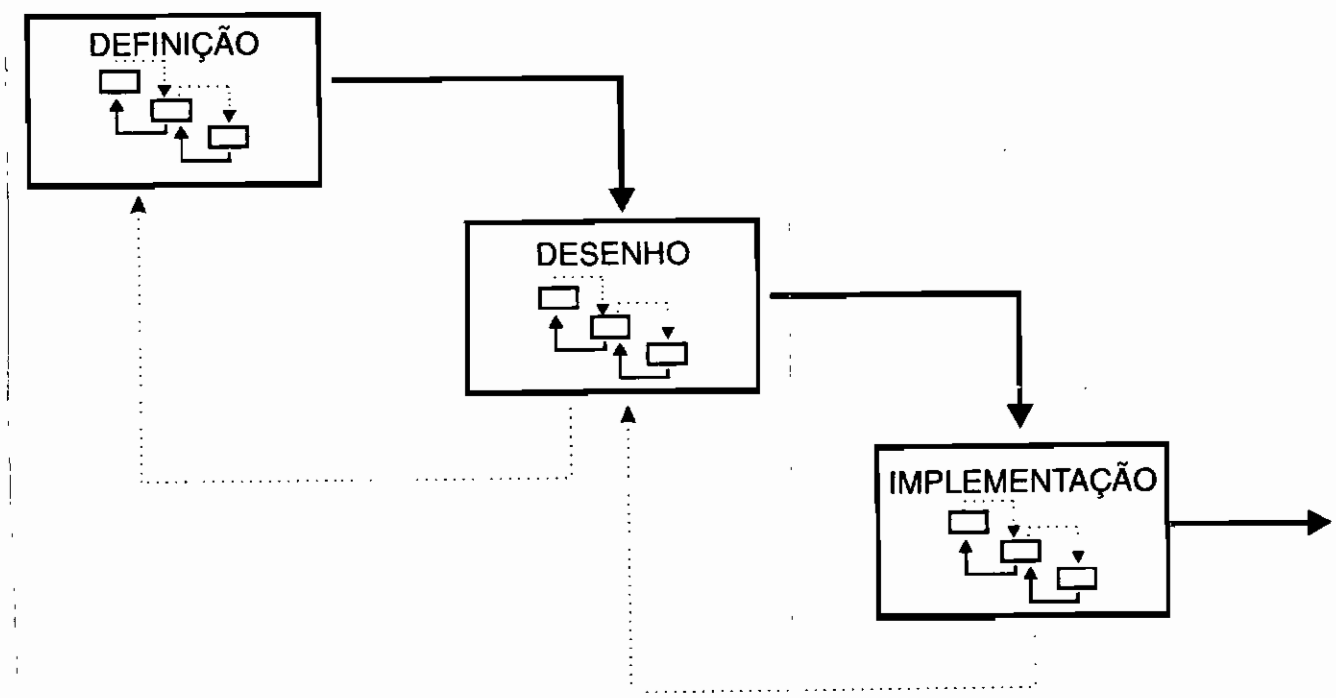


Figure 4: Recursividade entre Níveis de Abstração no Processo de Desenvolvimento de Software

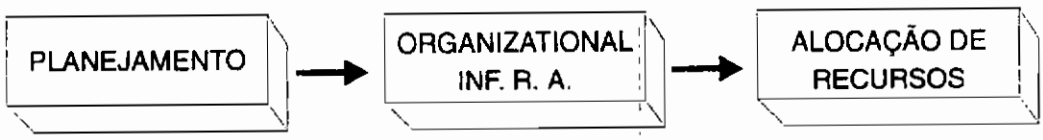


Figure 5: Modelo de Bownam, Davis e Wetherbe

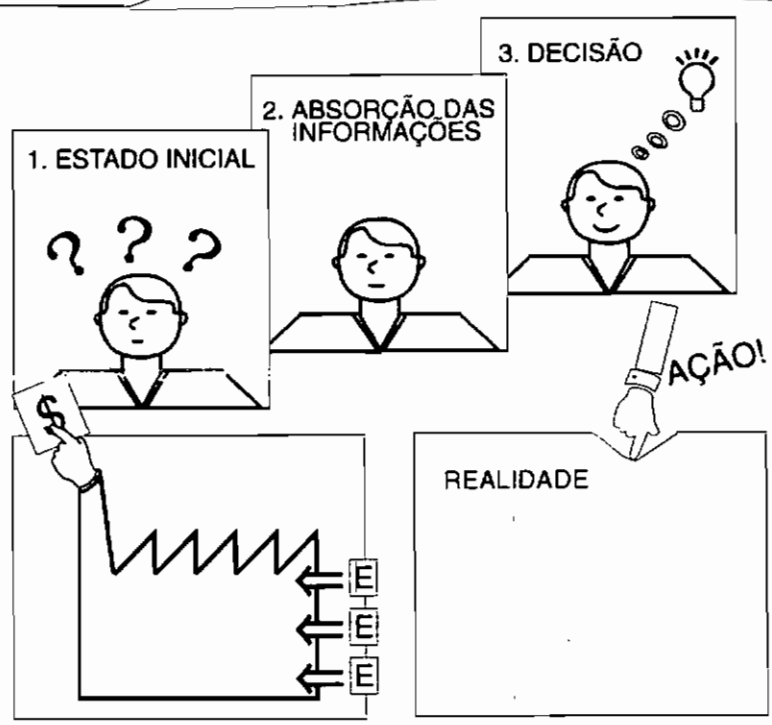


Figure 6: Modelo de Ações

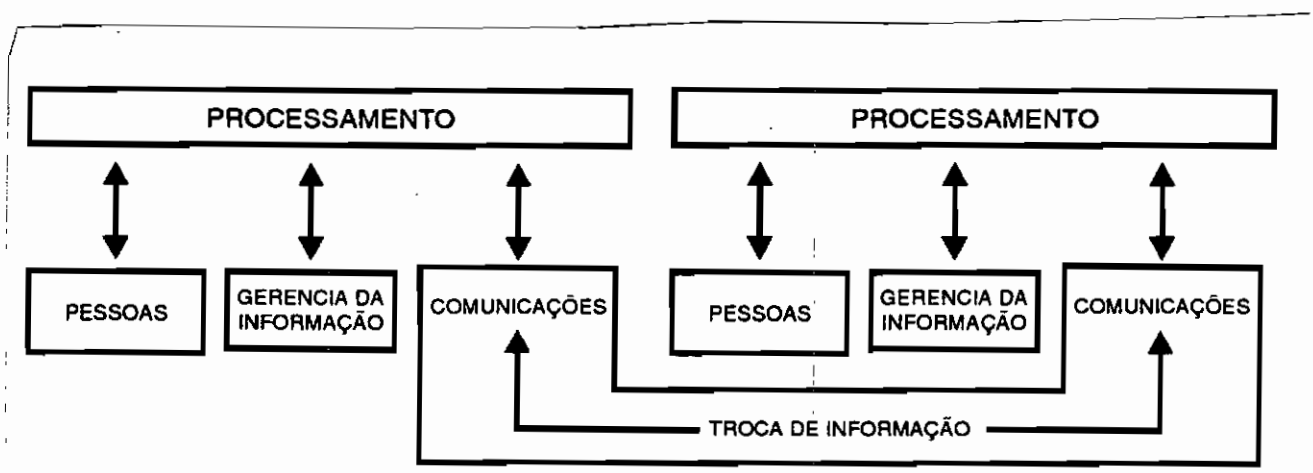
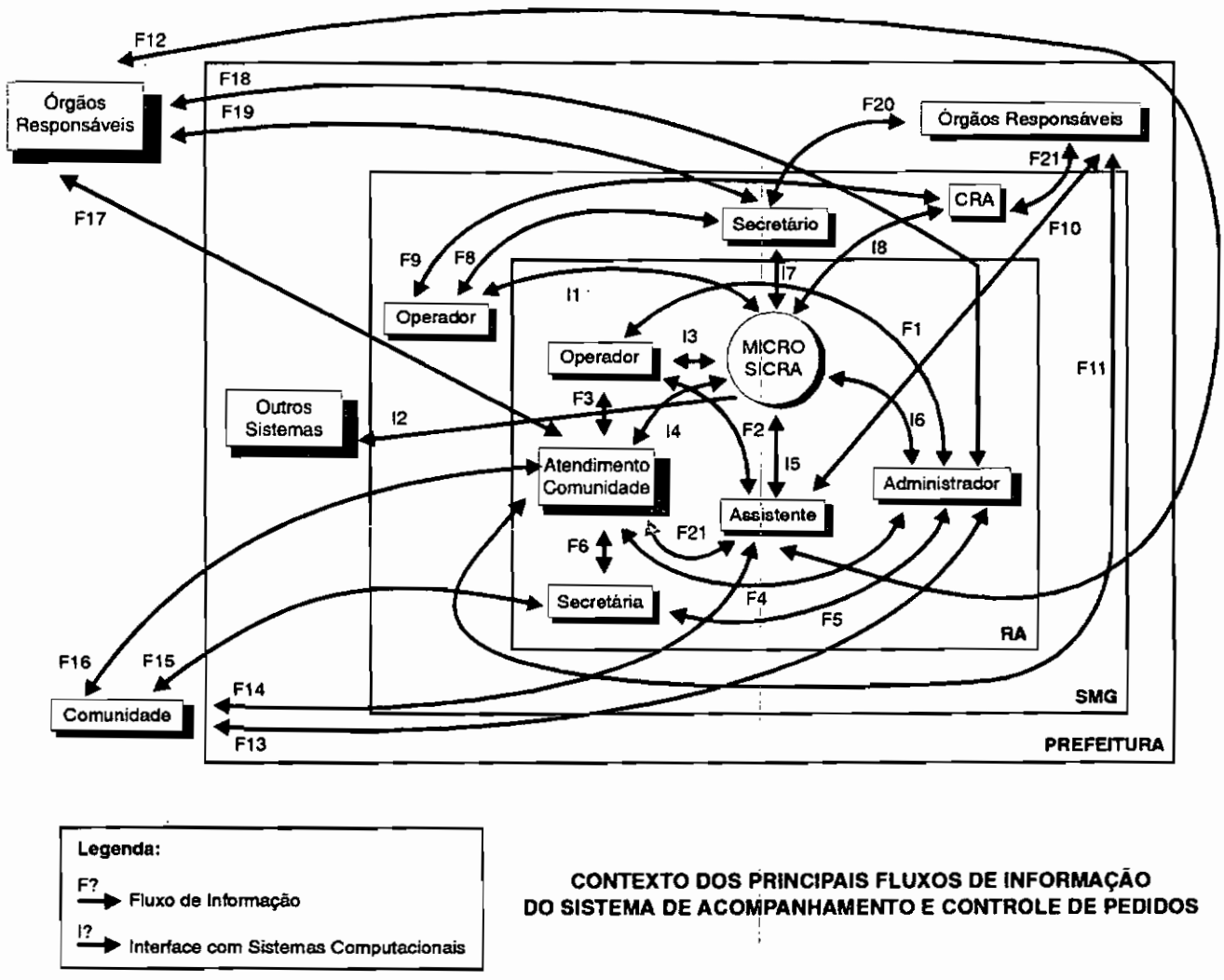


Figure 7: Modelo do IEEE Computer Task Force on ECBS



CONTEXTO DOS PRINCIPAIS FLUXOS DE INFORMAÇÃO DO SISTEMA DE ACOMPANHAMENTO E CONTROLE DE PEDIDOS

Figure 8: Gráfico de Camadas

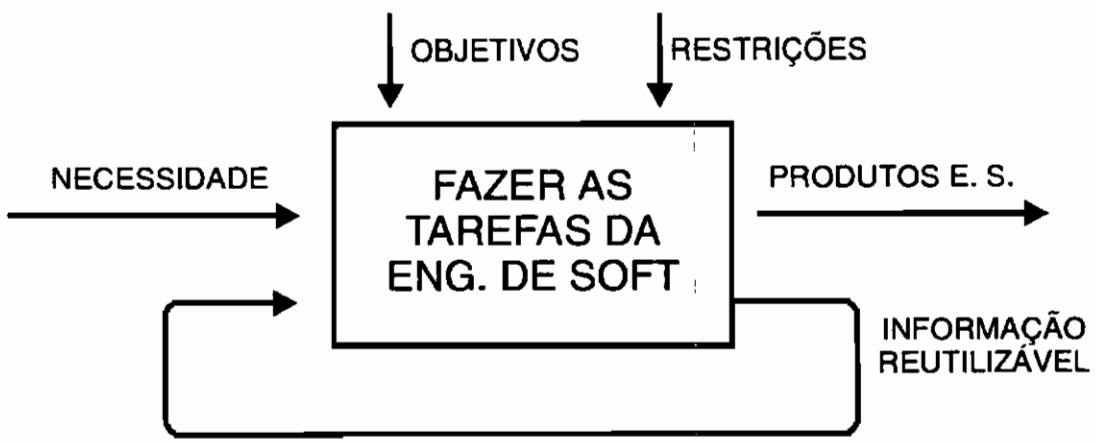


Figure 9: Modelo de Reuso

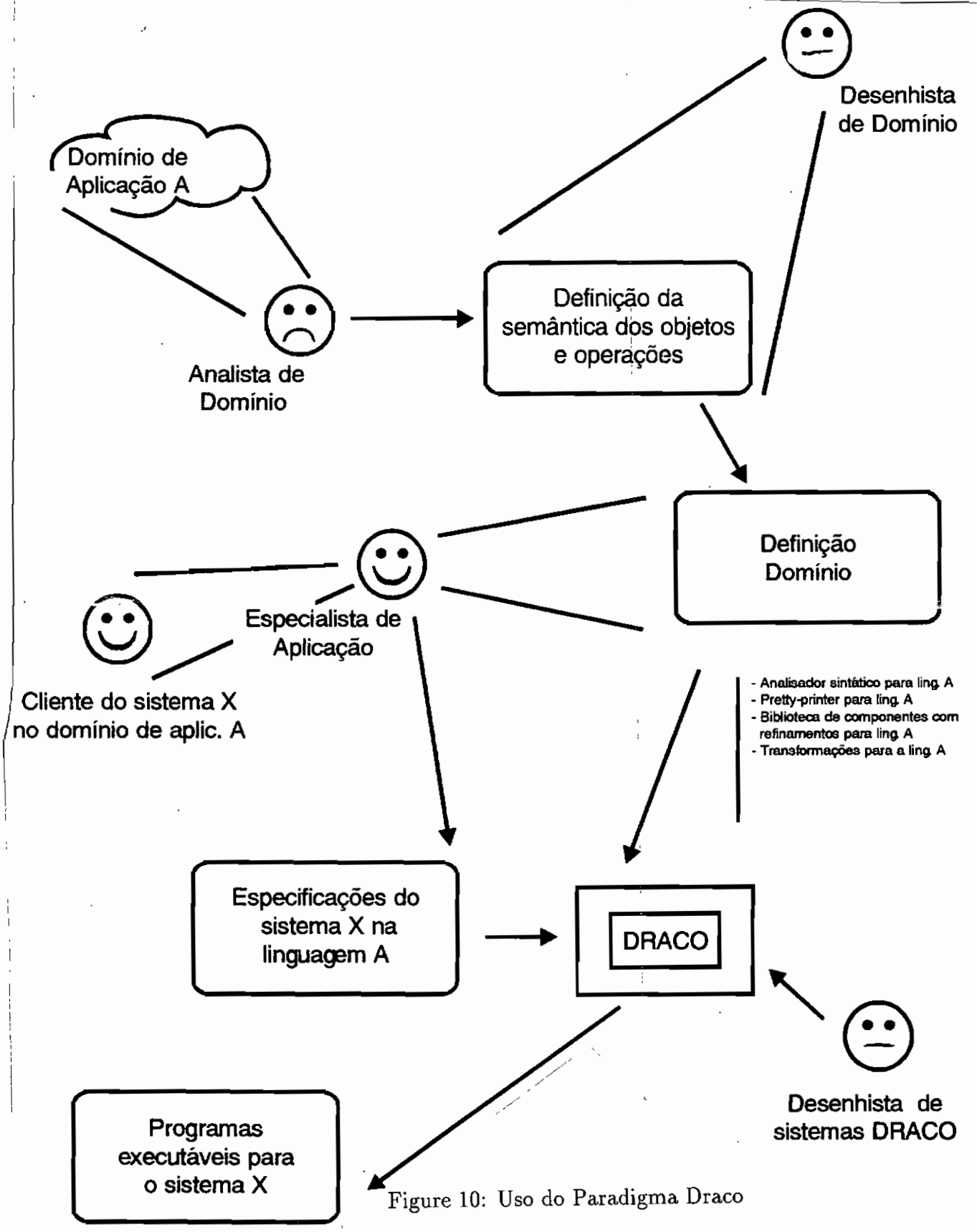
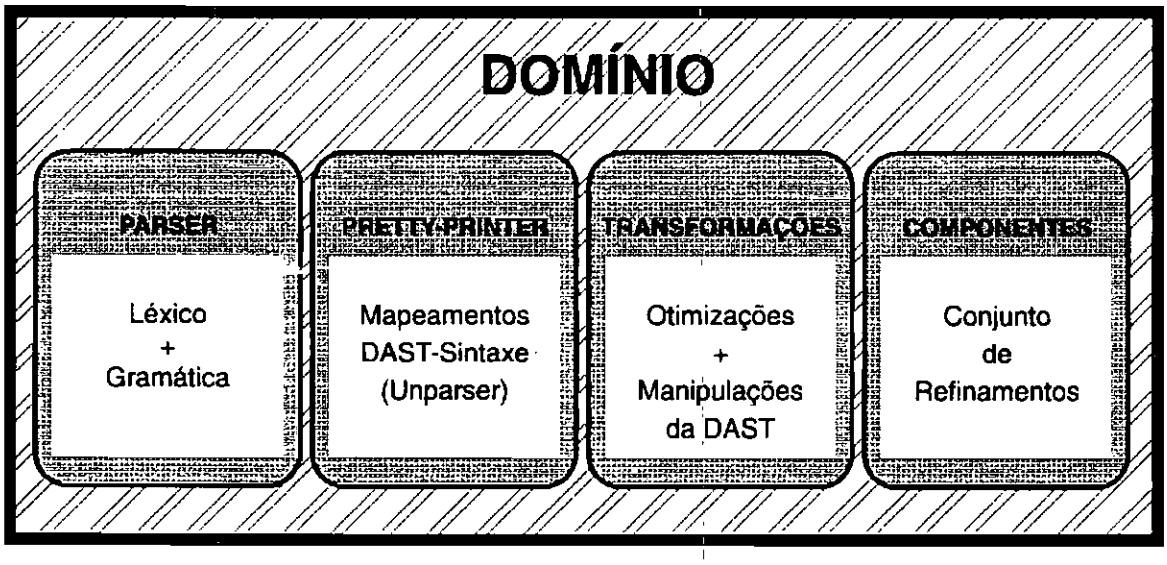


Figure 10: Uso do Paradigma Draco



ESTRUTURA DE UM DOMÍNIO

Figure 11: As Partes de Draco

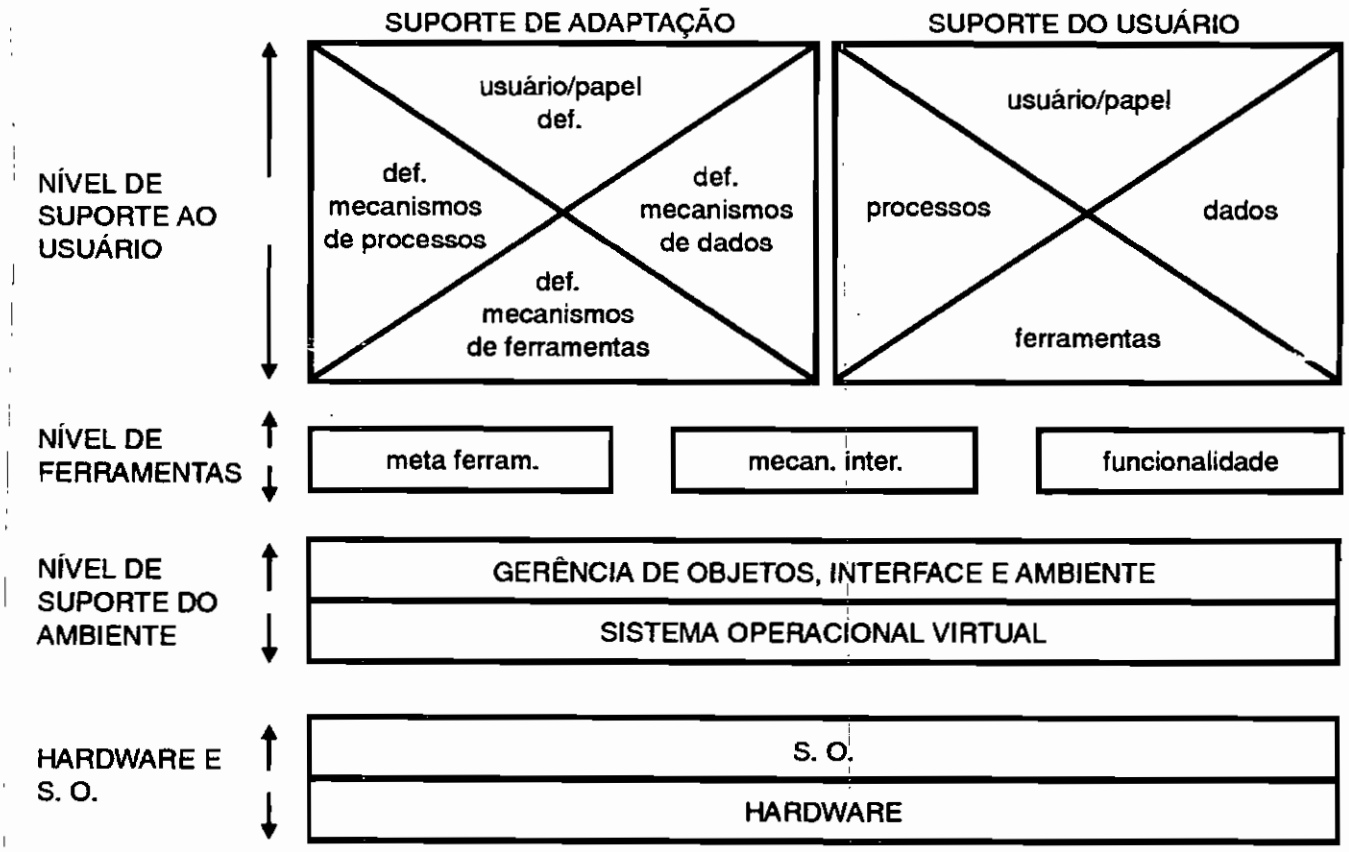


Figure 12: Ambientes de Desenvolvimento de Software

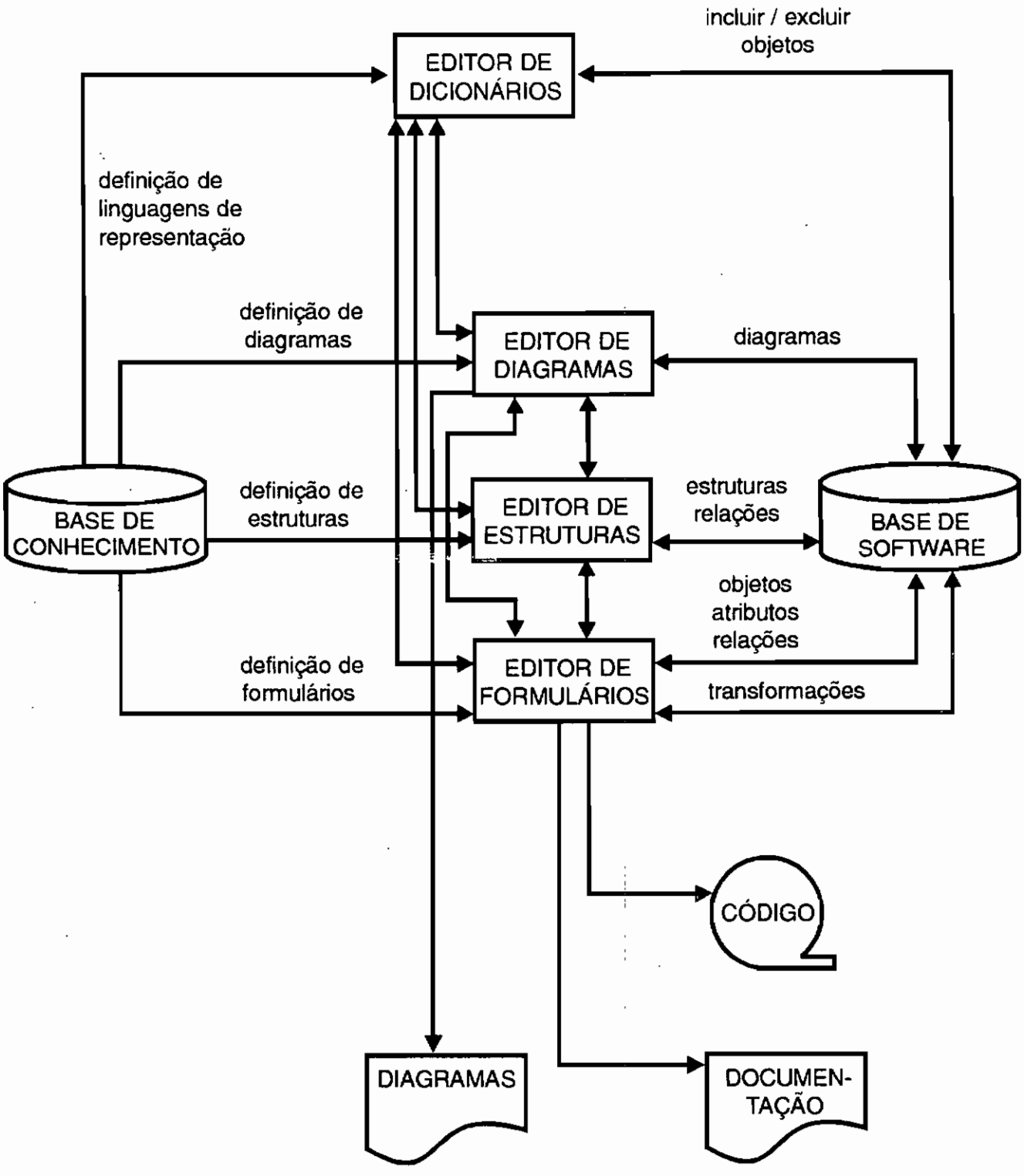


Figure 13: O Ambiente Talismã

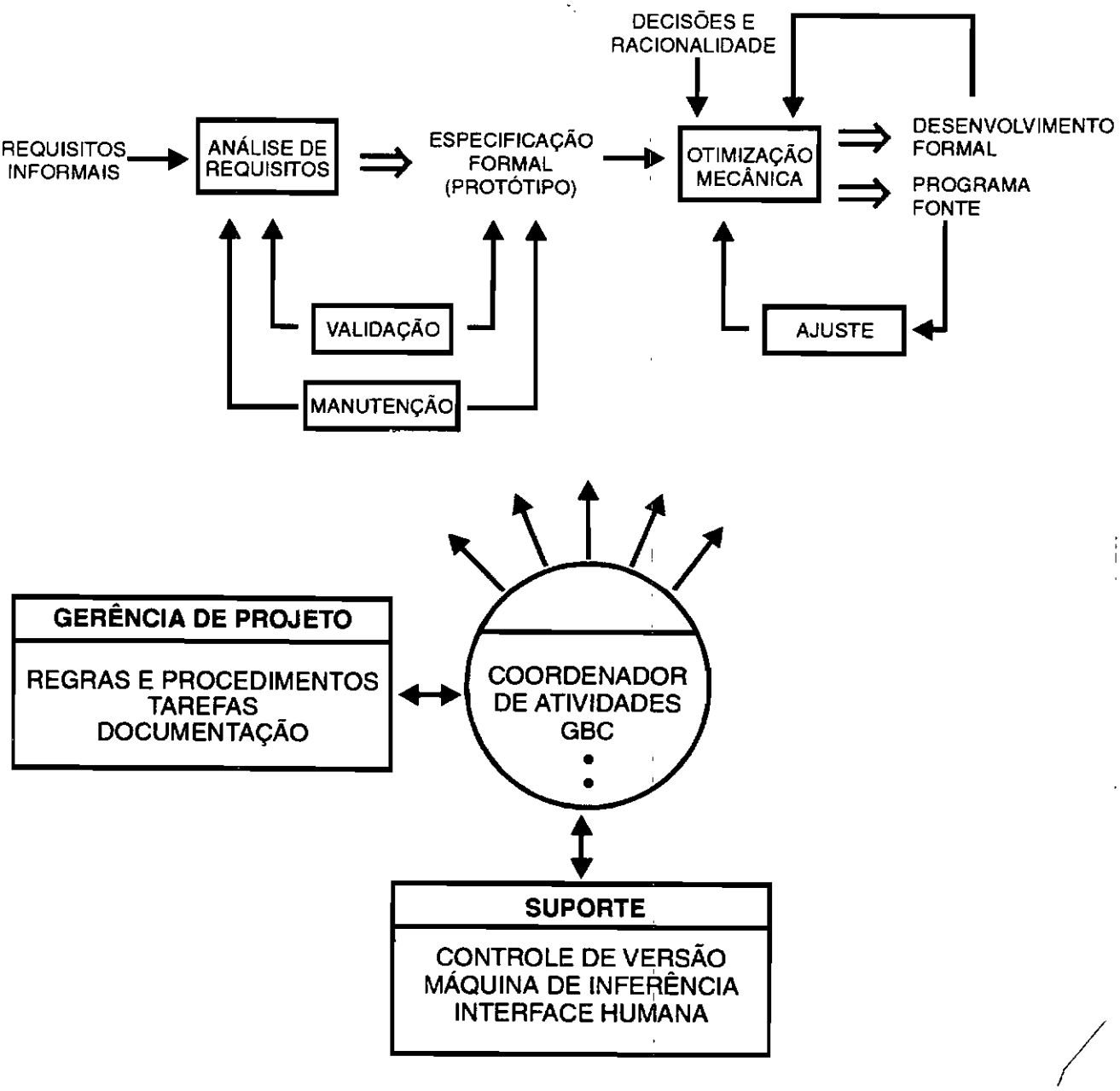


Figure 14: Modelo de Base de Conhecimento (Paradigma Revolucionário)

A engenharia de requisitos estabelece o processo de Definição de Requisitos como um processo no qual o que deve ser feito é elicitado, modelado e analisado. Este processo deve lidar com diferentes pontos de vista, e usar uma combinação de métodos, ferramentas e pessoal. O produto desse processo é um modelo, do qual um documento, chamado requisitor é produzido. Este processo acontece num contexto previamente definido a que chamamos de Universo de Informações.

Figure 15: Definição de Requisitos

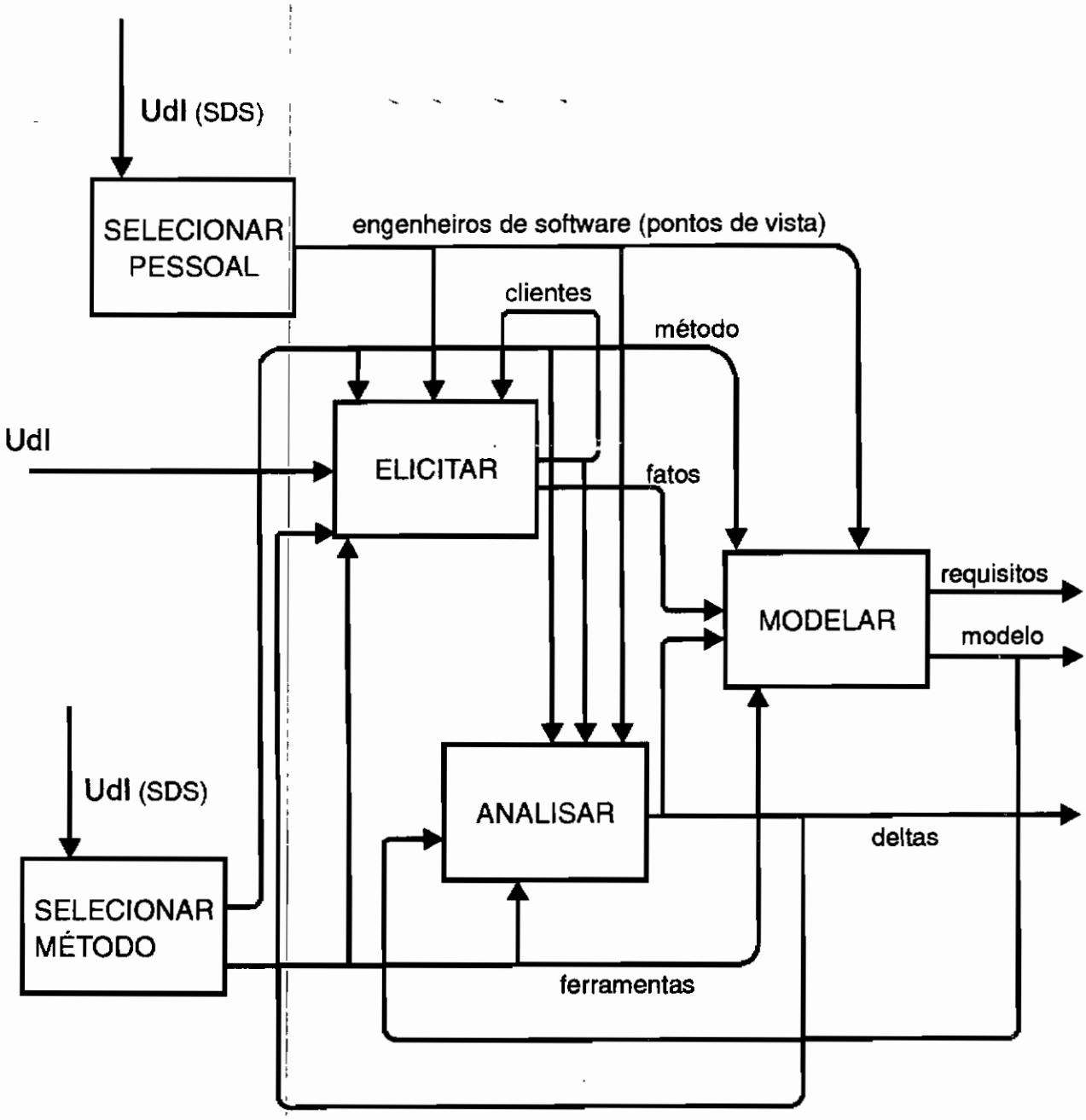


Figure 16: Modelo SADT para Engenharia de Requisitos

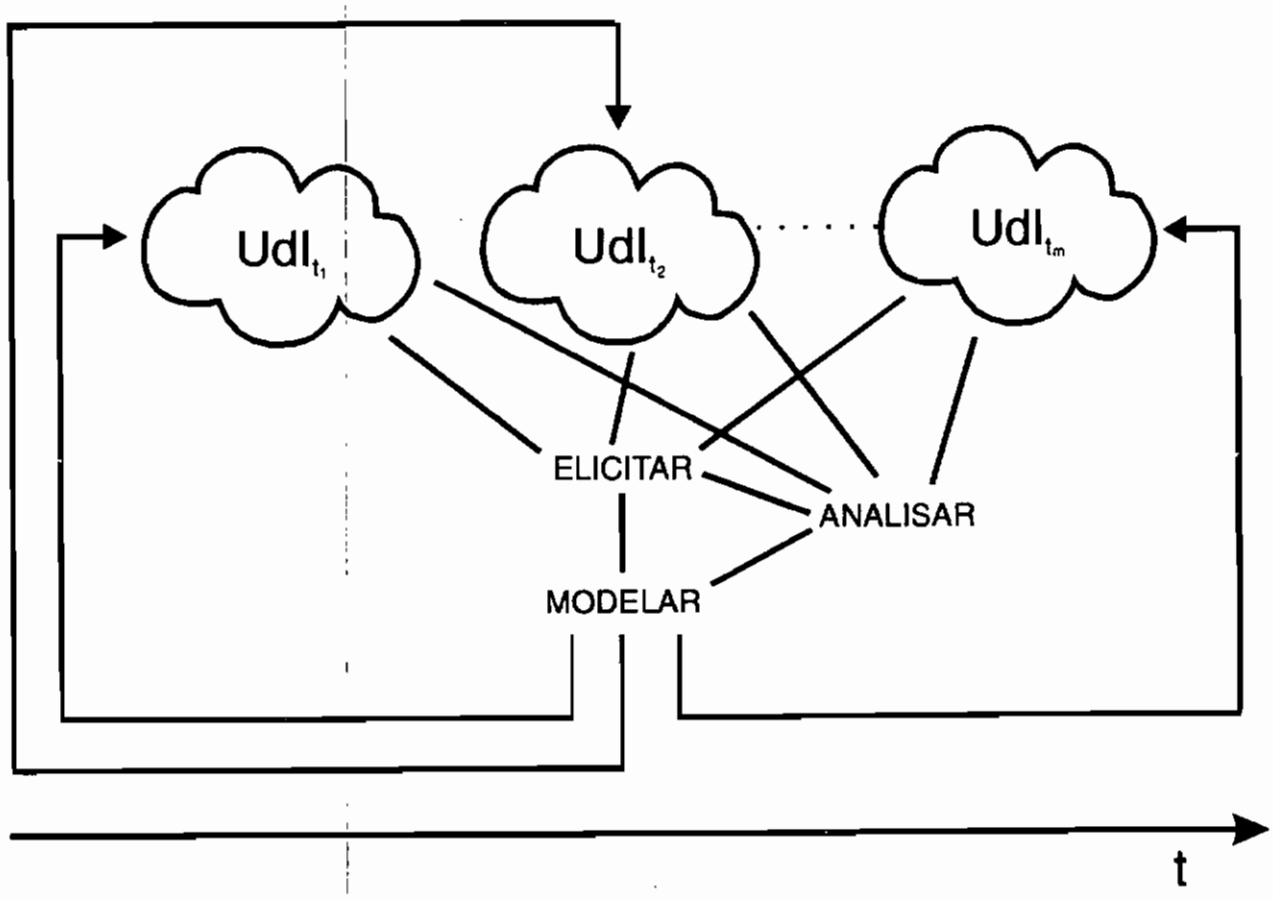


Figure 17: Evolução do UdI

References

[Balzer 83] Balzer, Green and Cheatman, *IEEE Computer*, 1983.

[Bowman 81] Bowman, B., Davis, G., Wetherbe, J. Modeling for MIS, *Datamation*, July 1981.

[Daniels 71] Daniels, A. and Yeates, D. *Basic Trainning in Systems Analysis*, National Computing Center, London, UK, 1971.

[Carvalho 88] Carvalho, L.C.S *Análise de Sistemas, O Outro Lado da Informática*, LTC – Rio de Janeiro, 1988.

[Curtis 88] Curtis, B., Krasner, H. and Iscoe, N. A Field Study of the Software Design Process for Large Systems, *Communications of the ACM*, Vol. 31, N. 11, Nov. 1988.

[Freeman 87] Freeman, P., Reusable Software Engineering: Concepts and Research Directions, in *Tutorial: Software Reusability*, Freeman (ed.), IEEE Computer Society Press, 1987.

[IEEE 93] Task Force on ECBS, Systems Engineering of Computer-Based Systems, *IEEE Computer*, Vol. 26, N. 11, Nov. 1993.

[Jackson 93] Jackson, M. *Systems Development*, Prentice-Hall International, E.C, New Jersey, 1983.

[Neighbors 84] Neighbors, J. The Draco Approach to Constructing Software from Reusable Components, *IEEE Trans. on Soft. Eng.*, vol. 10, n.9, Sep., 1984, pp. 564-573.

[Penedo 88] Penedo, M.H. and Riddle, W.E. Guest Editor' Introduction: Software Engineering Environment Architectures, *IEEE Trans. on Soft. Eng.*, vol. 14, n.6, Jun. 1988, pp. 689-695.

[Pressman 90] Pressman, R. *Software Engineering*, McGraw Hill, 1990.

[Rockart 79] Rockart, J.F., Chief Executives Define Their Own Data Needs, *Harvard Business Review*, vol. 49, Mar-Apr 1979, pp. 115-126.

[Ross 77] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. In *Tutorial on Design Techniques*, Freeman and Wasserman (eds.), IEEE Computer Society Press, Ca, 1980, pp. 107–125.

[Staa 92] Staa Informática, *Ambiente de Desenvolvimento Talismãn*, Manual do Usuário, Staa Informática, 1992.

Engenharia de Requisitos
Notas de Aula, Parte III

Julio Cesar Sampaio do Prado Leite
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

1994

1 Introdução

Escolhemos duas citações de eminentes cientistas para demonstrar a importância da tarefa de ELICITAR como um dos componentes de DEFINIR REQUISITOS. A primeira é a atribuída a von Neumann que diz:

> There is no sense in being precise about something when you do not even know what you are talking about.

A segunda citação é de Polya em seu livro *How to Solve it.* Polya estabelece três questões básicas para que se possa iniciar a tarefa de resolver um problema. Essas questões são:

- What is the unknow?
- Do you know a related problem?
- Could you restate the problem?

Conforme vimos na Parte II dessas Notas, a principal justificativa para o emprego dos métodos, técnicas e ferramentas da Engenharia de Requisitos é em função do custo de correção de erros, ou seja a medida que refinamos o nosso software cresce o custo das mudanças efetuadas nesse software. Além disso, as citações acima salientam o óbvio, muitas vezes esquecido, isto é, para construir o software temos que saber exatamente o que ele deve fazer. Dentro da Engenharia de Requisitos cabe à Elicitação a tarefa de identificar os fatos que compõem os requisitos do sistema, de forma a prover o mais correto e mais completo entendimento do que é demandado daquele software. Mas aqui então, cabe a pergunta, o que é elicitar?

Elicitar:

> [Var. eliciar + clarear + extrair] V.t. d. **1.** descobrir, tornar explícito, obter o máximo de informações para o conhecimento do objeto em questão.

Através do estudo dos problemas relacionados à elicitação, identificamos três partes que compõem essa tarefa (Figura 1). São elas:

- Identificação de Fontes de Informação.
- Coleta de Fatos.
- Comunicação.

Portanto, apesar de estarem intimamente ligadas entre si e com as tarefas de MODELAR e ANALISAR (vide Figura 16 da Parte II), essa subdivisão permitirá o entendimento e estudo mais detalhado de uma série de técnicas ligadas a tarefa de ELICITAR. Vamos verificar que muitas dessas técnicas não são propriamente técnicas oriundas da ciência da computação, pelo contrário, provêem principalmente das ciências sociais como a sociologia e a psicologia. Apesar disso, nós como engenheiros de software, temos que entende-las e contextualiza-las para o nosso uso, visto sua indispensabilidade.

Nosso objetivo nessa Parte das Notas é detalhar as subtarefas da elicitação. Na Seção 2 descreveremos como chegar às fontes de informação, na Seção 3 apontaremos uma série de técnicas utilizadas para coletar os fatos do Universo de Informações e que são a base para descrever os requisitos do software. A Seção 4 descreve algumas das técnicas que devem ser utilizadas na comunicação com os atores do UdI. A Seção 5 fala das interfaces entre essas subtarefas e como elas se relacionam com as outras tarefas de definir requisitos. Finalizamos com um resumo do que foi apresentado.

2 Identificação de Fontes de Informação

Conforme vimos na Parte II, é extremamente importante que possamos definir o contexto onde a Engenharia de Requisitos vai ocorrer. O passo inicial conforme visto é o de sabermos qual é o nosso Universo de Informações. É desse universo que extrairemos as informações necessárias na tarefa de elicitação. A qualidade da delineação desse universo é função do macrosistema ao qual o software vai servir. Dependendo do caso, teremos uma imagem mais ou menos nítida de seus contornos.

O UdI contém todas as fontes de Informação que vamos utilizar, mesmo que durante a fase de elicitação tenhamos que re-delinear esse UdI (Figura 2). Portanto é fundamental que possamos nos orientar no UdI para que possamos estabelecer uma estratégia de investigação dessas fontes de informação.

2.1 Técnicas para Identificação de Fontes de Informação

Poucos trabalhos na literatura têm se dedicado exclusivamente a este tópico, sendo que Burstin [Burstin 84] é um dos poucos que investigou essa tarefa. Burstin criou uma série de heurísticas para identificar usuários e compor uma árvore desses usuários, que ele chamou de *abstract user tree*. Essa árvore além de mapear os usuários envolvidos no processo, relaciona a cada usuário um conjunto de requisitos, segundo aquele usuário[1]. Basicamente o que Burstin consegue com seu método é mapear os usuários de uma forma abstrata segundo uma hierarquia, na qual os usuários que têm necessidades mais concretas aparecem nas folhas da árvore (Figura 3).

É importante observar que atores não são a única fonte de informação disponível num UdI. Outras importantes fontes de informação podem ser:

[1] É importante notar que usuário aqui tem o mesmo significado que o termo atores que empregamos na definição do UdI.

- documentos do UdI, que podem incluir:
  - documentação do macrosistema,
  - políticas da organização (exemplo: Plano Diretor de Informática)
  - manuais de equipamentos de hardware ou software,
  - memorandos, atas de reunião, contratos com fornecedores e outros documentos.
- livros sobre os temas relacionados,
- outros sistemas já existentes na empresa,
- outros sistemas já existentes no mercado,

2.2 Heurísticas Gerais

Apesar da enorme disponibilização das várias fontes de informação, é importante priorizar essas fontes, para que não se perca tempo desnecessariamente. É impraticável o acesso a todas as fontes de informação disponíveis no UdI. Para tal priorização, não existe uma regra fixa e cada caso é uma realidade distinta. No entanto, podemos listar algumas heurísticas que consideramos importantes.

- Procurar identificar os donos do sistema, pessoas ou setores de uma organização que são os principais *clientes* do software a ser construído, ou seja aqueles que cobram do SDS a entrega do produto software.
- Procurar identificar os atores que serão impactados em suas rotinas de trabalho com o funcionamento do software.
- Procurar estabelecer a rede de comunicação existente entre os componentes do macrosistema (seja um sistema de engenharia ou um sistema de informação).
- Procurar identificar com a ajuda dos atores mencionados acima, outros que possam ter algo a dizer sobre o macrosistema para o qual o software será desenvolvido.
- Procurar identificar se há grupos de interesse relacionados ao sistema em questão.
- Procurar identificar a existência de soluções, pacotes, já disponíveis no mercado.
- Procurar ler os documentos mais referenciados pelos atores identificados.
- Procurar descobrir quais os livros ou outros sistemas relacionados com o tema, ou mencionados pelos atores.
- Descobrir, através de perguntas diretas aos atores identificados, quais outras fontes de informação devem ser consultadas.

Uma vez identificadas e priorizadas, as fontes de informação precisam ser investigadas e isso é tarefa da coleta de fatos.

3 Coleta de Fatos

A Figura 4 extraída do livro do Sá Carvalho (vide Parte II) consegue expressar de uma maneira bastante suscinta algumas das estratégias principais para a Coleta de Fatos. Sem o intuito de cobrir todas as estratégias, fornecemos abaixo uma lista das que podem ser usadas na tarefa de coletar fatos. Algumas dessas estratégias são oriundas de ciências sociais, outras das ciências cognitivas (aqui com forte relacionamento com a Inteligência Artificial) e outras da própria Engenharia de Software. Em especial, duas referências bibliográficas são indicadas para uma revisão de várias dessas técnicas, [Goguem 93] para aspectos de ciência social e [Neale 89] para inteligência artificial.

3.1 Técnicas de Coleta de Fatos

Leitura de Documentos Conforme vimos anteriormente é importante que a elicitação possa ter acesso ao conhecimento escrito. A leitura de documentos é usualmente feita pelos engenheiros de requisitos. Recentes propostas de pesquisa [Maarek 89] têm mostrado que ferramentas podem ajudar na extração de abstrações de textos em linguagem natural.

Da leitura de documentos, além das principais abstrações pode-se também entrar em contato com o vocabulário da aplicação. O conhecimento do vocabulário é de extrema importância e [Leite 90] aponta uma maneira de como tratar desse aspecto através da elicitação do Léxico Ampliado da Linguagem (LAL).

As principais vantagens são: a facilidade de acesso às fontes de informação e volume de informações que podem ser extraídas dessas fontes. As principais desvantagens são: a dispersão das informações e o volume de trabalho requerido para a identificação dos fatos. No entanto a utilização de ferramentas, como citadas acima podem minorar essas desvantagens.

Observação Técnica bastante utilizada, na qual o engenheiro de requisitos procura ter uma posição passiva no UdI observando o ambiente onde o software irá atuar. A observação é uma das estratégias para que o engenheiro de requisitos inicie-se no área de aplicação do software. Normalmente a estratégia de observação permitirá ao engenheiro de software fazer anotações gerais sobre os objetos observados, e sobre a linguagem utilizada (vocabulário).

As principais vantagens da observação são: o baixo custo e a pouca complexidade da tarefa. Por outro lado, as principais desvantagens são: a dependência do ator desempenhando o papel de observador e a superficialidade decorrente da pouca exposição ao universo que está sendo observado.

Entrevistas Entrevistas são o meio mais usual com o qual o engenheiro de requisitos coleta fatos. Há diversos tipos de entrevistas: entrevistas estruturadas, entrevistas informais, entrevistas tutoriais. A entrevista estruturada normalmente requer que o engenheiro de requisitos já disponha de algum conhecimento sobre o problema, de forma a que possa preparar as perguntas. A entrevista tutorial é aquela em que o cliente está no comando, ou seja é mais uma aula sobre o UdI. A entrevista informal ou não estruturada fornece uma maior flexibilidade ao entrevistador e normalmente é utilizada na fase mais exploratória.

As principais vantagens são: a possibilidade de contato direto com os atores que detêem conhecimento sobre os objetivos do software e a possibilidade da validação imediata através de processos de comunicação que enfatizam a confirmação. As principais desvantagens são: o problema do conhecimento tácito e as diferenças de cultura entre entrevistado e entrevistador.

O conhecimento tácito é aquele conhecimento que é trivial para o entrevistado e não o é para o entrevistador, mas que por ser tão trivial, nunca é lembrado como importante pelo entrevistado e portanto não é transmitido ao entrevistador.

Questionários Questionários são utilizados quando se tem um bom conhecimento sobre o problema (aplicação) e se quer varrer um número grande de clientes. Os questionários são importantes para que se possa ter uma idéia mais definida de como certos aspectos do UdI/software são percebidos por um grande número de pessoas. Além disso, se bem planejados, os questionários possibilitam análises estatísticas.

As principais vantagens do questionário são: a padronização das perguntas e a possibilidade de tratamento estatístico das respostas. As principais desvantagens são: a limitação do universo de respostas e a pouca iteração, visto a impessoalidade deste técnica.

Análise de Protocolos Esta estratégia consiste em analisar o trabalho de determinada pessoa, através de verbalizações dessa pessoa. Ou seja procura-se entender mais sobre o trabalho de alguém perguntando-se a essa pessoa como o trabalho é feito. Usualmente a verbalização ocorre no ato do trabalho, isto é, enquanto faz sua tarefa a pessoa fala em voz alta o que esta fazendo. Outra maneira de usar a verbalização é pedir para a pessoa recordar como faz o trabalho. Ainda outra maneira é colocar diante da pessoa uma situação possível e pedir que ela fale como aquela situação seria resolvida. O objetivo dessa estratégia é fundamentalmente procurar estabelecer a racionalidade utilizada na execução de tarefas. É uma estratégia bastante usada em Aquisição de Conhecimento (Inteligência Artificial).

As principais vantagens são: a possibilidade de elicitar fatos não facilmente observáveis e permitir um melhor entidemento dos fatos, vistos que estes são explicados e justificados. As principais desvantagens são: a técnica centra-se principalmente na performace do entrevistado e sofre do problema de que o que se diz é diferente do que se faz.

Participação Ativa dos Atores UdI (Clientes) Esta estratégia procura incorporar ao grupo de engenheiros de software os atores que demandam o software. Esses atores precisam aprender as linguagens de modelagem utilizadas, para poderem, principalmente, ler as descrições produzidas pelos engenheiros de software com o objetivo de criticá-las. Também é possível uma maior integração desses atores, que passariam também a modelar com os engenheiros de software o problema.

As principais vantagens da participação são: induz o envolvimento dos clientes e usuários no processo de identificação dos fatos e da elicitação do conhecimento e facilita o processo de validação. As principais desvantagens são: o treinamento dos clientes e usuários em técnicas de informática e uma falsa impressão de que, pelo participação pura e simples de representantes dos clientes e usuários, o processo foi executado de maneira eficaz.

Enfoque Antropológico Nessa estratégia usa-se uma técnica inversa da descrita acima. Aqui os engenheiros de software devem procurar integrarem-se ao UdI de forma a terem um conhecimento o a mais amplo possível do problema. Esta integração deve se dar de tal forma que o engenheiro de software passe a ser visto como um cliente, isto é podendo ter as mesmas perspectivas que teria um cliente. É uma estratégia lenta, mas que vem recebendo muita atenção, em especial com o emprego das técnicas de etenografia (*Ethnography*) [Sommerville 93].

A principal vantagem do enfoque antropológico é a possibilidade de uma visão de dentro para fora mais completa e perfeitamente ajustada ao contexto. No que diz respeito as desvantagens, a principal refere-se ao tempo gasto e também a pouca sistematização do processo da etenografia.

Reuniões Reuniões podem ser classificadas de várias maneiras dependendo como elas são organizadas. Elas podem ser uma extensão do conceito de entrevista ou podem ser vistas como uma maneira de se ter a participação ativa dos atores da UdI. Técnicas de *brainstorm* podem ser utilizadas para a fase de exploração dos requisitos. Métodos como JAD (Joint Application Design) [Andrews 91], têm sido usados de maneira eficaz para acelerar o processo de definir requisitos. JAD é um método que estrutura uma reunião de maneira a que se possa conseguir um clima de cooperação.

As principais vantagens de reuniões são: a possibilidade de se dispor de múltiplas opiniões e da criação coletiva. Por outro lado, as principais desvantagens são: a possibilidade de dispersão e o custo.

Reutilização Um ponto já abordado na Parte II é o contexto do Reuso, ou seja seria extremamente produtivo se ao procurar definir requisitos para um determinado software, pudessemos reutilizar fatos já coletados. A área de reutilização considera que esse reuso de alta abstração é possível quando se tem disponível um `domínio` [Arango 89]. Um domínio seria o encapsulamento do conhecimento de toda uma área de aplicação. Portanto, ao se definir um novo software naquela área de aplicação os conceitos já estariam disponíveis para serem reutilizados, sem que fosse necessário gastar um tempo para entendê-los e modela-los. Na Figura 10 da Parte II mostramos o cenário de uso do Paradigma Draco, um paradigma que implementa a reutilização de domínos.

As principais vantagens da reutilização são: a produtividade e a qualidade, visto que os componentes a serem reutilizados já foram validados anteriormente. A principal desvantagem é a dificuldade de se prover reutilização, sem modificação, ao nível de abstração da definição de requisitos.

Recuperação do Desenho de Software Uma outra estratégia seria o estudo de software disponíveis na organização. Como muitas vezes esses software não têm uma documentação fiel a sua implementação é necessário que se faça um trabalho de arqueologia, isto é procure-se recuperar a arquitetura e os requisitos desse software. Ultimamente, vários trabalhos de re-engenharia tem sido propostos [Leite 91] como forma, não só de recuperar o que exatamente fazem os sistemas existentes, como também otimizar a tarefa de mantê-los. A Figura 5 apresenta uma estratégia geral para a re-engenharia de software.

As principais vantagens da re-engenharia são: a disponibilidade das informações e a possibilidade de reuso, isto é uma vez recuperadas as informações é possível o reuso. Por outro lado as desvantagens são principalmente referentes a dificuldade do processo de reuso e a dispersão das informações.

3.2 Heurísticas Gerais

Cada uma das técnicas apresentadas acima tem uma série de heuristícas associadas e muitas delas já apresentam um ferramental de suporte às suas partes sistematizadas. No entanto,

cabe aqui listar algumas heurísticas de caráter geral que acreditamos serem muito importantes para a coleta de fatos.

- Perguntar, Perguntar, Perguntar.
- Sempre Perguntar: O que? Porque? Como? Por quem?
- Pergunte o Óbvio.
- Procure esclarecer o que é óbvio no UdI.
- Organize as respostas: durante *versus* depois.
- Volte a perguntar.
- Organize as perguntas, as respostas, e o método usado.
- Viva no UdI por um tempo.
- Tenha uma visão antropológica.
- Observe.
- Estude, estude, estude.
- Seja humilde. Procure aprender.
- Aprenda, aprenda, aprenda.

4 Comunicação

Para que a elicitação tenha sucesso é fundamental que os engenheiros de requisitos possam se comunicar eficazmente com os clientes. No entanto é notório que existem incríveis barreiras entre clientes e engenheiros de requisitos. Utilizamos outra figura do livro do Sá Carvalho para caracterizarmos esse problema (Figura 6). As diferenças de conhecimento entre clientes e engenheiros de software são reflexos de cultura diferente e que não são fáceis de resolver. Um aspecto importante é o que é normalmente conhecido como conhecimento tácito, isto é conhecimento que é trivial para uns, e que dificilmente são explicitados, e não trivial para os outros.

4.1 Técnicas para Comunicação

Vários aspectos são importantes na tentativa de facilitar a comunicação entre clientes e engenheiros de requisitos. Abaixo listamos alguns desses aspectos.

- **Apresentação:** De que maneira a informação é apresentada. Pesquisas apontam que diferentes formas de apresentação da informação dificultam ou auxiliam sua compreensão. Este aspecto é valido nas duas direções, isto é clientes – engenheiros e engenheiros – clientes.

- **Entendimento:** O estabelecimento de um contexto comum facilita o entendimento. Por exemplo, na sequência 5, 10, 2, 9, 8, 4, 6, 7, 3, 1 a ordem é alfabética, no entanto é praticamente impossível que essa questão seja respondida corretamente se apresentada apenas a sequência númerica. O estabelecimento de contexto e objetivo é fundamental para inciar-se um entendimento mútuo entre clientes e engenheiros de software.

- **Linguagens:** As linguagens são reflexo de culturas, portanto culturas diferentes têm linguagens diferentes. Cabe ao engenheiro de requisitos procurar entender a linguagem de seus clientes antes de entender suas necessidades. O conhecimento da linguagem do cliente é importante como meio de facilitar a comunicação.

- **Nível de Abstração:** Mesmo em se tratando de uma única cultura a comunicação pode ser extremamente ruidosa se os indivíduos estiverem dialogando em diferentes níveis de abstração. Este problema se agrava quando os indivíduos têm cultura distinta.

- **Retroalimentação:** Uma das maneira eficazes de garantir a passagem da informação do emissor para o receptor de maneira correta é obrigar ao receptor recolocar a comunicação até que o emissor responda positivamente a recolocação.

Um artigo que aborda diretamente esse problema, sob o ponto de vista cognitivo, e sua relação com o processo de definição de requisitos é um artigo de Bostrom [Bostrom 89]. Neste artigo ele apresenta uma técnica de programação neuro-linguística, Modelo de Precisão, que objetiva reduzir os ruídos de comunicação entre clientes e engenheiros de software numa reunião de definição de requisitos.

O Modelo de Precisão identifica três grupos de problemas em comunicação e apresenta algumas heurística para evitá-los. Abaixo listamos os grupos e as heuríticas utilizadas.

- Padrões de referência, aspecto já explicado acima sob o nome de entendimento. Abaixo listamos os tipos de padrões.

  - Resultados: Aqui se procurar confirmar entre os participantes os objetivos da reunião. Exemplo: Qual a intenção dessa reunião? O que queremos atingir com esta reunião?
  - Retrocesso: É o problema que listamos acima como o de Retroalimentação, isto é a verificação de uma afirmação anterior. Exemplo: Vamos resumir o que discutimos até agora? João, você quer dizer ... e ....?
  - Se: Este tipo de padrão procura estimular a criatividade, criando situações fictícias. Exemplo: Agindo como o usuário como este responderia a tal questão? Imagine que o usuário principal entra em sua sala e pede mais um relatório, o que você diria?

- Procedimentos são conjuntos de afirmações verbais ou questões que permitem ao comunicador guiar a direção da entrevista e assegurar que a discussão permaneça dentro do padrão de resultados estabelecido. Abaixo listamos os procedimentos.

  - Evidência: Exemplo, Como sabe se o resultado foi obtido? Que evidência é necessária para se saber se o resultado foi obtido?
  - Relevância: Exemplo, Obrigado, a pergunta parece boa, mas não vejo uma ligação direta com o caso.

- Ponteiros: procuram clarificar expressões vagas, ou mal definidas. Exemplo, Você poderia ser mais claro? A que ponto do problema você esta se referindo?

O uso de técnicas de comunicação, conforme mostrado acima, aumenta a possibilidade de uma comunicação mais clara entre as partes. Apesar da enfâse dada na comunicação verbal é também importante termos em mente o melhor uso possível da linguagem escrita, principalemente quando usamos a linguagem natural, isto é o Português. Dentre as técnicas disponívies para uma melhor comunicação através da linguagem escrita, recomendamos o uso de um manual de estilo utilizado por grandes editores, como por exemplo o uso do manual da Abril [Abril 90]. O uso da linguagem jornalística direta e clara é a melhor técnica para a comunicação escrita, quando o meio for o da linguagem natural.

4.2 Heurísticas Gerais

Independente do uso da comunicação verbal ou escrita, é importante salientar algumas heurísticas gerais. Estas heurísticas procuram ressaltar pontos importantes na boa comunicação. Listamos abaixo as heurísticas que consideramos importantes.

- Uma **boa** figura vale 1000 palavras.
- Duplo trafégo na comunicação (retroalimentação).
- Evitar ruídos.
- Evite metáforas com sua área de conhecimento (informática).
- Procure indentificar o ponto de vista de seu interlocutor.
- Aprenda com humildade.

5 Aspectos Gerais

É importante observar que, apesar da grande similiaridade entre elicitação de requisitos e aquisição de conhecimento, a aquisição de conhecimentos tem um preocupação especial com a performace do especialista na execução de sua especialidade. A performace dos atores do UdI não é importante, de maneira geral, na elicitação de requisitos.

Conforme haviamos dito, a divisão da tarefa de elicitar tem como objetivo tão somente facilitar nosso entendimento sobre o problema. No entanto, quando buscamos elicitar os requisitos de um sistema, precisamos usar uma combinação de técnicas onde a clara divisão apresentada fica nebulosa. Muitas vezes temos que utilizar a técnica de observação para poder determinar a fonte de informação e ao mesmo tempo temos que nos comunicar com um candidato a entrevistas e portanto aplicar uma técnica de comunicação antes mesmo da entrevista. Também temos que registrar que fontes de informação serão exploradas antes mesmo de iniciar o processo de coleta de fatos.

Cabe ao profissional de Engenharia de Requisitos selecionar as técnicas a serem utilizadas e estabelecer de que maneira elas serão integradas. Hoje se sabe que o uso de múltiplos modelos nos ajudam, não só no entendimento do problema, como também em relação a representação de situações mais complexas. Muitas dessas técnicas, aqui apresentadas, têm possibilidade de

serem integradas. Como por exemplo as técnicas do modelo de precisão à entrevistas, ou a técnica de identificação de documentos com a técnica de questionários e assim por diante.

No entanto, é importante utilizar uma modelagem de apoio de forma a que os fatos elicitados fiquem corretamente representados para futuro tratamento. Primeiramente esses fatos ficam representados por linguagem natural, mas são sistematizados segundo as técnicas de modelagem utilizadas.

A escolha das técnicas a serem utilizadas bem como seu esquema de integração dependerá do Universo de Informação de onde se quer elicitar os requisitos e da equipe participante dessa escolha. O ponto importante é ter conhecimento sobre essas técnicas e identificar a situação onde uma técnica é superior a outra, levando-se em conta o binômio custo–qualidade.

Vamos ver na Parte IV algumas das representações que podem ser utilizadas para facilitar a tarefa de coletar os fatos. Essas representações têm ou deveriam ter um conjunto de técnicas associadas que determinam a maneira como será feita a coleta.

Finalizamos listando abaixo algumas heurísticas gerais à coleta de fatos, onde a enfâse esta nas justificativas (por quê?), parte essencial para o correto registro da racionalidade utilizada no processo de DEFINIR REQUISITOS.

- Perguntar o que? Como? Por quê?
- Com quem falar? Por quê?
- Estudar o que? Por quê? Onde começar?
- Como apresentar? Para quem? Como?
- Rever.

6 Resumo

Nesta Parte revemos a importância da elicitação e detalhamos técnicas e heurísticas que ajudam o engenheiro de software no entendimento do sistema objeto. Vimos a importância de corretamente selecionarmos as fontes de informação e como identifica-las. Vimos também a pletora de técnicas ao nosso dispor para elicitar os fatos do mundo real. Ressaltamos também os aspectos da comunicação verbal e escrita que impactam sobremaneira o processo de elicitação. Chamamos a atenção para o aspecto de seleção e integração de diversas técnicas e métodos. Também focalizamos o registro da racionalidade e da integração dessa prática ao processo de elicitação.

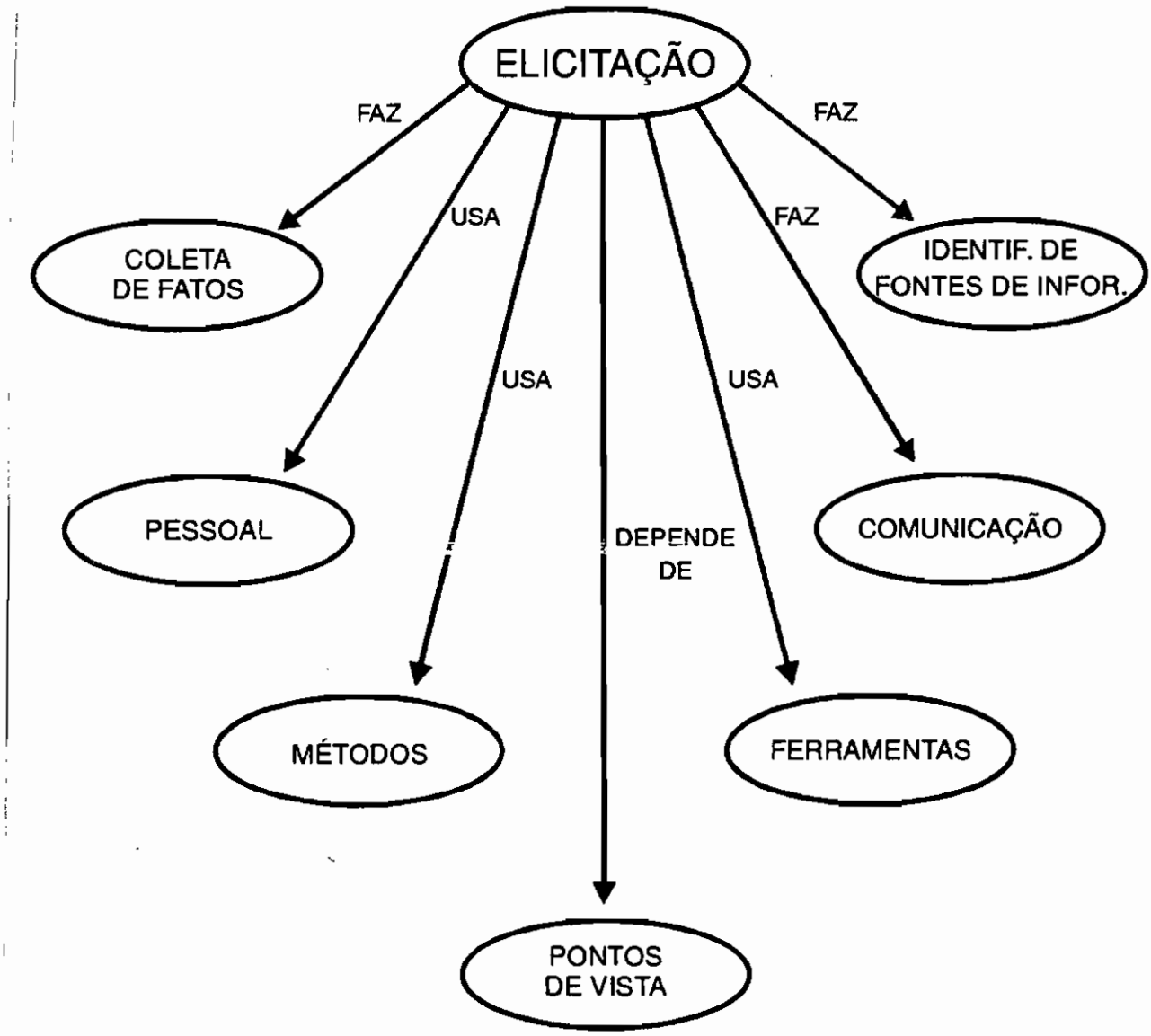


Figure 1: Elicitação

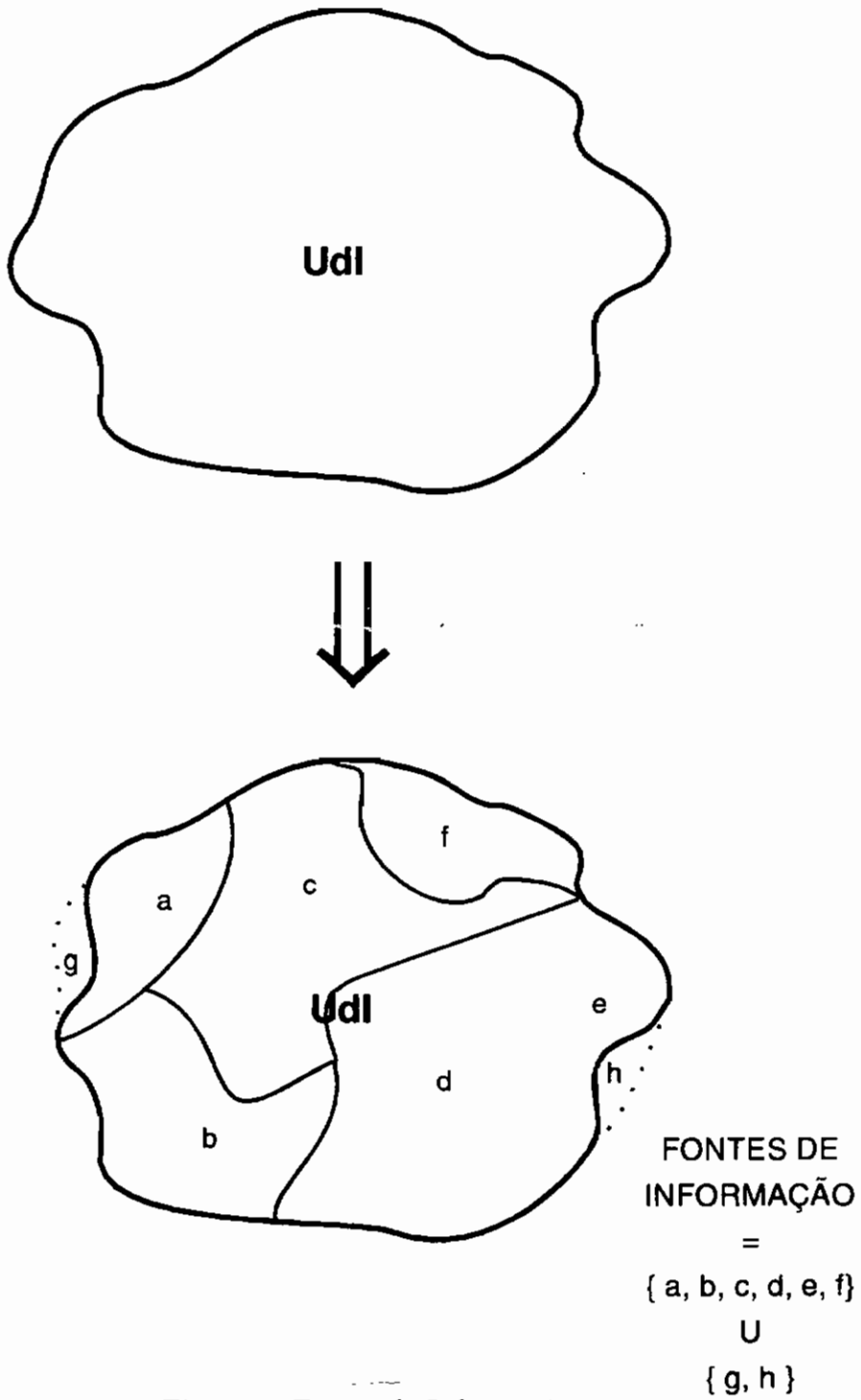


Figure 2: Fontes de Informação

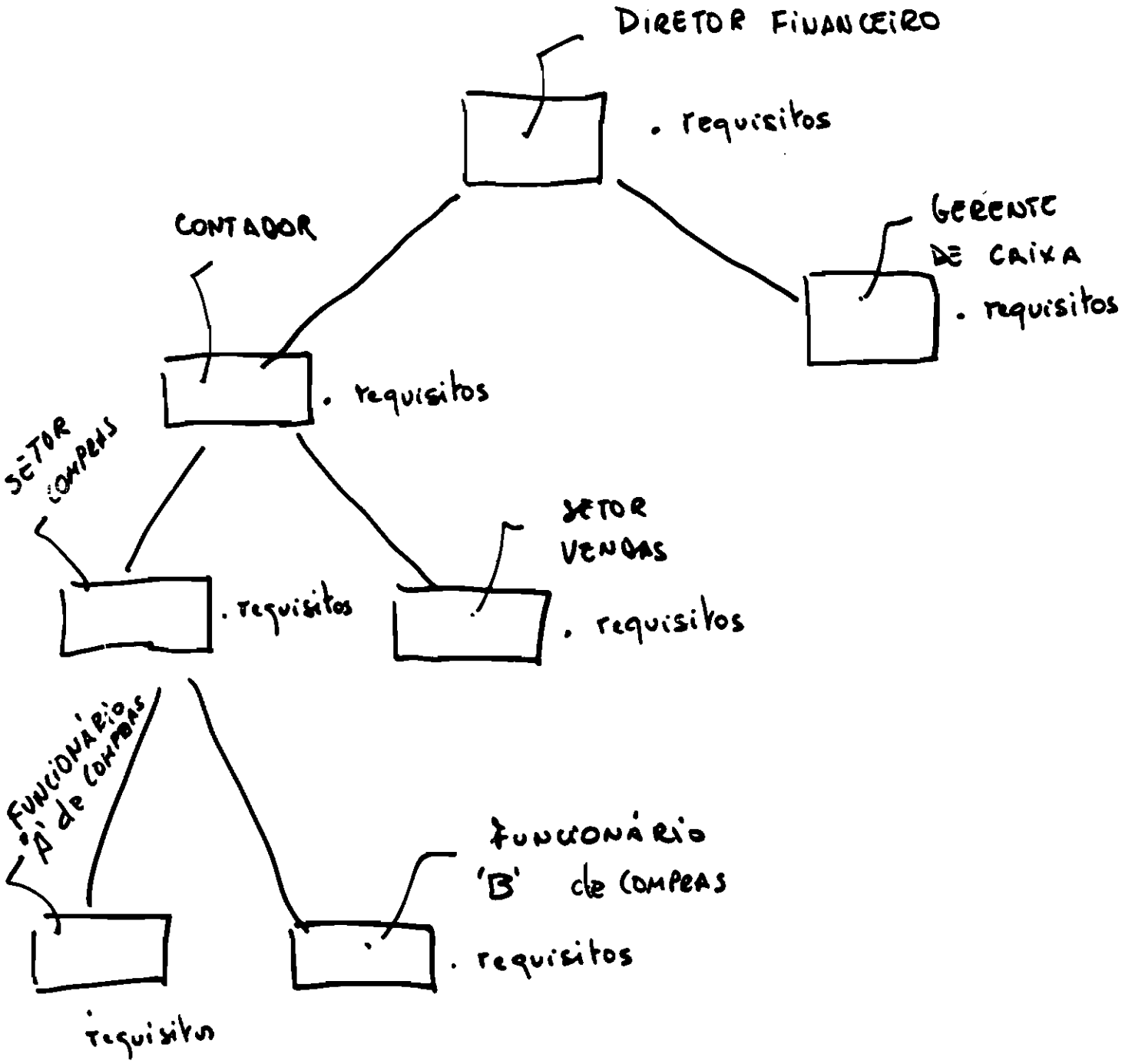


Figure 3: Abstract User Tree

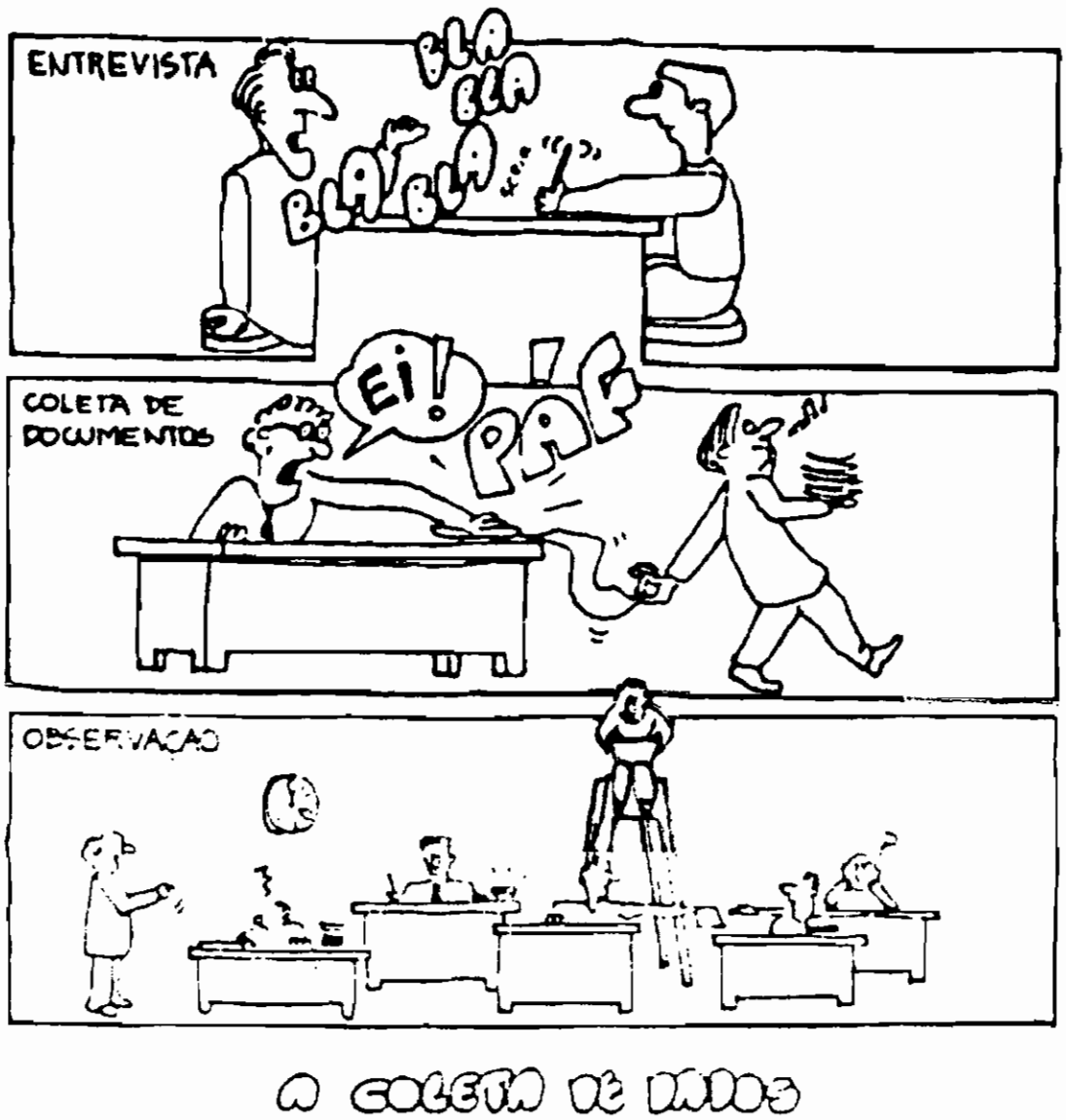


Figure 4: Coleta de Fatos

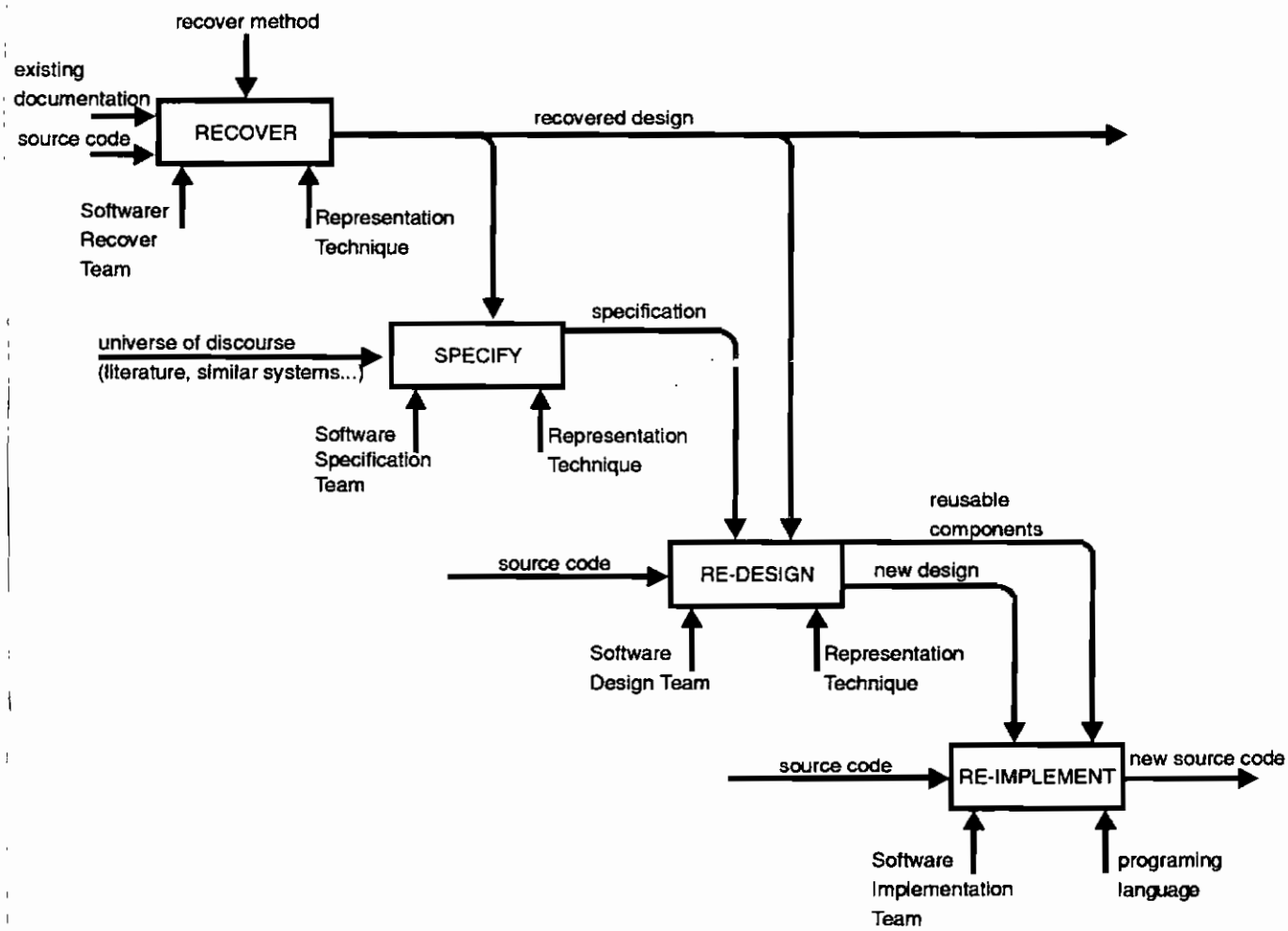
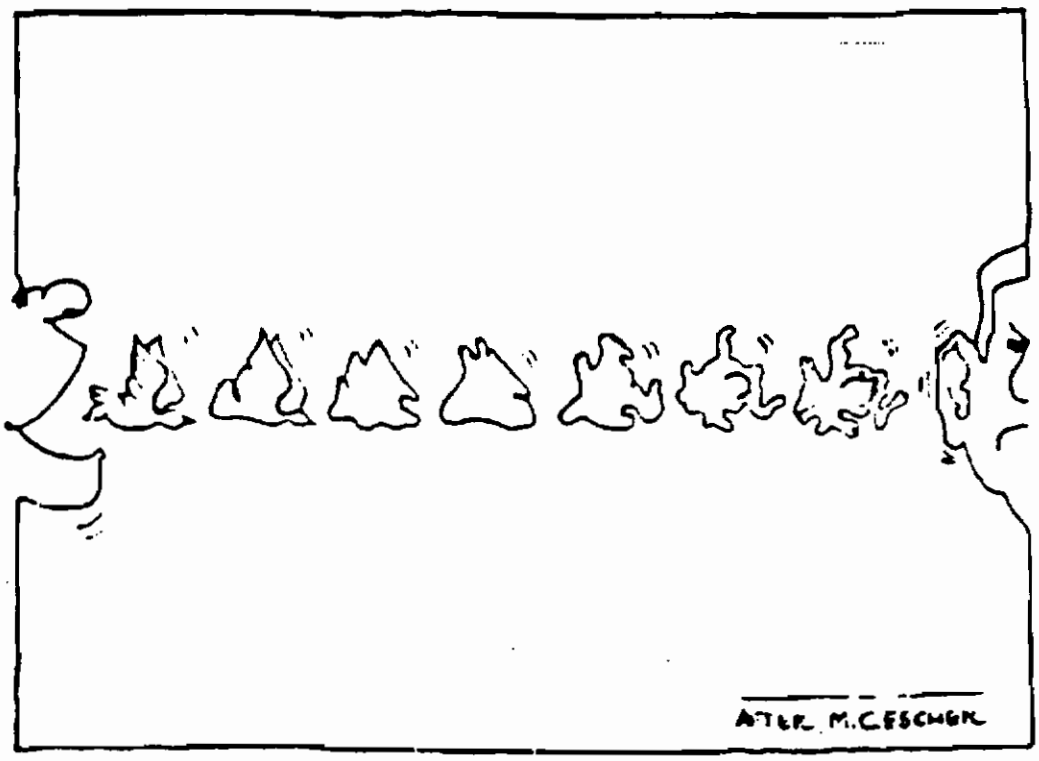


Figure 5: Re-Engenharia



ANALISTA X USUÁRIO

Figure 6: Comunicação

References

[Abril 90] Editora Abril, *Manual de Estilo Editora Abril*, Editora Nova Fronteira, 1990.

[Andrews 91] Andrews D. C. JAD: A crucial dimension for rapid applications development. *Journal of Systems Management*, Mar. 1991, pp. 23-31.

[Arango 89] Arango, G. Domain Analysis: From Art Form to Engineering Discipline. *In 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, 1989), IEEE Computer Society Press, pp. 152-159.

[Burstin 84] Burstin, M. *Requirements Analysis of Large Software Systems*, Ph.D. diss., Tel-Aviv Univ., Israel, 1984.

[Bostrom 89] Bostrom, R. P. Successful Application of Comunication Techniques to Improve the Systems Development Process, *Information & Management*, 16 (1989).

[Goguem 93] Goguem, J. and Linde, C. Techniques for Requirements Elicitation, *First International Symposium on Requirements Engineering*, IEEE Computer Society Press, 1993.

[Leite 90] Leite, J.C.S.P e Franco A.P.M., O Uso de Hipertexto na Elicitação de Linguagens da Aplicação. *Anais do IV Simpósio Brasileiro de Engenharia de Software* Sociedade Brasileira de Computação, Out. 1990.

[Leite 91] Leite, J.C.S.P e Souza, A., Re-Engenharia de Software, Um Novo Enfoque para um Velho Problema, *Anais do Congresso Nacional de Informática* SUCESU, São Paulo, 1991.

[Maarek 89] Maarek, Y. and Berry, D. M., The Use of Lexial Affinities in Requirements Extraction, *IWSSD-5*, IEEE Computer Society Press, 1989.

[Neale 89] eale, I.M., First Generation Expert Systems: a review of knowledge acquisition methodologies, *Journal of Knowledge Engineering Review*, 1989.

[Sommerville 93] Sommerville, I., et. all, Integrating Ethnography into the Requirements Engineering Process, *First IEEE International Symposium on Requirements Engineering*, IEEE Computer Society Press, 1993.

Engenharia de Requisitos
Notas de Aula, Parte IV

Julio Cesar Sampaio do Prado Leite
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

1994

1 Introdução

Conforme vimos na Parte II dessas Notas, a Engenharia de Software tem centrado sua atenção quase que exclusivamente em modelagem. Os progressos são muitos e alguns deles já se refletem na prática. No entanto, para a Engenharia de Requisitos a modelagem precisa apresentar uma série de características que não necessariamente estão presentes nas propostas que procuram servir fundamentalmente o processo de desenho do software. Dessa maneira, ao lidarmos com modelagem no contexto da Engenharia de Requisitos, temos que nos valer de algumas das propostas da Inteligência Artificial.

Da mesma forma que fizemos com a elicitação dividiremos a tarefa de MODELAR em partes menores. Essas partes servem para que possamos entender e aquilatar as características das técnicas de modelagem. Novamente alertamos que muitas vezes é difícil separar essas partes, mas acreditamos que conhecendo-as, melhoramos nossa visão do processo de modelagem. Essas partes são (Figura 1):

- Representação
- Organização
- Armazenamento

A ciência da computação tem feitos consideráveis avanços no que diz respeito a modelagem. Linguagens que podem gerar modelos de considerável nível de abstração já são utilizadas na prática, e é grande o progresso nos esquemas de formalização dessas linguagens. Diferentes propostas de linguagens existem e essas são normalmente conhecidas como linguagens de especificação. Os modelos produzidos por estas linguagens são extremamente úteis na tarefa de prover semântica para as informações coletadas.

Conforme vimos anteriormente, a tarefa de DEFINIR REQUISITOS produz modelos e uma forma amigável de registro dos requisitos. Essa descrição amigável dos requisitos normalmente é feita em linguagem natural. Os modelos portanto provêem a semântica dos requisitos e serão utilizados posteriormente no processo de produção de software para que se possa atingir a implementação.

Abaixo, falaremos de cada uma das partes de modelagem e procuramemos dar exemplos de linguagens ou técnicas que se aplicam em especial a cada uma dessas tarefas. Vale observar que quando analisamos uma linguagem, os aspectos de representar, organizar e armazenar aparecem misturados. Portanto, nos exemplos que aparecerão sob a cada uma dessas tarefas, procuraremos ressaltar o que há de fundamental em cada linguagem.

2 Representação

A representação é a pedra mestra da modelagem, é responsabilidade da representação ter uma semântica bem definida, de tal maneira que os modelos nelas expressos tenham uma âncora semântica. Várias são as propostas de linguagens formais, mas realmentes são poucas aquelas que podem ser ditas formais. Consideramos formais aquelas linguagens (representações) que são executadas sem ambiguidade por uma máquina.

Exemplos de elementos básicos de representação são:

- tipos
- relações
- operações

Uma técnica bastante utilizada para descrever relações é `pré-pós condições`. Uma técnica usada para descrever algumas operações é `máquinas de estado finito`. Uma técnica utilizada para descrever tipos é a técnica de `axiomas`.

Algumas representação são formais, outras mais sistematizadas. O exemplo clássico de uma linguagem bem definida é a representação lógica (Figura 2). A representação é formal, desde de que os predicados (tipos) estejam bem definidos.

Na Figura 3 temos um exemplo em que operações e relações são representadas utilizando-se uma combinação de pré-pós condição com máquinas de estado finito. O presente exemplo descreve parcialmente um processo de produção de chips baseado em litografia por feixe de elétrons [Sanches 94].

A Figura 4 apresenta exemplo de uso do Léxico Ampliado da Linguagem (LAL) uma representação fundamentada na idéia de que coisas observáveis no UdI têm sua semântica definida no próprio UdI. Nota-se, que esta representação é extremamente simples, visto que seu objetivo é o de ajudar no endentimento da linguagem utilizada em um determinado UdI.

Na Figura 5 vimos um exemplo de representação em que operações são representadas por um gráfico de estruturas e por um gráfico de redes. Essa representação gráfica é utilizada pelo método JSD [Masiero 92] para representar a ligação de `ações` com suas `entidades` e de `processos` como o mundo real (processos com o número 0 associado) e entre si. Na listagem das ações e das entidades, devemos apontar os atributos ligados a essas ações e entidades.

3 Organização

Os aspectos de organização dizem respeito a maneira como as linguagens organizam seus conceitos ou possibilitam que suas descrições sejam organizadas. Um exemplo é a linguagem StateCharts [Harel 88] (Figura 6) que procura organizar máquinas de estado em níveis de decomposição.

As Figuras 7 e 8 mostram um exemplo utilizando o método SADT. O SADT [Ross 77] apresenta uma série de aspectos organizacionais, sendo que o principal é a estrutura de `decomposição`. Através de refinamentos sucessivos, com os limites de 3 a 6 para número de caixas, passamos a descrever cada vez com maior detalhe um determinado sistema. Vale lembrar também que a linguagem preserva as entradas, saídas, controles e mecanismos quando do detalhamento por decomposição. No nosso exemplo só temos 1 nível (A0) que é um detalhamento do sistema (A-0). Cada caixa do nível 1 é decomposta gerando até 6 diagramas do nível 2.

Na Figura 9 vimos uma estrutura muito utilizada em Inteligência Artificial para representar relações. O ponto forte dessa estrutura é a possibilidade de organizar relações. Notem que nessa figura existem tanto relações de decomposição, como relações de `especialização`.

Na Figura 10 temos uma outra estrutura oriunda da Inteligência Artificial chamada de script. Essa estrutura apresenta uma organização mais sofisticada que encapsula tipos e operações.

Nas Figuras 11 e 12 apresentamos uma regra e uma arquitetura típica de um sistema de produção. Esta arquitetura é muito utilizada para a implementação de sistemas especialistas e pode ser muito útil a Engenharia de Requisitos porque permite com que conhecimento impreciso e heurístico possa ser codificado.

A Figura 13 apresenta o esquema básico de organização da representação LAL descrita acima.

4 Armazenamento

Para que a modelagem seja efetiva é preciso que seja armazenada a contento, e principalmente que possa ser recuperada quando for necessário. Esquemas de `classificação`, `indexação` e `apresentação` são indispensáveis para que o armazenamento seja eficaz.

Na Figura 14 listamos premissas apontadas por Krueger [Krueger 92] como fundamentais para que se possa reutilizar produtos de software. Como o armazenamento é parte importante no contexto de reuso, listamos aqui então essas premissas. É importante observar que os três tópicos apresentados no parágrafo acima estão indiretamente referenciados por Krueger. Prieto-Diaz [Prieto-Diaz 87] foi pioneiro ao introduzir um esquema de classificação facetada que atinge o ponto central de como armazenar componentes e de como possibilitar mecanismos de recuperação.

A classificação facetada de Prieto-Diaz tem dois níveis de descritores e três facetas para cada descritor. O primeiro descritor refere-se a funcionalidade e tem três facetas: função, objeto e meio. O segundo descritor refere-se ao ambiente e tem também três facetas: tipo do sistema, área funcional e cenário.

Nas Figuras 15 e 16 apresentamos um exemplo do aspecto de classificação. Nestas figuras mostra-se claramente o conceito `meta`, isto é algo que é geral e encontra-se num nível de abstração superior. Nessas figuras diferencia-se entre nível meta (meta conceito), nível de domínio (conceito) e nível de instância (exemplo). Note que aqui a semântica de domínio é uma liberdade em relação a definição de domínio (classe de aplicações).

5 Aspectos Gerais

É importante ressaltar que um modelo (fruto de uma linguagem ou combinação de lingua-

gens de representação) pode ser `profundo` ou `raso`. Com isso queremos dizer que um modelo pode ser bastante forte semanticamente ou muito menos forte. Um exemplo é se compararmos o modelo KAOS (Figura 17) com o modelo LAL (Figura 7). Modelos rasos são úteis principalmente nas fases exploratórias da Engenharia de Requisitos.

A seguir descreveremos de maneira suscinta quatro instrumentos de modelagem. Dois são métodos e os outros dois são técnicas. As técnicas estão mais orientadas para a facilidade de comunicação com os clientes e os métods são mais ligados nos aspectos caros a Engenharia de Software, isto é mais precisão e menos ambiguidade.

6 Léxico Ampliado da Linguagem

O Léxico Ampliado da Linguagem (LAL) é uma técnica que procura descrever os símbolos de uma linguagem. A idéia central do LAL é a existência da linguagens da aplicação. Esta idéia parte do princípio que no UdI existe uma ou mais culturas e que cada cultura (grupos sociais) tem sua linguagem própria. Portanto, o principal objetivo a ser perseguido pelos engenheiros de software na tarefa de elicitação do LAL é a identificação de palavras ou frases `peculiares` ao meio social da aplicação sob estudo. Somente após a identificação dessas frases e palavras é que se procurará o seu significado. A estratégia de elicitação é ancorada na sintaxe da linguagem e é formada por duas fases de coleta de fatos:

- identificação de símbolos da linguagem e
- identificação da semântica de cada símbolo.

6.0.1 Identificação de Símbolos

Através de uma técnica de coleta de fatos (p.ex. entrevistas informais, observação, leitura de documentos), o engenheiro de software anota as frases ou palavras que *parecem ter um significado especial na aplicação.* Estas palavras são, em geral, palavras chaves que são usadas com frequência pelos atores da aplicação. Quando uma palavra ou frase parecer ao engenheiro de software sem sentido, ou fora de contexto, há indícios de que esta palavra ou frase deve ser anotada. O resultado dessa fase é uma lista de palavras e frases.

A grande diferença entre a elicitação aqui proposta e a elicitação comumente praticada por analistas de sistemas é o enfoque. Enquanto na análise de sistemas, as estratégias de abordagem são usadas com o objetivo de elicitar as funções do sistema em estudo e suas saídas e entradas, na elicitação de linguagens da aplicação, as estratégias de abordagem são usadas com o objetivo de elicitar símbolos. Na elicitação de linguagens da aplicação uma das principais heurísticas é justamente a de *não procurar identificar funções da aplicação observada, mas apenas os seus símbolos.*

6.0.2 Identificação da Semântica

Com base na lista de símbolos o engenheiro de software procede a uma entrevista estruturada com atores da aplicação, procurando entender o que cada símbolo significa. Esta fase é a fase na qual o Léxico Ampliado da Linguagem é usado como um sistema de representação. Esta representação requer que para cada símbolo sejam descritos `noções` e `impactos`. Noção é o que significa o símbolo, impacto descreve os efeitos do uso/ocorrência do símbolo na aplicação, ou do efeito de algo na aplicação sobre o símbolo. Esses efeitos, muitas vezes, caracterizam

restrições impostas ao símbolo ou que o símbolo impõe. A descrição de impactos e noções é orientada pelos princípios de *vocabulário mínimo* e *circularidade.*

O vocabulário mínimo prescreve que ao descrever uma noção ou um impacto, esta descrição deve minimizar o uso de símbolos externos à linguagem, e que quando estes símbolos externos são usados, devem procurar ter uma representação matemática clara (p.ex. conjunto, união, interseção, função). A circularidade prescreve que as noções e os impactos devem ser descritos usando símbolos da própria linguagem. As experiências têm demonstrado que, na explicação da noção e do impacto, os atores da aplicação usam, naturalmente, do princípio de circularidade. A obediência ao vocabulário mínimo é de responsabilidade do engenheiro de software. O resultado dessa fase é o Léxico Ampliado da Linguagem.

6.0.3 Validação

As duas fases de identificação são interligadas por um ciclo de validação. A validação pode ser feita de maneiras distintas onde usamos principalmente uma validação informal com algumas nuances de validação por pontos de vista. Basicamente o ciclo de validação se apoia em três tipos de retroalimentação, a saber:

- necessidade de identificar novos símbolos e
- correção de noções e impactos.

6.1 Heurísticas para a Criação do LAL

O Léxico Ampliado é formado por um conjunto de descrições. O objetivo dessas descrições é prover a semântica dos símbolos da linguagem. A descrição semântica dos símbolos segue um sistema de representação onde cada símbolo deve ser descrito através de noções e impactos. A representação do Léxico é simples, cada símbolo é uma entrada e as noções e impactos são ítens dessa entrada. Nestes ítens, toda referência a outras entradas (princípio da circularidade) deve ser sublinhada. A Figura 18 mostra uma parte deste léxico da linguagem de custódia de ações.

Nas entrevistas com os atores do UdI foi possível observar que nas frases usadas para explicar os símbolos, estes podem ter o papel de sujeito, verbo ou objeto numa frase, ou expressar uma situação. Um símbolo pode ter mais de um papel. Por exemplo: um símbolo tem o papel de sujeito em uma frase, e aparece em outra frase como objeto. Abaixo listamos algumas das heurísticas usadas para distinguir entre noções e impactos.

- Cada símbolo pode ter zero ou mais impactos.
- Cada símbolo tem zero ou mais sinônimos.
- Cada símbolo tem uma ou mais noções.
- Escreva cada noção ou impacto usando frases simples, que expressem uma única idéia.
- As noções e impactos de um determinado símbolo podem representar pontos de vistas diferentes ou serem complementares.
- A descrição das noções e impactos deve respeitar a circularidade e o vocabulário mínimo.
- Para um símbolo que é sujeito em alguma frase, as noções associadas devem esclarecer quem é o sujeito e o impacto deve registrar quais as ações que executa.

- Para símbolos que têm o papel de verbo, as noções associadas devem dizer quem executa a ação, quando ela acontece, e quais os procedimentos envolvidos na ação. Já os impactos associados devem identificar as situações que impedem a ocorrência da ação, quais os reflexos da ação no ambiente (outras ações que deverão ocorrer) e quais as novas situações decorrentes da ação.
- Para um símbolo que é objeto em alguma frase, as noções devem definir o objeto e identificar os outros objetos com que se relaciona. No impacto, as ações que podem ser aplicadas ao objeto devem estar declaradas.
- Para símbolos que expressam uma situação, as noções devem esclarecer o que significa e quais são as ações que levaram a esta situação, e o impacto deve identificar outras situações e ações que podem ocorrer a partir desta situação específica.
- O Léxico deve ser feito cuidadosamente, à medida que o engenheiro de software vai aumentando seu conhecimento sobre a aplicação. Durante este processo de elicitação da linguagem, o engenheiro de software está sempre fazendo anotações, revendo e reescrevendo noções e impactos, e anotando novos símbolos relevantes para a aplicação.

Através do LAL é possível observar a integridade e variedade de locuções e termos desta linguagem, própria de uma aplicação. Pois, além da descrição de cada símbolo, os sinônimos estão explicitamente indicados, como também os relacionamentos entre os diversos símbolos.

As experiências têm demonstrado que os atores de uma determinada aplicação seguem naturalmente o princípio de circularidade, quando tentam explicar o significado do símbolo para o ambiente. Portanto, esta idéia de se ater aos termos da linguagem e o uso do Léxico apontam caminho próprio para a definição e validação da linguagem específica, base do conhecimento da aplicação em questão.

7 Requisitos como Sentenças em Linguagem Natural

Como dissemos na Introdução, DEFINIR REQUISITOS deve produzir um modelo, que será a base para a tarefa de desenho, e uma representação em linguagem natural que facilite o entendimento entre os vários atores do Universo de Informações. Portanto uma das maneiras de se representar requisitos é através de sentenças simples. Esse tipo de representação apresenta três grandes vantagens e três grandes desvantagens. As vantagens são:

- simplicidade,
- facilidade de entendimento pelos usuários e
- orientação à função

As desvantagens são:

- vagueza,
- orientação à função e
- necessidade de conhecimento prévio.

As vantagens são relacionadas principalmente com a facilidade de comunicação com os clientes. As desvantagens são justamente a falta de um maior estrutura, uma vez que nos concentramos nas entradas e saídas do sistema de software, isto é nas funções. A seguir detalhamos a representação de requisitos em formas de sentenças.

7.1 Sentenças de Requisitos

Existem três classes de sentenças: entrada, saída e mudança de estado. Cada requisito tem que seguir uma das duas estruturas abaixo.

> *O sistema deve* + [verbo + objeto | frase verbal] + [complemento de agente | null] + [condição | null].
>
> *O sistema deve* + [verbo + objeto | frase verbal] + [complemento de agente | null] + { a) condição-1, b) condição-2, ... condição-n}

Definições:

- **Verbo** é um verbo simples que expresse a funcionalidade daquele requisito.
- Dependendo do tipo do verbo, `objeto` pode ser um objeto direto ou um objeto direto seguido de um objeto indireto.
- **Frase verbal** é uma frase que expressa a funcionalidade do requisito.
- **Complemento de agente** é a identificação de um agente relacionado com o requisito. Algumas vezes esse complemento pode ser descrito pelo objeto indireto. Um agente pode ser uma pessoa, uma instituição, um grupo ou um dispositivo físico externo ao software.
- Uma **condição** é uma sub-sentença que reflete uma situação específica. situations.

Uma forma de adicionar mais estrutura, por exemplo no que diz respeito a conectivos é apresentada abaixo.

> *O sistema deve* +verbo + objeto + "para" + complemento + ["quando" | "se"] + condição.

7.2 Organização

Os *requisitos* são um conjunto de sentenças numeradas e classificadas segundo sua classe. Em caso de listas muito grandes, uma boa política é agrupar os requisitos com menor distância conceitual.

Abaixo listamos alguns exemplos.

Loja de Vídeo

1. O sistema deve emitir um recibo para o cliente. (saída)
2. O sistema deve cadastrar o cliente. (entrada)
3. O sistema deve transformar uma fita disponível em fita emprestada, quando a fita for alugada pelo cliente. (mudança de estado)

Biblioteca

1. O sistema deve cadastrar bibliotecários. (entrada)
2. O sistema deve cadastrar os usuários. (entrada)
3. O sistema deve achar para os bibliotecários, qual o usuário que está com um determinado livro. (saída)
4. O sistema deve tornar um livro em livro emprestado, quando um usuário pegar este livro emprestado. (mudança de estado)

8 SADT

O método SADT estrutura idéias sobre sistemas. Ele organiza os sistemas como modelos[1] tal que cada modelo tem um próposito e um ponto de vista bem definidos. O ciclo de produção de um modelo SADT objetiva fundamentalmente melhorar o entendimento do modelador sobre o sistema que esta sendo modelado.

SADT utiliza três principios básicos:

- *Everything worth saying about anything worth saying something about must be expressed in six or fewer pieces.* Isto é para expressar algo que mereça ser expresso, devemos nos expressar em 6 ou menos pedaços.
- Existe um ponto de vista.
- Existe um propósito.

Baseado nessas premissas, o SADT utiliza o conceito de decomposição para descrever sistemas. A descrição será um modelo do sistema de acordo com os princípios acima. Dessa maneira, SADT assume que o problema sendo modelado pode ser percebido como um sistema, isto é tem objetivos, entradas e saídas bem definidos.

No modelo descrito na Figura 7, descrevendo a operação da loja de vídeo, objetivo é modelar as principais atividades de uma loja de fitas de vídeo e o ponto de vista utilizado é utilizando conhecimento comum, retratar o funcionamento de lojas de vídeo. No entanto poderiamos modelar a operação da loja de vídeo com um diferente propósito e um diferente ponto de vista. Como por exemplo: modelar as transações financeiras da loja de vídeo e dar enfâse aos aspectos de contabilidade da loja.

A linguagem de representação utilizada pelo SADT, com já vimos anterirormente, é uma linguagem de setas e caixas com as seguintes características.

- Cada modelo é apresentado segundo duas perspectivas, a perspectiva de dado (datagrama) e a perspectiva de processo (actigrama).
- cada modelo é composto de uma série de diagramas, os quais representam a decomposição hierárquica (Figura 19). O no A-0 é um diagrama de só uma caixa. Todos os outros nos seguem o princípio que garante que nenhum diagrama terá mais do que seis caixas.

[1] Neste caso estamos usando o termo modelo de uma maneira mais restrita, aqui a melhor metáfora seria a de uma maquete quando usada em arquitetura.

- As setas em SADT representam controle, entrada, saída e mecanismo.

Para construir um modelo, o modelador deve seguir um processo com 6 etapas:

- coletar fatos,
- determinar objetivo e ponto de vista,
- produzir um lista de coisas,
- produzir uma lista de atividades,
- agrupar as atividades (no caso do actigrama) ou agrupar as coisas (no caso do datagrama) e
- desenhar o diagram A-0.

Tanto requisitos funcionais como não funcionais (restrições) podem ser expressos num modelo SADT. Requisitos não funcionais podem ser explicitamente representados pela seta controle, que determina de que maneira uma atividade é feita ou restringe uma coisa. Requisitos funcionais são expressos basicamente pelas setas de saída. SADT tem uma sintaxe bem definida, mas o mesmo não podemos dizer sobre sua semântica.

Uma caractéristica importante da linguagem SADT é a seta de controle. Uma regra em SADT determina que nenhuma atividade (uma caixa no actigrama) pode existir sem uma seta de controle. Ou seja, é impossível ter uma atividade sem restrições ou sem políticas que regulem seu comportamento. Nós acreditamos que esta particulariedade do SADT é relacionada a idéia de sistemas de controle, os quais fazem uma distinção muito clara entre entrada e controle. Controle é o conceito mais difícil para iniciantes em SADT, mas uma vez dominado passa a ser um excelente aliado na expressão de modelos.

Outra caractéristica da linguagem do SADT é a seta de mecanismo. Esta seta torna possível a identificação de responsabilidade na consecução de uma atividade. No caso do datagrama permite expressar uma atividade que serve de suporte a uma coisa.

O método SADT também dispõe de um processo de validação baseado em revisões. Em SADT esse processo é chamado de ciclo autor/leitor. Neste ciclo, leitores apontam problemas através de anotações nos diagramas. O autor reage a essas anotações escrevendo de volta (em cor diferente) nos diagramas. Em alguns casos a argumentação entre leitores e autores leva a uma conversa entre os dois para esclarecer possíveis mal entendidos.

Assim como Ross, acreditamos que a linguagem SADT impõe uma disciplina de pensamento que nos ajuda no entendimento e na estruturação de situações do mundo real. O aspecto que é importante salientar é que fazer modelos em SADT incorpora um processo de aprendizado, quer pelo uso da linguagem como também pelo processo de revisão do ciclo autor/leitor. É muito difícl descrever controle, mecanismo, entrada e saída para cada caixa e conecta-las se não se tem um entendimento do sistema que ser quer modelar, por isso acreditamos que modelos em SADT são muito úteis para a Engenharia de Requisitos.

8.1 Utilizando a Linguagem do SADT

Em computação a estrutura conhecida como árvore é muito utilizada. Esta estrutura é basicamente uma árvore de cabeça para baixo, isto é, a `raiz` é em cima e as folhas são em baixo. Na árvore da computação, a raiz é representada por um ponto, conhecido como `raiz-nó`. Deste

nó partem linhas, chamadas **arcos**. Este arcos, então, terão em suas pontas outros **nós**. Esses nós podem ter arcos partindo deles e novos nós vão surgido, até que se chega a nós dos quais não partem mais arcos. Estes nós, então, são as **folhas** da estrutura árvore. Observe que ,nesta explicação, o que na verdade estamos fazendo é uma hierarquia semelhante à hierarquia familiar (árvore genealógica), ou seja, a raiz é o pai de todos os nós, que então serão pais dos netos da raiz e assim por diante (ver Figura 19).

O conjunto de diagramas de um modelo SADT se organiza como uma árvore. O primeiro deles, a raiz, é justamente o diagrama A-0. O segundo diagrama é composto de todos os nós filhos da raiz. O primeiro diagrama, conforme já vimos, é chamado de diagrama A-0 e este segundo diagrama é chamado A0.

A partir do diagrama A0, se utilizarmos a metáfora da árvore, observamos que a cada nível de profundidade da árvore, poderemos ter mais de um diagrama. Ou seja, poderemos ter vários diagramas em cada nível n, onde n é maior do que 1. Para evitar árvores mal formadas, impõem-se uma restrição à proliferação de filhos de cada nó. Esta restrição é a seguinte: cada nó, só pode ter no máximo 6 filhos e no mínimo 3.

A seguir enunciamos algumas regras gerais para a produção de modelos SADT.

1. Os diagramas de um modelo SADT representam uma hierarquia de abstrações (pai, filho, neto, ...). Detalha-se de cima para baixo as atividades de um sistema.

2. Um modelo SADT pode ser descrito por duas perspectivas diversas, a perspecitva de atividades (actigrama) ou a perspectiva de coisas (datagrama).

3. Tudo que é entrada para o pai, é entrada para seus descendentes. Tudo que é saída para o pai, é saída para seus descendentes. Tudo que é controle para o pai é controle para seus descendentes. Tudo que é mecanismo para o pai é mecanismo para seus descendentes.

4. Todo diagrama em qualquer nível, exceto o A-0, deve ter, no mínimo, 3 processos e, no máximo, 6 processos.

5. Uma seta num diagrama filho pode ser um detalhamento de uma seta no diagrama pai.

6. A decomposição de uma atividade ou de uma coisa termina quando o detalhamento não é mais necessário.

7. A leitura de um conjunto de diagramas deve refletir o objetivo e o ponto de vista enunciados no diagram a-0.

8. O nome de qualquer um dos componentes de um modelo SADT é único.

9. Toda atividade tem, no mínimo, um controle e uma saída.

10. Toda coisa tem, no mínimo, uma entrada e uma saída.

11. Na passagem do nível N para o nível N+1 podemos ter no máximo X diagramas onde X é o número de atividades/coisas do diagrama no nível N.

9 Jackson System Development

Michael Jackson é um conhecido cientista na área de Engenharia de Software. Seu primeiro livro, JSP - Jackson Structured Programming, de certa maneira revolucionou a maneira com que programas eram desenvolvidos. Jackson propôs que os programas fossem desenvolvidos em função das estruturas de dados que estes iram manipular, ao ínves da maneira tradicional em que os dados são acessórios as descrições orientadas a processo. O método JSD herdou algumas das ídéias de JSP, mas constitui-se em um trabalho muito mais amplo e com seus fundamentos muito bem embasados.

JSD é um método de desenvolvimento de sistemas. É um método para especificar e implementar sistemas de software. A hipótese básica do JSD é a de que um modelo do mundo real produz uma máquina que descreve como o mundo real se comporta. Os modelos JSD são descrições de situações reais organizadas em volta de entidades do mundo real. Essas entidades reunem em volta de si um série de ações que acontecem na realidade. Essas ações acontecem no ponto particular do mundo real e são consideradas como atômicas.

O sistema de software, que modela o mundo, simula esse mundo. As entidades são modeladas como um rede de processos que se comunicam por passagem de mensagem e por inspeção de leitura. Essa rede é a base para a simulação. Dessa maneira a funcionalidade do sistema é entendida por JSD como respostas do sistema a consultas sobre a simulação executada num computador. Esse entendimento faz uma clara distinção entre a estrutura básica do sistema e as funções que ele executa. O principal argumento para este entendimento é, que dessa maneira, os software seriam mais fáceis de manter.

O desenvolvimento em JSD começa pela coleta de fatos para ajudar na identificação das ações e das entidades, as quais serão representadas por listas. Através da identificação das ações é possível descrever a dinâmica do mundo real. Em paralelo com a identificação de ações também, identificam-se as entidades. Depois da identificação, as ações são agrupadas nas entidades, com o cuidado que não se tenha entidades escondidadas dentro de outras. As ações são agrupadas a uma entidade no diagrama de estrutura, que usa uma árvore e os mecanismos de controle clássicos:

- sequência,
- interação e
- seleção.

A entidade é a raiz da árvore e as ações as folhas. A dinâmica do mundo real são descritas pelos mecanismos de controle (Figura 5). Além da descrição gráfica, JSD tem uma linguagem própria para descrever linearmente a entidade, esta linguagem chama-se texto estruturado.

No modelo inicial as entidades passam a se chamar processos e são interligadas por duas formas de comunicação: sequência de dados (uma fila) e vetor de estado (inspeção de memoria do processo). No primeiro modelo expresso pela rede, cada entidade é representada duas vezes, uma sendo a entidade real (modelo 0) e outra sendo o modelo dessa realidade (modelo 1). Dessa maneira expressa-se na rede a comunicação do modelo com o exterior. A Figura 5 mostra o modelo inicial.

Uma segunda versão da rede, as funções, expressam a funcionalidade requirida pelos sistema. Funções são as provedoras das informações no sistema, que as responde de acordo com a solicitação e o estado da simulação. Funções podem ser embutidas em processos ou podem elas próprias serem um processo.

Uma terceira versão da rede cuida do sincronismo entre os processos. Após o terceiro modelo, deriva-se então o diagrama de implementação, que é o desenho do sistema a ser implementado. Como nosso interesse no JSD prende-se ao desenho lógico, não nos detemos sobre o modelo de sincronismo e nem o modelo de implementação, detalhes podem ser encontrados em [Masiero 92] que apresenta de uma forma suscinta o método JSD.

10 Resumo

Nosso objetivo nessa Parte foi alinhar alguns conceitos gerais sobre modelagem. Ressaltamos a importância do aspecto organização e mostramos a diferença fundamental entre especialização e decomposição. Também ressaltamos os pontos básicos de representações. Mostramos também a importância de corretamente armazenarmos os conhecimentos modelados, para que sejam passíveis de reutilização no futuro. Como o interesse fundamental da Engenharia de Requisitos é entender o que se quer do software, salientamos o uso de modelos rasos, que são mais simples de serem utilizados e validados. Vale observar aqui, que um maior detalhamento e comprometimento surgirá na concepção da arquitetura para satisfazer esses requisitos. Finalizamos apresentando em mais detalhes duas técnicas (LAL e Sentenças) e dois métodos que podem ser utilizados na modelagem de requisitos.

Página em Branco

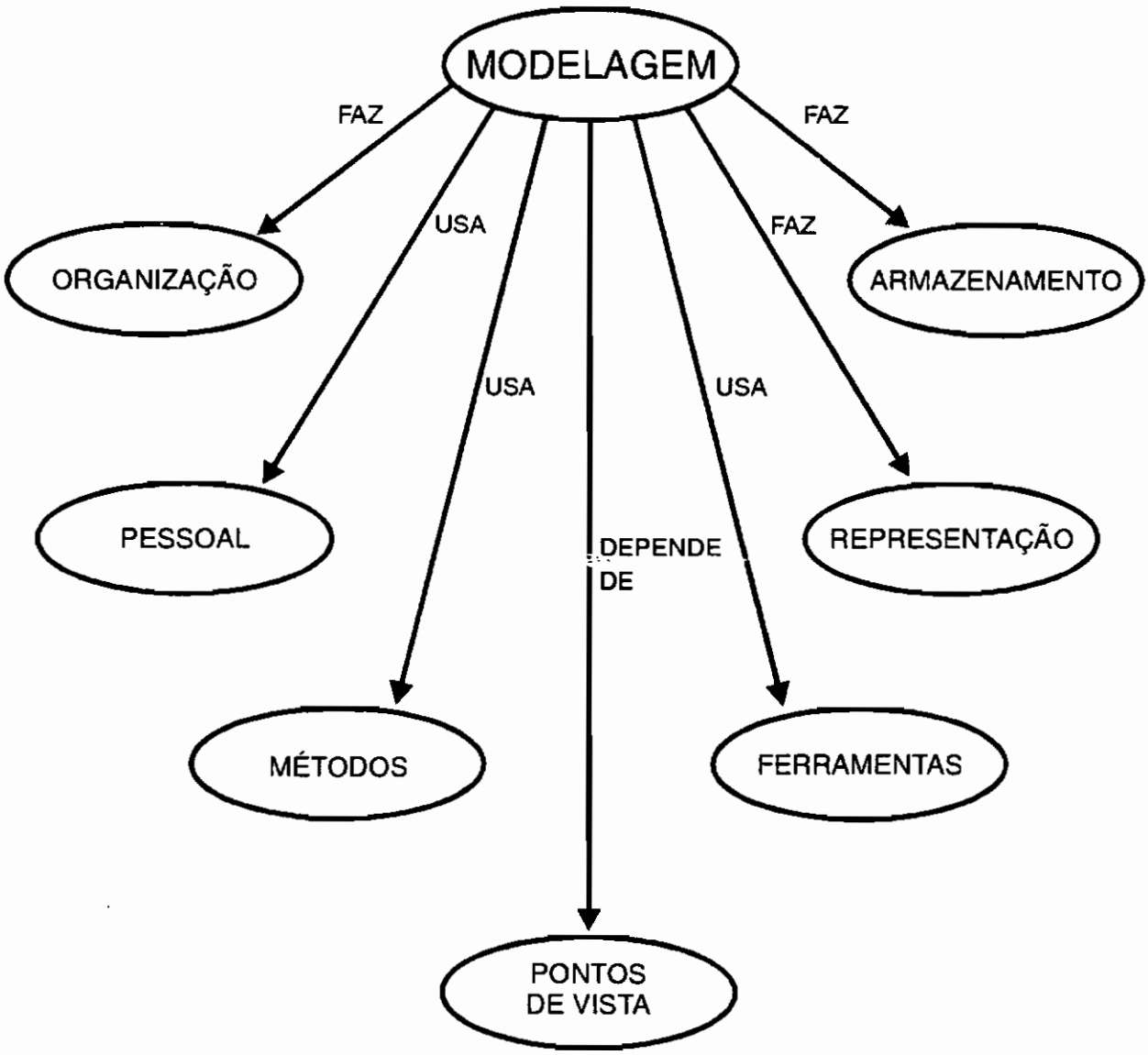


Figure 1: Modelagem

É-UM (PEDRO, CARIOCA)

É-UM (CARIOCA, BRASILEIRO)

$\forall x \; \forall y \; \forall z$ É-UM (x,y) $\wedge$ É-UM (y,z)

$\longrightarrow$ É-UM (x,z)

Figure 2: Lógica

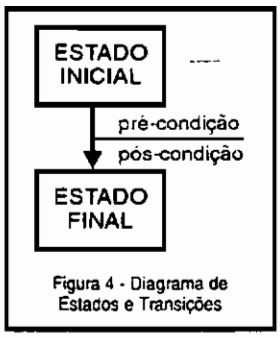
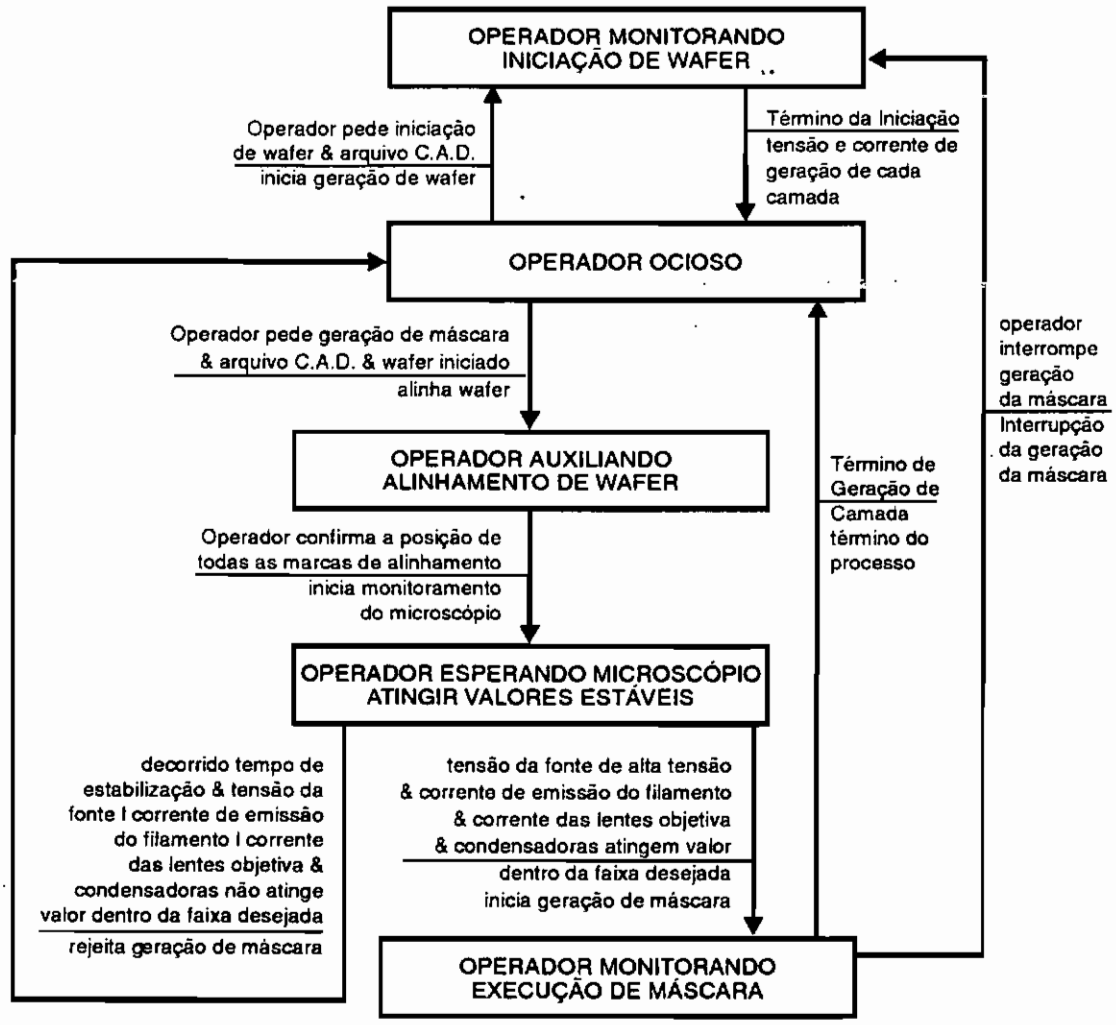


Figura 4 - Diagrama de Estados e Transições



Esquema da Dinâmica Representando o Comportamento do Processo sob o Ponto de Vista do Operador

Figure 3: Máquinas de Estado e Pré e Pós Condições

EXEMPLO DE LAL DE UMA BIBLIOTECA

(Ana Paula Franco)

- CONSULTA

  - NOÇÃO:
  1- Leitura de um **exemplar** dentro da **biblioteca.**
  2- Realizada por um **usuário.**

  - IMPACTO:
  1- **Exemplar** está na **biblioteca.**
  2- **Exemplar** não está na estante (**obra está na estante**).
  3- Após **consulta**, **bibliotecário** deve **colocar obra na estante.**

- DAR CABEÇALHO DE ASSUNTOS

  - NOÇÃO:
  1- Tarefa realizada pelo **bibliotecário** durante a **classificação** da **obra.**
  2. Especificação dos **assuntos** da **obra.**

- DATA DE PUBLICAÇÃO

  - NOÇÃO:
  1- Mês e ano em que foi publicada uma **obra.**

- DATA DE DEVOLUÇÃO

  - NOÇÃO:
  1- Data estabelecida para **devolução** de uma **obra emprestada.**

  - IMPACTO:
  1- Se **data de devolução** é menor que a data atual, **obra está atrasada.**

Figure 4: Léxico Ampliado da Linguagem

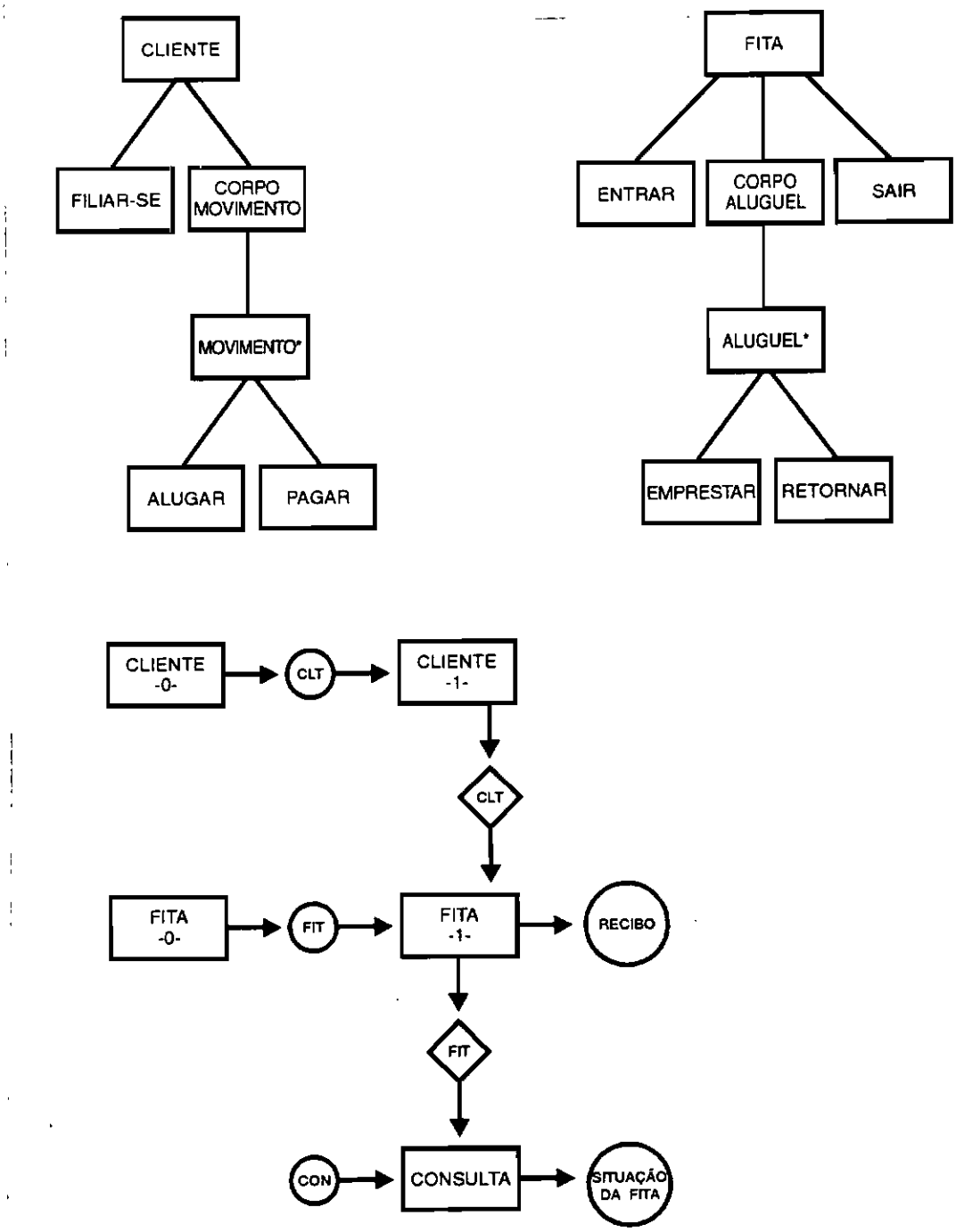

CLIENTE
FILIAR-SE
CORPO MOVIMENTO
MOVIMENTO*
ALUGAR
PAGAR
FITA
ENTRAR
CORPO ALUGUEL
SAIR
ALUGUEL*
EMPRESTAR
RETORNAR
CLIENTE -0-
CLT
CLIENTE -1-
CLT
FITA -0-
FIT
FITA -1-
RECIBO
FIT
CON
CONSULTA
SITUAÇÃO DA FITA

Figure 5: JSD

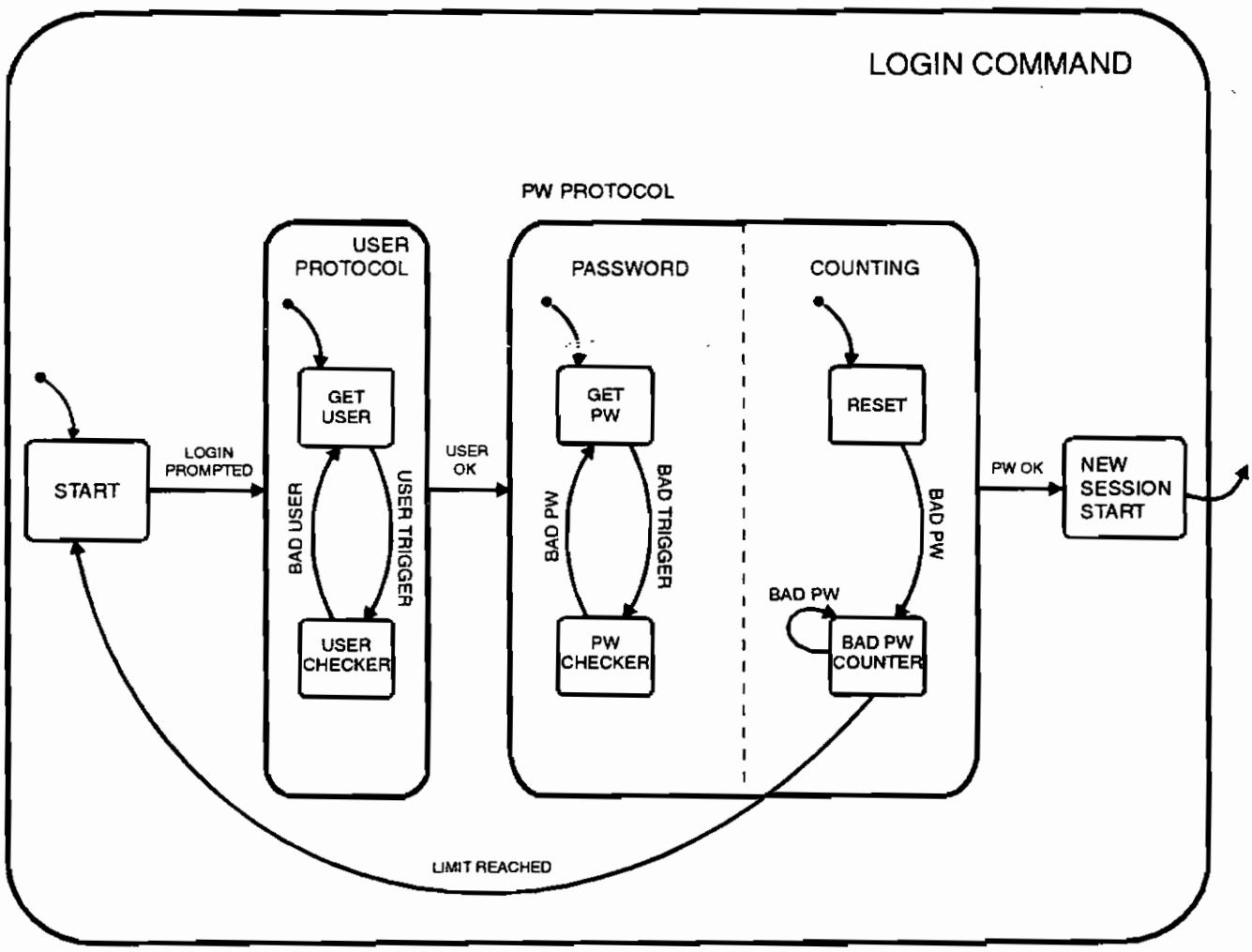
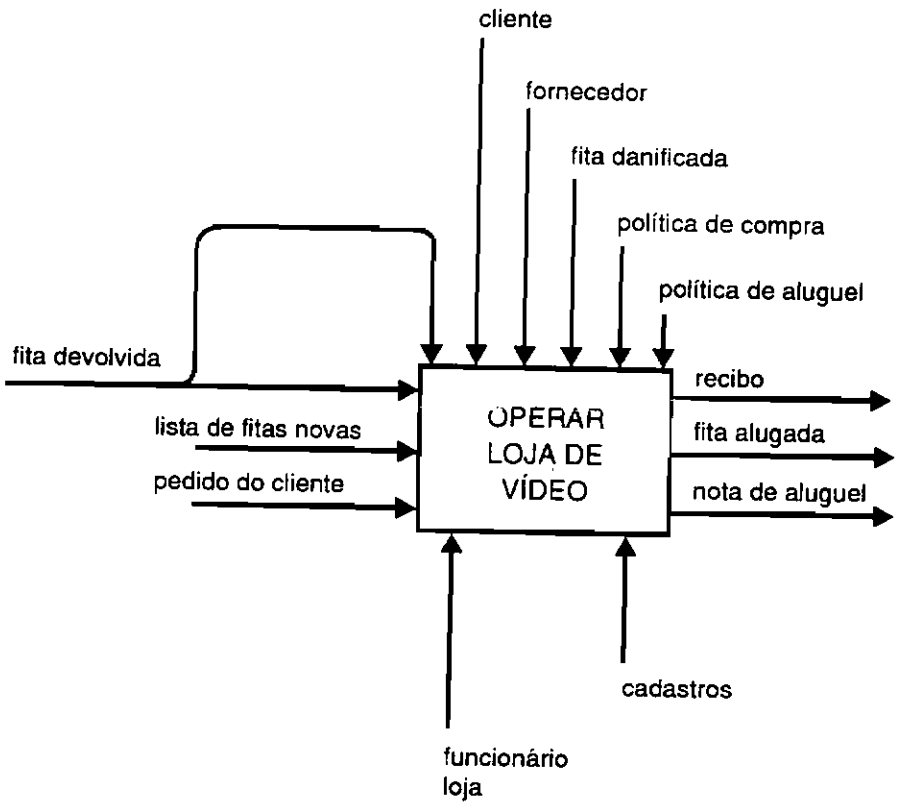


Figure 6: Estadograma



OBJETIVO: MODELAR AS PRINCIPAIS ATIVIDADES DE UMA LOJA DE FITAS DE VÍDEO

PONTO DE VISTA: UTILIZANDO CONHECIMENTO COMUM, RETRATAR FUNCIONAMENTO DE LOJAS DE VÍDEO

Figure 7: SADT

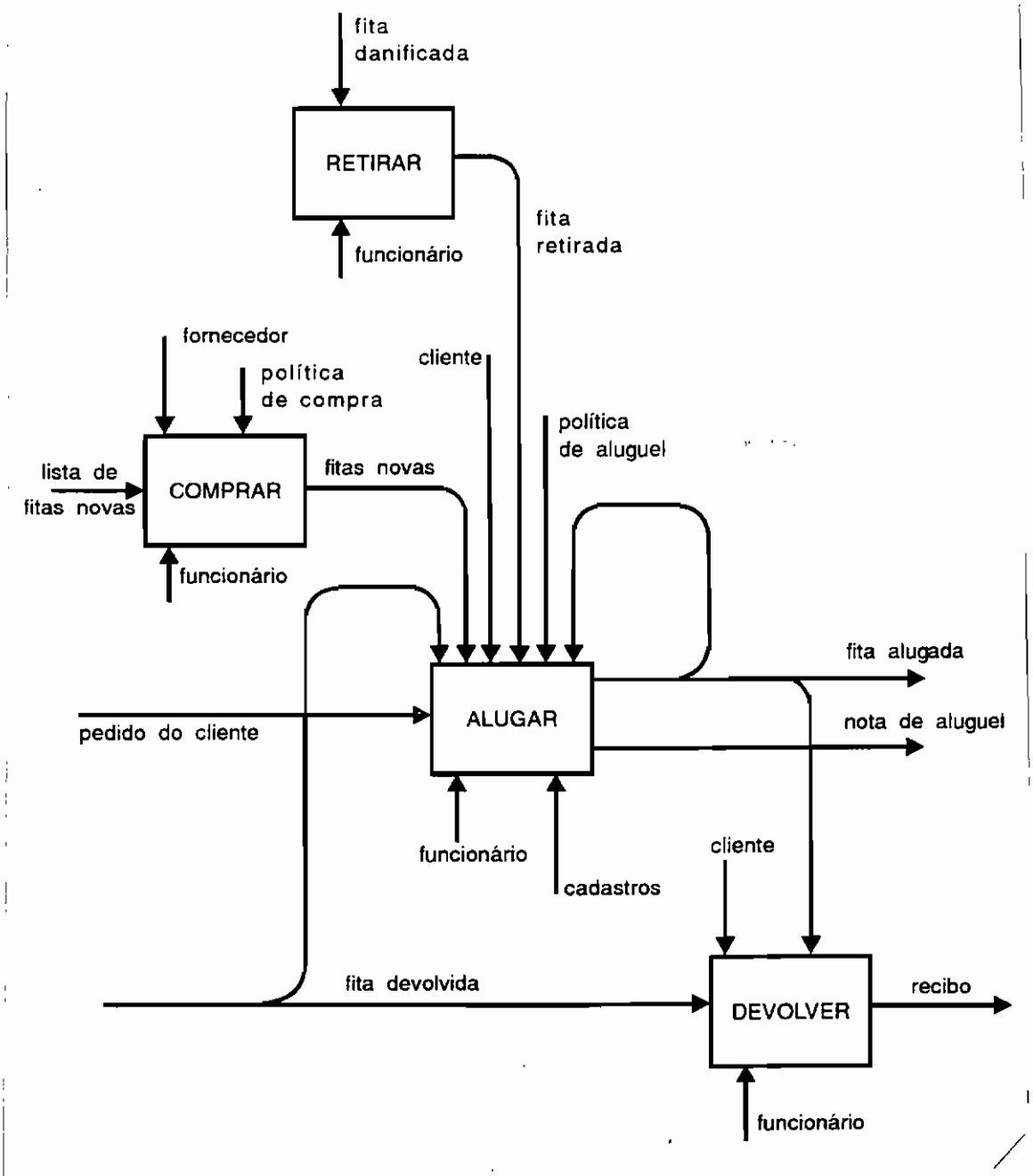


Figure 8: SADT

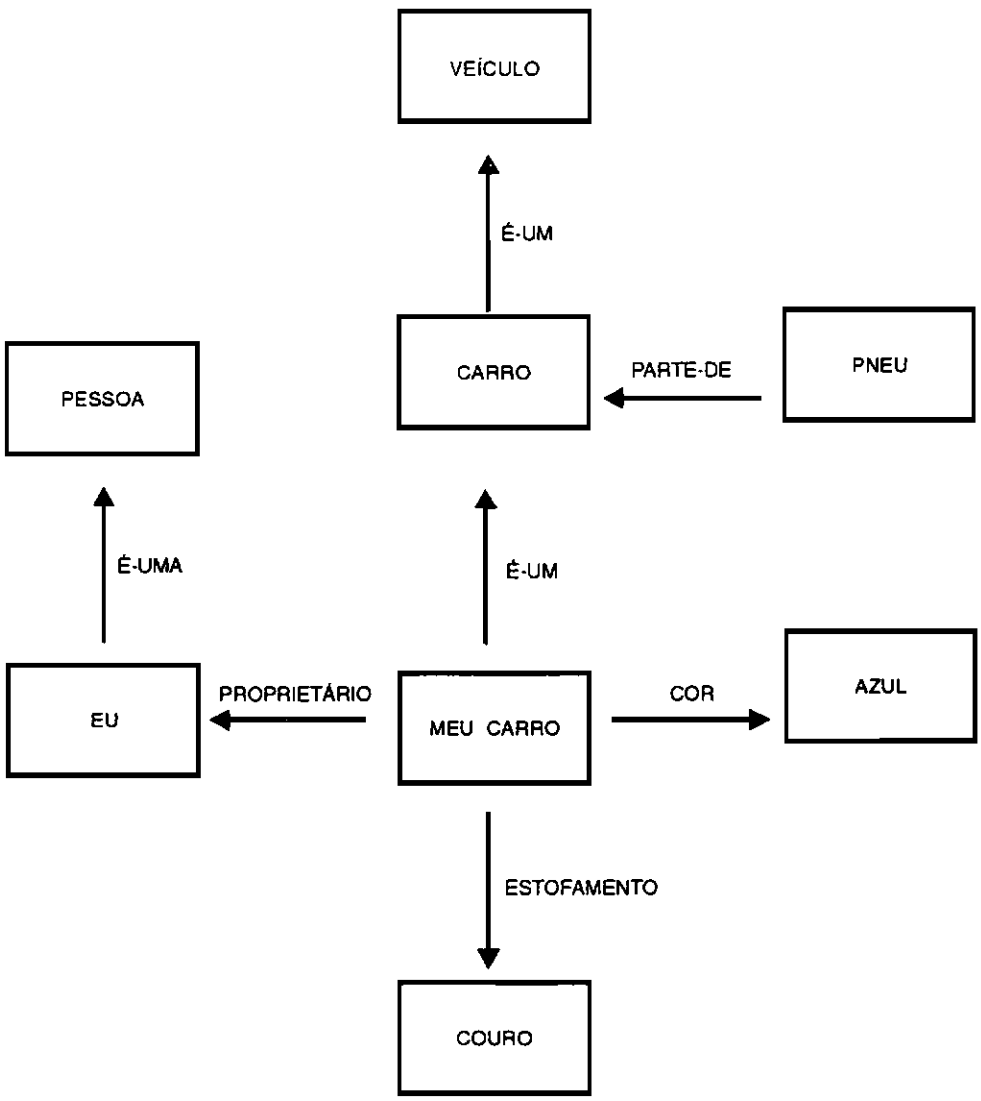


Figure 9: Redes Semântica

BOB'S

SUPER-CLASSE: LANCHONETE

CLASSE: FAST FOOD

PAPÉIS:

- Cx - caixa
- A - atendente
- U - usuário
- Cz - cozinheiro

ENTRADA:

- U tem fome
- U tem $

SAÍDA:

- U não tem fome
- U tem menos $
- Cx tem mais $

CENA 1:

- U entra no Bob's
- U olha tabela de preço
- U dirige-se à caixa

CENA 2:

- U pede ítens
- Cx digita ítens
- U entrega $
- Cx entrega [$] e ficha a U

CENA 3:

- U entrega ficha a A
- U diz ítens
- A grita ítens
- Cz prepara ítens
- A entrega ítens a U

Figure 10: Script

Se diretor do filme é conhecido,

e atriz do filme é bonita

e filme é comédia

e cinema é perto.

Então veja o filme.

```
( ((diretor = filme = nome-dir)  (conhecido = nome-dir)
   (atriz = filme = nome-atr)  (bonita = nome-atr)
   (tipo-filme = filme = tipo-fil)  (comédia = tipo-fil)
   (cinema = filme = nome-cin) (perto = nome-cin))
⇒
   (add-to um  (veja = filme)) )
```

Figure 11: Regra

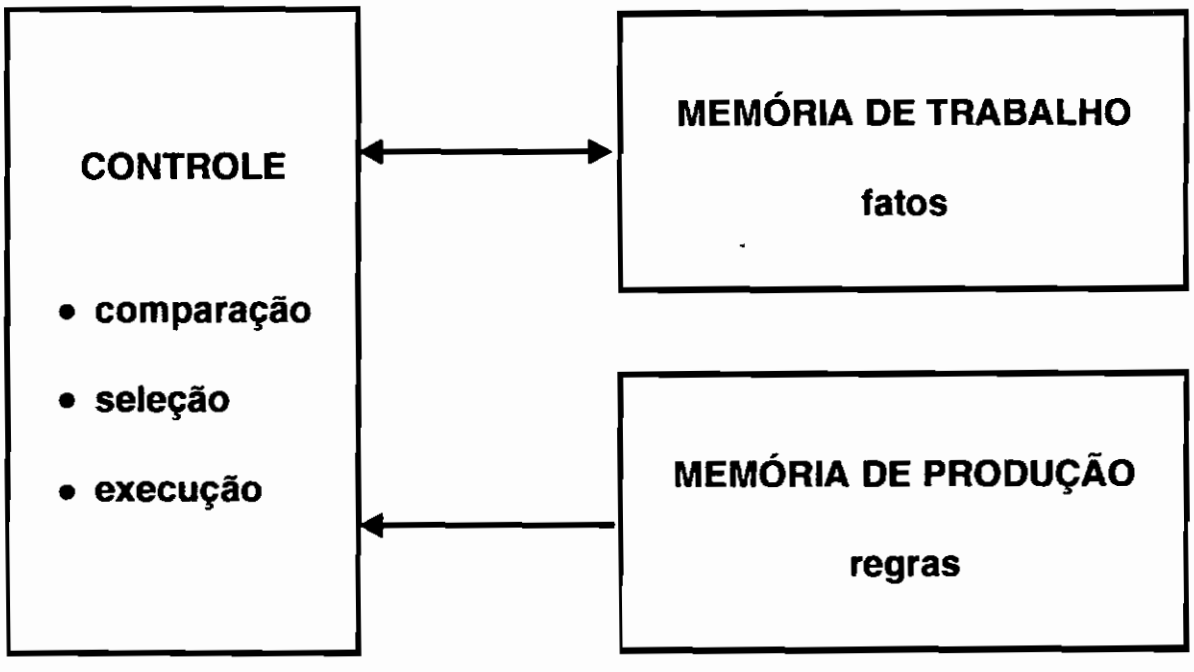
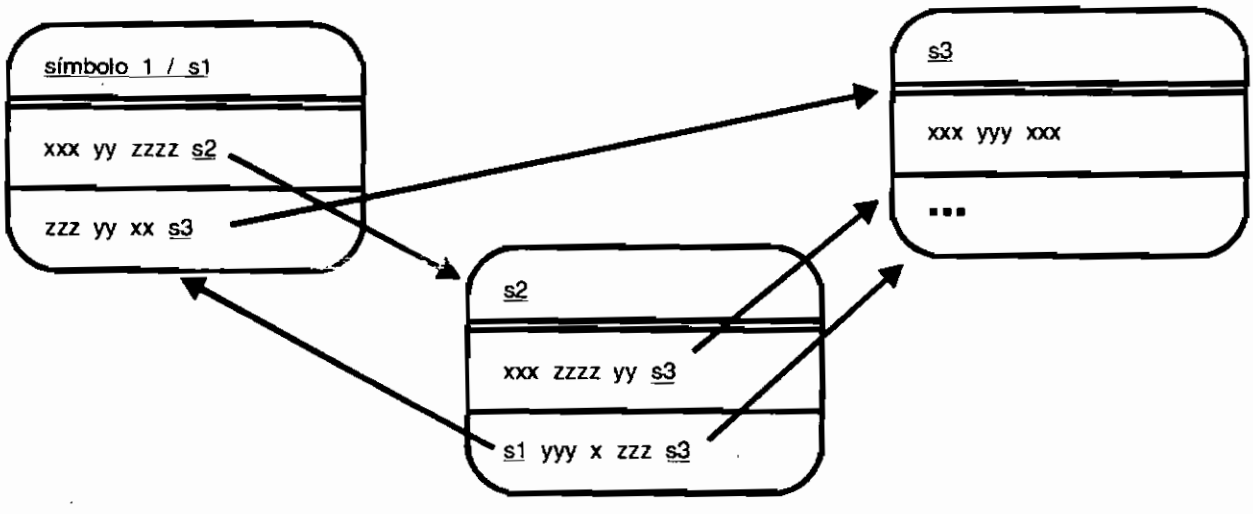


Figure 12: Sistemas de Produção

Léxico Ampliado da Linguagem ⟶ Hiperdocumento



Fragmentação - cada entrada expressa um conceito (símbolo).

Ligações - na descrição semântica dos símbolos, as referências a outros nós existem naturalmente (princípio da circularidade).

Figure 13: LAL e Hipertexto

a) Para que uma técnica de reuso seja efetiva, ela deve reduzir a distância cognitiva entre o conceito inicial de um sistema e sua implementação executável final.

b) Para que uma técnica de reuso seja efetiva, deve ser mais fácil reutilizar os artefatos do que desenvolver o software do nada.

c) Para selecionar um artefato para reuso, deve-se saber o que ele faz.

d) Para reutilizar um artefato de software de maneira efetiva, deve-se ser capaz de achá-lo mais rápido do que construí-lo.

Figure 14: Premissas de Krueger

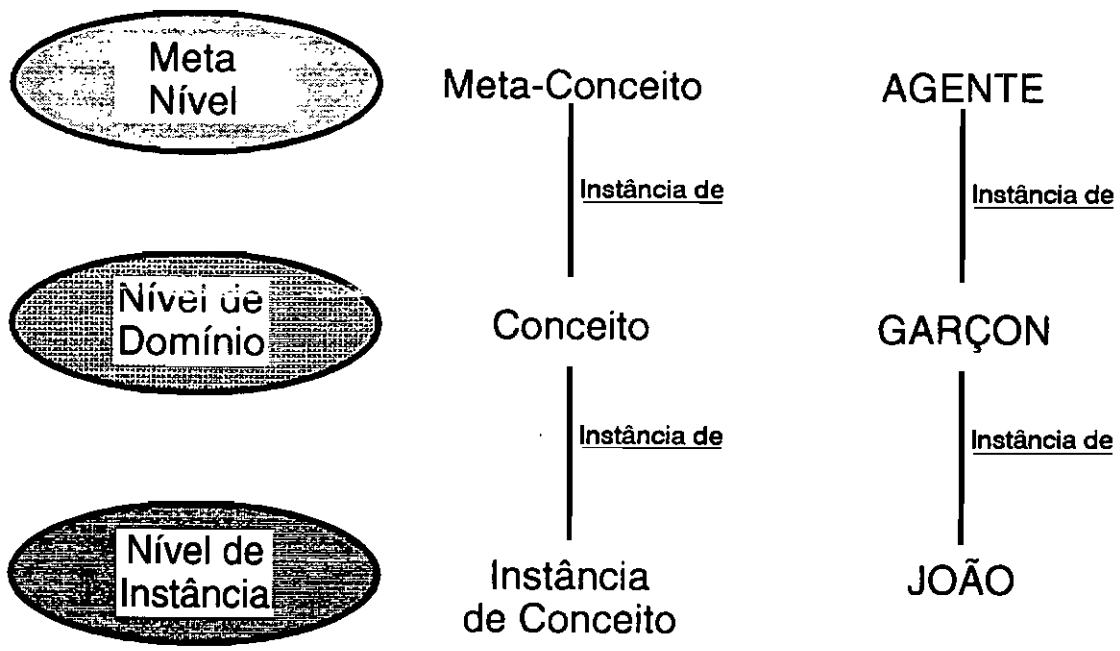


Figure 15: Meta Conceito X Conceito X Instância

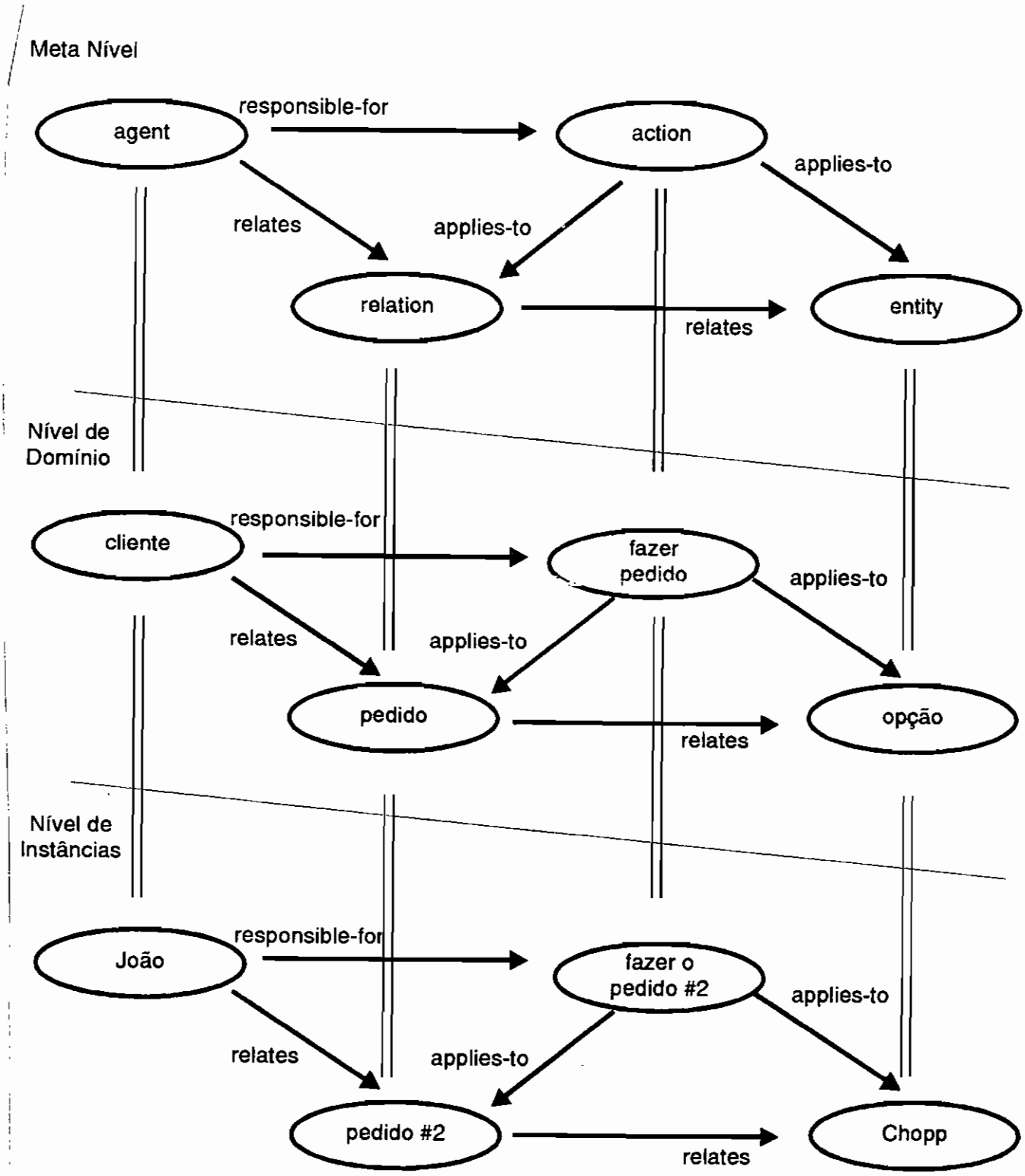


Figure 16: Exemplo

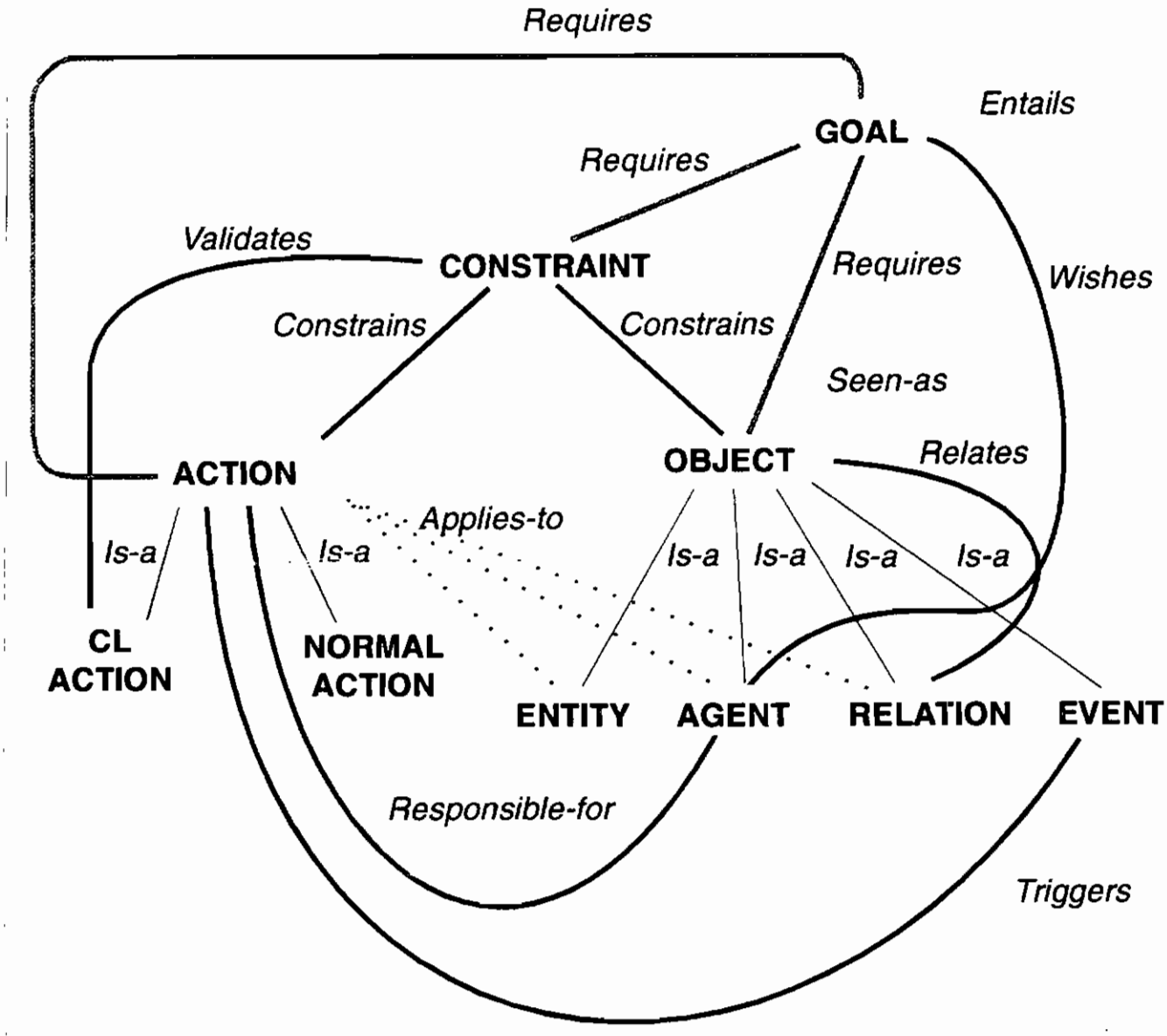


Figure 17: Modelo KAOS

- CLIENTE / CONTA-CLIENTE / CONTA-FILHA

- NOÇÃO:

1- Número da **conta de um cliente cadastrado** na **bolsa de valores** como **cliente** da **distribuidora.**

- IMPACTO:

1- Após **operação de compra** ou **venda**, a **bolsa deposita** ou **retira** as **ações nominativas** da **conta-cliente.**

2- Após **operação de compra**, a **distribuidora transfere** as **ações ao portador compradas**, da **conta-mãe** para a **conta-filha.**

3- Após **operação de venda**, a **distribuidora transfere** as **ações ao portador vendidas**, da **conta-filha** para a **conta-mãe.**

- CONTA-MÃE / CONTA-OPERACIONAL

- NOÇÃO:

1- Número da conta de uma **distribuidora cadatrada** na **bolsa.**

- IMPACTO:

1- Após uma **compra** de **ação ao portador**, para um cliente através de uma **distribuidora**, o **lote comprado** será **trasferido** da **conta-mãe** para a **conta-filha.**

Figure 18: LAL da Custódia de Ações

References

[Harel 88] Harel, D. On Visual Formalisms, *Communications of the ACM*, Vol. 31, No. 5, 1988.

[Krueger 92] Krueger, A Survey on Software Reusability, *ACM Computing Surveys*,v. 24, n. 2, Jun., 1992, pp.131-184.

[Masiero 92] Masiero, P.C., *Análise Estruturada de Sistemas pelo Método de Jackson*, Editora Edgard Blucher, LTDA, São Paulo, 1992.

[Prieto-Diaz 87] Prieto-Diaz, R. and Freeman, P.A., Classifying Software for Reusability, *IEEE Software*, vo. 4, n. 1, Jan. 1987, pp. 6-16.

[Sanches 94] Sanches, M., Maffeo, B., Leite, J.C.S.P., Ferramentas e técnicas para modelagem da essência de sistemas de tempo real para controle e monitoramento de processos. *Aceito para publicação nos Anais do Congresso Latino-Americano de Controle e Automação, 6*, 1994.

[Ross 77] Ross, D. Structured Analysis (SA): A Language for Communicating Ideas. In *Tutorial on Design Techniques*, Freeman and Wasserman, Eds., IEEE Computer Society Press, Long Beach, CA 1980, pp. 107-125.

Engenharia de Requisitos
Notas de Aula, Parte V

Julio Cesar Sampaio do Prado Leite
Pontifícia Universidade Católica do Rio de Janeiro
R. Marquês de S. Vicente 225 Rio de Janeiro 22453
Brasil

1994

1 Introdução

Nesta parte das Notas vamos centrar nossa atenção na tarefa ANALISAR. Vale lembrar que a palavra análise é utilizada por muitos autores como o termo que abrangeria o que aqui chamamos de Engenharia de Requisitos. Para nós a palavra análise tem sua semântica fundamentalmente ligada a tarefa de verificação e validação. Ou seja, fundamentalmente acreditamos que a tarefa analisar só se aplica quando temos uma representação, que pode então ser analisada.

Da mesma forma que fizemos para as outras tarefas de DEFINIR REQUISITOS, vamos apontar as partes da análise. Essas partes servem para melhor entender a tarefa de ANALISAR. Essas partes são (Figura 1):

- Identificação de Partes,
- Verificação e
- Validação.

No caso da análise o grande problema dessa divisão entre partes é distinguir exatamente o que vem a ser verificação e o que consideramos validação (Figura 2). Algumas estratégias, como a inspeção, por exemplo, podem tanto ser ditas como uma estratégia de validação, já que envolve o conhecimento do indivíduo e portanto não formal, mas também podemos justifica-la como verificação visto que estamos procurando identificar problemas em modelos.

A Figura 3 mostra o que chamamos do loop da análise, isto é o processo básico que utilizamos para análisar um modelo. A identificação de partes ocorre entre o MODELO e COMUNICAÇÃO na perna direita da figura. A decisão PROBLEMAS é justamente a pergunta constante que se deve fazer sobre o modelo. Se tudo estiver bem, o modelo nos serve, senão teremos que retornar ao UdI.

Outra forma de visualizar este loop da análise, é através do que chamamos de COMPUTAÇÃO DELTA (Figura 4). Deltas são os tipos de problemas que uma estratégia de análise é capaz de achar. Dependendo da estratégia, teremos maior ou menor facilidade de comunicarmos ao UdI a necessidade de maiores informações. Para cada estratégia estudada, fazemos uma avaliação

da sua computação delta. As diferenças entre as estratégias de validação são determindas pelo tipo e qualidade das entradas fornecidas para a computação Delta.

Nosso objetivo nessa Parte das Notas é detalhar as subtarefas da análise. Na Seção 2 descreveremos como identificar as partes que serão validadas ou verificadas, na Seção 3 cuidaremos da verificação e Seção 4 cuidaremos da validação. Na Seção 4 comentamos sobre aspectos gerais na Seção 5 descrevemos o método de Inspeções e na Seção 5 detalhamos a estratégia de Pontos de Vista. Finalizamos com um resumo do que foi apresentado.

2 Identificação de Partes

Antes da tarefa de análise é importante que saibamos como está organizado e armazenado o modelo objeto de nossa análise. Ou seja, precisamos, antes de partirmos para a análise, fazer uma especie de planejamento no qual procuramos identificar partes do modelo. Essa identificação de partes tem uma estreita ligação tanto com a modelagem (armazenamento, organização) como com a elicitação (identificação das fontes de informação).

No caso da modelagem, dependendo do método adotado teremos maior ou menor facilidade de identificar partes do modelo. Esta identificação é importante, porque nos ajudará na tarefa de análise, visto que analisar sistemas complexos por inteiro é não é uma tarefa sensata.

Por outro lado quando formos proceder a análise é importante não só termos as partes que iremos analisar, mas também a ligação dessas partes com fontes de informação do Universo de Informações. Na verdade essa tarefa de identificar partes desempenha importante papel na noção de rastreamento de requisitos. O rastreamento é uma propriedade importante, mas infelizmente pouco explorada, ela deve ser responsável por ligar a fonte de informação do UdI a informação conforme registrada no modelo (Figura 5).

Identificando as partes e ligando-as ao UdI, para que o processo do loop tenha a quem se dirigir no UdI, podemos então estabelecer políticas de análise, isto é quais as partes que terão prioridade, que tipo de varredura será executada e se usaremos um enfoque estatístico. Vale lembrar que estas questões são questões que a área de teste coloca em evidência. Uma das máximas utilizadas na área de testes é que muitas vezes a maioria dos problemas encontram-se em uma pequena parte do sistema.

3 Verificação

Se quizermos ser rígidos, só poderiamos entender verificação entre modelos e se possível sem ajuda humana. No entanto vamos considerar verificação uma análise de modelos sem que haja direta comparação com o Universo de Informações. Essa análise poderia ser desempenhada tanto por seres humanos como por software segundo regras bem definidas.

Várias estratégias de verificação de software tem sido propostas pela literatura. Se por hipótese, definirmos que algumas dessas estratégias também podem ser aplicadas a elicitação de requisitos, teremos um conjunto, do qual se sobressaem as seguintes estratégias:

- Inspeções,
- Uso de formalismos e
- Reusando Domínios.

A técnica de Inspeções [Fagan 86] e um método de revisão de software que pode ser aplicada a definição de requisitos. A grande vantagem do método de Inspeções é a sua estruturação de um trabalho cooperativo. Fundamentalmente, Inspeções utiliza-se de uma equipe que procura descobrir o máximo de defeitos possíves no software enquanto documento, isto é sem utilizar-se da demonstração do comportamento desse software. Inspeções são feitas baseadas em critérios pre-determinados e em um conjunto de perguntas chave (*cheklist*). Através do processo de uma equipe de inspeção descobre-se problemas em um documento. No caso da definição de requisitos esse documento poderia ser, por exemplo, uma lista de requisitos. Inspeções encontram defeitos, os quais consequentemente controlam a computação Delta:

$$\Delta = \textbf{defeitos}\ (\text{fatos, checklist, informação})$$

No uso de formalismos, onde o engenheiro de software faz o papel de um provador de teoremas, a verificação ocorre dada a possibilidade de se identificar inconsistências, as quais consequentemente controlam a computação Delta:

$$\Delta = \textbf{inconsistências}\ (\text{fatos, informação})$$

Usando-se a técnica de reutilização de domínios [Reubenstein 89], [Fickas 87], que é uma técnica na qual estratégias e heurísticas de Inteligência Artificial são usadas, pretende-se criticar os requisitos cotejando-os contra um domínio previamente codificado. Portanto a formação desse domínio seria baseado em fatos de sistemas similares que já tenham sido elicitados. O uso dessa técnica permitiria que ferramentas do tipo assistentes inteligentes fornecessem críticas sobre os requisitos preparados pelo engenheiro de software. Dada a disponibilidade de um domínio, é possível determinar `fatos errados` e *fatos faltantes*. No caso do uso da técnica de reutilização de domínios, a computação Delta é controlada por fatos errados e fatos faltantes:

$$\Delta = \begin{cases} \textbf{fatos errados}(\text{fatos do domínio, fatos}) \\ \textbf{fatos faltantes}(\text{fatos do domínio, fatos}) \end{cases}$$

O uso de domínios possibilita que seja feita uma diferenciação entre problemas de correteza e completeza, facilidade não presente em muitas das estratégias de análise. O problema com o uso de domínios é, não só o seu alto custo, como também a complexidade de sua construção [Arango 89].

4 Validação

A validação de software, ou seja a confirmação de que o produto é aquele desejado pelo usuário, ocorre, normalmente, no fim do ciclo de vida. O teste do sistema, como é comumente conhecida esta validação, é o teste integrado dos programas do sistema pelo usuário. No nosso caso a validação é feita no próprio processo de elicitação de requisitos, uma seja uma validação anterior a própia especificação.

Várias estratégias de validação de software tem sido propostas pela literatura. Se por hipótese, definirmos que algumas dessas estratégias também podem ser aplicadas a elicitação de requisitos, teremos um conjunto, do qual se sobressaem as seguintes estratégias:

- Comprovação informal,
- Prototipagem e
- Usando Pontos de Vista.

Na comprovação informal a validação ou a revisão dos requisitos é basicamente uma tarefa de leitura de descrições em linguagem natural e do uso dos clientes para detetar problemas na expressão dos requisitos. As estratégias para a comprovação informal são várias, mas elas tem em comum, a falta de um apoio automatizado, e a excessiva dependência das abilidades analíticas dos leitores (vide Parte III, Participação Ativa do Usuário). A computação Delta é controlada por um lista de erros.

$$\Delta = \textbf{lista de erros}(\text{informação, fatos})$$

Em prototipação vários tipos de protótipos tem sido propostos como um meio de obter uma retroalimentação do universo de informações. Alguns deles usam linguagens de alto nível (linguagens do tipo geradores de aplicação), enquanto outros usam linguagens de especificação executáveis. A idéia básica é que pela prototipação é possível validar os requisitos/especificação contra as expectativas do usuário. A computação Delta, neste caso, é controlada pelo comportamento dos fatos:

$$\Delta = \textbf{comportamento dos fatos}(\text{expectativas do usuário, fatos})$$

Na Análise de Pontos de vista temos a computação Delta controlada por três tipos de problemas: inconsistências, fatos errados, fatos faltantes.

$$\Delta = \begin{cases} \textbf{inconsistências}(fatos_a, fatos_b) \\ \textbf{fatos errados}(fatos_a, fatos_b) \\ \textbf{fatos faltantes}(fatos_a, fatos_b) \end{cases}$$

5 Aspectos Gerais

Dos métodos e técnicas de verificação/validação detalharemos Inspeções e a Análise de Pontos de Vista. Inspeções tem se mostrado um método extramamente positivo e a Análise de Pontos de Vista é um método novo em que a automação desempenha um papel importante.

Consideramos Inspeções como um método de verificação apesar do papel importante que desempenham os atores em cada um de seus papéis na aplicação do metodo. Como não se confronta o documento com a realidade, mas sim com critérios e perguntas já pre-estabelecidas e que objetivam fundamentalmente buscar defeitos, rotula-se Inspeções de verificação.

A Análise de Pontos de Vista como meio para a validação de requisitos, oferece uma alternativa ao uso de domínios, já que pode diferenciar problemas de correteza e completeza e não depende da construção de um domínio. Esta capacidade de diferenciar entre correteza e completeza possibilita atacar um ponto crucial do desenvolvimento de software, isto é, será que estamos construindo o software desejado, ou não? Atacando o problema da validação logo no ínicio do processo de construção estamos evitando ocorrer em possíveis erros de projeto, que mais tarde serão, não só de difícil correção, como de alto custo. Classificamos a Análise de Pontos de Vista como validação apesar de que o trabalho de comparação entre visões ser automatizado, visto que o enfoque principal é na comparação de retratos do UdI (Figura 6).

6 Inspeções

O método de Inspeções pode ser aplicado na definição de requisitos para a verificação em documentos de requisitos, quer eles sejam produzidos por clientes e sem uma estrutura própria ou produzidos por engenheiros de software, sendo neste caso uma lista de requisitos. Inspeções é um método gerencial de reuniões que objetivam o descobrimento de defeitos em documentos. Há definições claras sobre os papéis que cada membro da equipe tem que desempenhar e que resultados deve produzir. A produtividade do uso de Inspeções na revisão de desenhos e de

código, segundo vários relatos da indústria, tem sido excelentes, isto é consegue-se efetivamente descobrir defeitos antes de que o sistema seja testado.

O objetivo básico de Inspeções é verificar se o produto produzido por um processo de produção está de acordo com os critérios de sucesso do processo. O método estabelece um processo de seis passos a saber:

- planejamento,
- visão geral,
- preparação,
- inspeção,
- conserto e
- acompanhamento

O planejamento define os participantes, verifica se o produto satisfaz os críterios de entrada, produz uma agenda para a reunião e marca o local. A visão geral cuida da colocação do contexto para todos os membros da equipe. A preparação constitue na leitura dos documentos a serem verificados. A inspeção é propriamente o processo de identificar erros nos documentos utilizando as listas de perguntas chave. O conserto objetiva eliminar os defeitos e o acompanhamento procura assegurar que esses defeitos foram realmente eliminados. Nota-se que a primeiro e o último passo são efetivamente feitos fora da reunião.

Os papéis previstos por Fagan são: moderador, autor, codificador e testador. Veja que esses papéis estão muito orientados a especificação e codificação. No entanto para documentos como requisitos o codificador e é aquele profissional que utilizará o documento analisado como entrada para o seu trabalho. Portanto em requisitos o codificador será na verdade o especificador. Cabe ao testador verificar aspectos relacionados a testabilidade do produto.

Uma variação do método de Fagan [Martin 90] é a utilização de mais de um time no processo de Inspeções (Figura 7). Estes times teriam um único moderador e trabalhariam em paralelo. A vantagem dessa estratégia é que se aumenta sensívelmente o número de erros detectados. Na verdade essa estratégia combina o método de Inspeções com a idéia de pontos de vista.

7 Pontos de Vista

Na tarefa de modelar as expectativas dos usuários dentro de um universo de informações, o engenheiro de software pode encontrar, e usualmente encontra, opiniões diferentes sobre o problema em questão. Diferentes engenheiros de software, quando modelando as expectativas de usuários num `mesmo` universo de informações, produzem diferentes modelos. O mesmo engenheiro de software, quando modelando o mesmo universo de informações pode fazê-lo usando diferentes perspectivas (por exemplo, um modelo de dados x um modelo de processos). Tudo isso é conhecido. O ponto importante é que alguns métodos de engenharia de software usam pontos de vista com o objetivo de produzir um modelo que "melhor" espelhe as expectativas dos usuários no universo de informações. Um exemplo de tal método é CORE [Mullery 79].

O princípio de que mais fontes de informação proporcionam um melhor entendimento de um caso, tem sido usado por séculos em tribunais. Testemunhas diferentes podem ter lembranças diferentes ou complementares. Usando o mesmo princípio no processo de elicitação, as possibilidades de detectar problemas de correteza e completeza serão maiores. No entanto, para se lucrar com este princípio é necessário comparar e analizar diferentes pontos de vista.

A análise e comparação de pontos de vista como proposto por Ross (SADT) e Mullery (CORE) são tarefas informais. Por serem tarefas informais dependem basicamente de um "bom" engenheiro de software. Apesar de advocarem o uso de pontos de vista, nem CORE nem SADT têm um modelo estruturado que possa realmente tirar proveito desse enfoque. A falta de uma representação própria torna difícil a existência de procedimentos para comparação e análise de visões oriundas de diferentes pontos de vista.

7.1 Resolução de Pontos de Vista e seus Problemas

Resolução de pontos de vista é entendido como um processo composto de quatros subprocessos: identificação, classificação, avaliação e integração (Figura 8). De acordo com nossa definição de elicitação, identificação e classificação fazem parte da validação de fatos, enquanto avaliação e integração fazem parte da comunicação.

A resolução de pontos de vista é utilizada no processo de validação dos fatos. A aquisição desses fatos supõe uma orientação baseada na busca de palavras chaves da aplicação e numa representação própria. Portanto, um dos problemas básicos abordados pela nossa pesquisa é: como ter uma representação que possibilite a aplicação de um modelo de resolução de pontos de vista.

Apesar da validação ser um processo que é intrinsicamente ligado ao processo de construir um modelo, é importante ressaltar que a representação usada no processo de validação por pontos de vista, **não** pretende ser a mesma usada para o modelo final produzido pela definição de requisitos. A resolução de pontos de vista e a representação aqui proposta têm o objetivo precípuo de ajudar o entendimento do problema, a modelagem é um processo posterior a este entendimento. Portanto a representação usada na resolução de pontos de vista é limitada. Para se modelar o entendimento da fase de definição de requisitos é necessário um esquema de modelagem conceitual bem mais elaborada do que a representação usada na resolução de pontos de vista.

Antes de melhor definirmos o problema convêm estabelecer algumas definições.

Pontos de Vista – Um ponto de vista é uma posição mental usada por uma pessoa quando examinando ou observando um universo de informações. Um ponto de vista é identificado por uma pessoa (e. g., seu nome) e seu papel no universo de informações (e. g., um engenheiro de software, um funcionário, um gerente).

Perspectiva – Uma perspectiva é o conjunto de fatos observados de acordo com um ponto de vista e modelados com um tipo especial de modelagem. Um exemplo de um tipo especial de modelagem, é a modelagem conhecida como modelo de dados. Em nosso método, usamos três tipos: a perspectiva de dado, a perspectiva de ator, e a perspectiva de processo.

Visão – Uma visão é uma integração de perspectivas. Esta integração é obtida por um processo chamado "construção de uma visão".

VWPl – A linguagem de pontos de vista, é a representação usada para que os fatos e seus relacionamentos sejam descritos dentro do esquema de resolução de pontos de vista. Esta linguagem é baseada em sistemas de produção, e usa uma definição de tipos aliada a arvóres de hierarquias.

Hierarquias – Hierarquias são usadas como parte da linguagem de pontos de vista com o objetivo de prover uma extensão semântica aos têrmos definidos na linguagem. Utiliza-se uma hierarquia de especializações (é uma) e uma hierarquia de decomposições (partes de) para aumentar a carga semântica provida pela tipagem das regras de produção.

Nosso trabalho não lida com os problemas de comunicação na resolução de pontos de vista, já que estes envolvem aspectos de negociação. Os problemas relacionados com a negociação entre atores do universo de informações têm sido estudados em autores preocupados com o impacto social dos sistemas de informação [Gerson 86] [Kling 80]. Nossa pesquisa se dedicou aos problemas de identificação de discrepâncias e a classificação dessas discrepâncias. Identificando e classisficando discrepâncias entre diferentes pontos de vista estamos criando uma agenda que pode orientar a avaliação e a integração desses pontos de vista.

7.2 A Estratégia Proposta

A estratégia proposta é composta de um método e uma linguagem, VWPl, de representação de pontos de vista. O método tem procedimentos para a formalização de pontos de vista, bem como procedimentos para a análise desse formalismo. A linguagem é o meio que codifica o formalismo e torna possível sua análise.

A linguagem é derivada de PRISM [Langley 86], uma arquitetura de sistemas de regras. Portanto, nossa estratégia é basicamente um processo que compara duas bases de regras, cada uma representando um ponto de vista diferente.

Conforme observado na Figura 3, o processo de validação de fatos é dependente do processo da coleta de fatos. Supõe-se que os fatos (palavras-chave) estão à nossa disposição antes da aplicação da resolução de pontos de vista.

Em seguida, apresentamos uma sucinta descrição do método para a produção de pontos de vista, uma descrição da linguagem VWPl e uma descrição dos procedimentos que analizam diferentes pontos de vista.

7.3 Método

Na Figura 6 há um descrição geral da estratégia. José e Maria, ambos engenheiros de software, modelam as intenções dos usuários. Ambos usam VWPl para expressar o universo de informações. Eles usam diferentes perspectivas (processo, dado e ator) e diferentes hierarquias (partes-de é-uma) com o objetivo de criar uma visão. Uma vez de posse de críticas, cada analista resolve os conflitos e integra sua percepção final em uma visão. Esta visão é expressa usando a perspectiva processo em conjunto com as hierarquias. Depois disso, os pontos de vista de José e Maria são comparados e analisados.

Portanto, de modo a identificar e classificar discrepâncias entre pontos de vista diferentes, visões devem ser extraídas de cada ponto de vista. Visões são produzidas por um processo chamado construção de visões. A construção de uma visão é baseada no seguinte:

- a disponibilidade de um método de coleta de fatos,

- um engenheiro de software de posse de um determinado ponto de vista, usa perspectivas e hierarquias para modelar um ponto de vista,
- perspectivas e hierarquias são analizadas por um analisador estático, e
- uma visão é um modelo integrando as diferentes perspectivas e hierarquias derivadas de um mesmo ponto de vista.

Com a disponibilidade de duas visões, torna-se possível comparar diferentes pontos de vista.

Conforme observado antes, é de conhecimento geral que engenheiros de software quando modelando o universo de informações, o fazem usando diferentes perspectivas. Um exemplo disso é o uso de modelos de dados e modelos de processos. Nossa pesquisa além desses usuais modelos põe em evidência os atores envolvidos no universo de informações. A idéia de se explicitar os atores é de usar a perspectiva daqueles que são responsávies pelos processos, isto é agentes humanos e agentes físicos.

O objetivo do uso de hierarquias é a de tentar embutir alguma descrição semântica na informação codificada em VWPl. As hierarquias são compostas de relações de especialização entre palavras chaves e de relações de decomposição entre palavras chaves.

Para construir uma visão, o engenheiro de software descreve o problema usando as três perspectivas e duas hierarquias. As perspectivas e hierarquias são comparadas e são produzidas: uma "lista de discrepâncias" e os "tipos de discrepâncias". Uma visão é a integração de perspectivas e de hierarquias conforme um ponto de vista e com o auxílio de uma "agenda", produzida pela análise de perspectivas.

A construção de uma perspectiva é um processo no qual se assume o uso do conceito de `vocabulário da aplicação`, de tal forma que as palavras chave do universo de informações são utilizadas na representação de pontos de vista.

A idéia principal é a de que o engenheiro de software primeiro analisa o problema e anota suas observações usando a perspectiva de ator em VWPl. As observações descritas usando as outras perspectivas são feitas independentemente e em tempos diferentes. Do mesmo modo, as observações relativas às hierarquias são codificadas.

Supõem-se que a comparação dessas perspectivas e hierarquias produzidas por um mesmo engenheiro de software possibilitará com que este produza uma perspectiva (processo) e hierarquias finais de melhor qualidade. Esta perspectiva e hierarquias são então consideraderadas a representação do ponto de vista selecionado.

São as seguintes, as diretrizes para a aquisição de perspectivas e hierarquias.

- A produção das hierarquias "é-uma" e "partes-de" constantes do universo de informações.
- Para cada perspectiva:
  - achar os fatos;
  - expressar os fatos usando as palavras chaves do domínio de aplicação;
  - classificar os fatos em: fatos objetos, fatos ações e fatos agentes; e
  - definir funcionalmente os fatos usando o formalismo de produções.
- Objetos representam tanto os objetos como os estados desses objetos.
- Não há imposições para a produção de hierarquias, o detentor do ponto de vista é quem decide o que deve ou não estar presente nas hierarquias.

- O intuito é a de deixar estas linhas gerais um pouco frouxas de modo a facilitar ao engenheiro de software, a expressão do ponto de vista em questão.

É importante ressaltar o meta uso da estratégia de resolução de pontos de vista. Ela é aplicada na própria construção de um ponto de vista. Isto é, a estratégia para validação de pontos de vista é usada para checar os modelos de perspectiva (dado, processo, ator). A seguir apresentaremos o suporte representacional do método proposto.

7.4 A Linguagem de Pontos de Vista

Embora não seja o objetivo da Análise de Pontos de Vista lidar com a modelagem de requisitos[1], é mister que um esquema representational esteja disponível. No nosso caso utilizamos um modelo conceitual muito simples, composto de:

- três tipos de entidades: objeto, ação e agente;
- atributos;
- hierarquias: é-uma e partes-de;
- relações. As relações são de dois tipos:
  - <entidade, atributo>, as quais constituem fatos, e
  - a <fatos, fatos>, as quais definem funcionalidade.

Este modelo conceitual é instanciado através de uma linguagem especial chamada VWPl. VWPl é uma linguagem derivada de PRISM, uma arquitetura para sistemas de produção. Não sendo uma linguagem de requisitos, sua utilidade é restrita ao processo de validação de fatos.

A idéia básica atrás de VWPl, é a de ter uma estrutura prè-definida para a construção de regras. A imposição de maior restrição no schema usual de lado direito (RHS) e lado esquerdo (LHS) de uma regra, torna possível que tenhamos alguma informação de ordem semântica na estrutura das regras. O enfoque usado em VWPl é semelhante aos usados em *case grammars*, onde as estruturas produzidas pelas regras da gramática correspondem a relações semânticas ao invés de somente relações sintáticas.

No caso de VWPl as relações semânticas são expressas pelo uso do que chamamos restrições de *tipos* e de *classes*. Tipos são os diferentes fatos usados, isto é, fatos `objeto`, fatos `ação` e fatos `agentes`. Classes são os diferentes papéis que cada fato pode ter numa regra. Numa regra um fato pode:

- ser retirado da memória de trabalho (nós chamamos a isto de *classe de entrada*),
- ser adicionado à memória de trabalho (nós chamamos a isto de *classe de saída*), ou
- permanece na memória de trabalho (isto é chamado de *classe invariante*).

[1]Outros autores têm centrado sua atenção neste tema, em particular a linguagem RML [Greenspan 84] é uma dos mais completos modelos conceituais aplicados à modelagem de requisitos.

Um fato é uma relação entre palavras chaves. As palavras chave para expressar fatos em VWPl são checadas contra uma lista de tipos antes de serem analisadas sintaticamente. Em outras palavras, um teste de pertinência semântica, é efetuado nas palavras chaves disponíveis para a descrição de uma visão. Um fato é composto de uma palavra-chave-fato e um atributo-fato. Um exemplo é "(livro =ident-livro =autor =titulo)", onde livro é a palavra-chave-fato, e *ident-livro, autor e titulo* são os atributos-fato.

Para cada perspectiva há uma combinação especial de tipos e de classes. Neste artigo somente usamos a perspectiva de dado e de ator. Para a PERSPECTIVA DE DADO usamos a seguinte estrutura de regras:

- LHS (lado esquerdo) – a *entrada* é um `objeto`, e a *invariante* pode ser um `agente` ou um `objeto`.
- RHS (lado direito) – a *saída* é uma `ação`.

Para a PERSPECTIVA DE ATOR usamos a seguinte estrutura.

- LHS (lado esquerdo) – a *entrada* é uma `agente`, e a *invariante* pode ser um `agente` ou um `objeto`.
- RHS (lado direito) – a *saída* é uma `ação`.

Hierarquias são codificadas como listas. As listas são organizadas pelo tipo de hierarquia (é uma, ou partes-de). Para cada tipo a raiz da hierarquia é a cabeça de uma lista seguida dos ramos daquela hierarquia.

Na Figura 9 damos um exemplo do uso da linguagem VWPl. Este exemplo mostra como uma simples sentença foi codificada de acordo com a perspectiva usada.[2]

7.5 O Analisador Estático

A análise de diferentes perspectivas e de diferentes visões é efetuada por uma série de heurísticas, que fazem uma análise de dois conjuntos de perspectivas ou visões. Conforme já observado esta análise é realmente uma análise entre dois conjuntos de regras (Figura 10).

Comparar dois grupos de regras só faz sentido quando há similaridade entre eles. No nosso caso, existe uma série de fatores que nos levam a esta similaridade; abaixo listamos os mais importantes.

- Os detentores de pontos de vista estão se baseando no mesmo universo de informações.
- O uso de um método em que o uso do conceito de "vocabulário de aplicação" norteia a coleta de fatos.
- O uso de uma linguagem especial que restrige como as regras são expressas.

O analisador estático proposto e implementado em *Scheme* [Abelson 87] tem duas importantes tarefas: achar quais as regras quem tem similaridade entre si, e uma vez que regras dos diferentes conjunto de regras são identificadas como similares, identificar e classificar as discrepâncias existentes entre elas. Regras que não encontram similares são classificadas como

[2]Este exemplo "de brinquedo" tem somente o objetivo de dar uma idéia da linguagem VWPl.

falta de informação. Apesar do analisador ser centrado em uma análise sintática há o uso de descritores de semântica tais como: hierarquias e "case grammars".

Sendo baseado na representação sintática de termos, o analisador utiliza-se de reconhecedores de padrões e reconhecedores parciais. Estes procedimentos de reconhecimento de padrões são aplicados entre fatos de dois conjuntos de regras diferentes, e têm diferentes algorítmos de *score* dependendo na informação semântica disponível sobre tipos e classes em cada fato. No projeto do analisador estático usamos várias idéias emprestadas do trabalho de Inteligência Artificial no campo da analogia [Hall 88].

Hall propõe o seguinte esquema para examinar propostas computacionais para a analogia:

- reconhecimento de um problema analogo,
- elaboração de um mapeamento entre fonte e destino,
- avaliação da analogia elaborada, e
- consolidação da informação gerada enquanto usando a analogia.

Em particular nosso metodo lida com três desses processos (Figura 10), sendo que cada um se carcteriza como um sub-problema da análise de pontos de vista. São os seguintes os problemas.

- **Reconhecimento** – Dado dois conjuntos de regras em VWPl, produzir os pares mais parecidos entre si.
- **Elaboração** – Dado os pares mais parecidos entre si, identificar os pares mais prováveis bem como as regras sem par.
- **Avaliação** – Dado os pares mais prováveis, identificar os fatos como contraditórios, ou como faltantes, ou como inconsistentes.

No processo de construir um mapa entre dois conjuntos de regras, a solução de cada sub-problema diminue o espaço de busca. Nesta redução do espaço de busca, os pares mais prováveis são identificados de tal maneira que uma análise mais detalhada torna-se possível. Para cada par, os fatos são comparados com objetivo de encontrar discrepâncias. A classificação de discrepâncias, isto é, determinar quais as discrepâncias relativas a falta de informação e quais as discrepâncias relativas a informações contraditórias, é feita baseada em scores e nas hierarquias. É óbvio que o analisador estático não pode afirmar nada sobre duas regras que não têm discrepâncias, mas que na realidade não estão corretas com respeito ao universo da informação.

Se tomarmos o exemplo dado anteriormente, o analisador estático nos fornecerá as seguintes messagens:

```
Fatos Contraditorios:

  (20 (1020 (((30 limpar-o-deck =numero-de-registro)
              30 limpar-o-deck-com-vassoura
                  =tipo =numero-de-registro . 0.5))))

memsagem 20: "as regras 10 e 20 tem
         diferentes fatos para representar
```

```
        uma acao considerada similar"

   (1 (1020 1 ((oficial) oficial) (=patente =nome) =nome)))

mensagem 1: "as regras 10 e 20 tem atributos diferentes para o mesmo
             agente ''officer''"

       (4 (1020 (((22 iate =numero-de-registro) 22 barco
                                                =numero-de-registro) . 0.66)))

mensagem 4:  "de acordo com a hierarquia
         e-uma os respectivos objetos
         estao em contradicao"

Fatos Faltantes:

        (16 (1020((vassoura) .21)))

mensagem 16:  "o objeto ''vassoura''
         esta faltando na regra 10"
```

7.6 Comparação com Outras Estratégias de Validação

Como vimos na Seção 2, várias estratégias de validação podem ser usadas na fase da definição de requisitos. Usando o mesmo esquema utilizado anteriormente, podemos distinguir a resolução de pontos de vista como sendo capaz de distinguir entre inconsistências, fatos errados e fatos faltantes.

Se compararmos com as estratégias já vistas, podemos apontar as seguintes vantagens (+) e disvantagens (-) da resolução de pontos de vista.

(+) automação.

(+) capacidade de diferenciar entre inconsistências, fatos errados e fatos faltantes.

(+) independente de análise de domínios.

(-) depende na qualidade dos pontos de vista.

(-) o custo da redundância.

O ponto importante dessa estratégia é a qualidade da entrada na computação do Delta. Uma vez possuindo um conjunto organizado de críticas classificadas segundo a correteza, a completeza e a inconsistências, as chances de expor comhecimento implícito são melhores, porque há uma menor dependência em observadores e leitores.

8 Resumo

Nesta Parte falamos sobre a tarefa de ANALISAR no contexto da Engenharia de Requisitos. Apontamos para que o termo análise é erradamente utilizado na literatura, visto que acreditamos so ser possível analisar após a modelagem. Desta maneira apresentamos as três partes da análise. Mostramos a importância da identificação de partes, não só para dominar a complexidade da tarefa, mas como também por ajudar a identificação das fontes de informação

ligadas as partes. Apresentamos nossa visão sobre a diferença entre validação e verificação e detalhamos duas estratégias de análise.

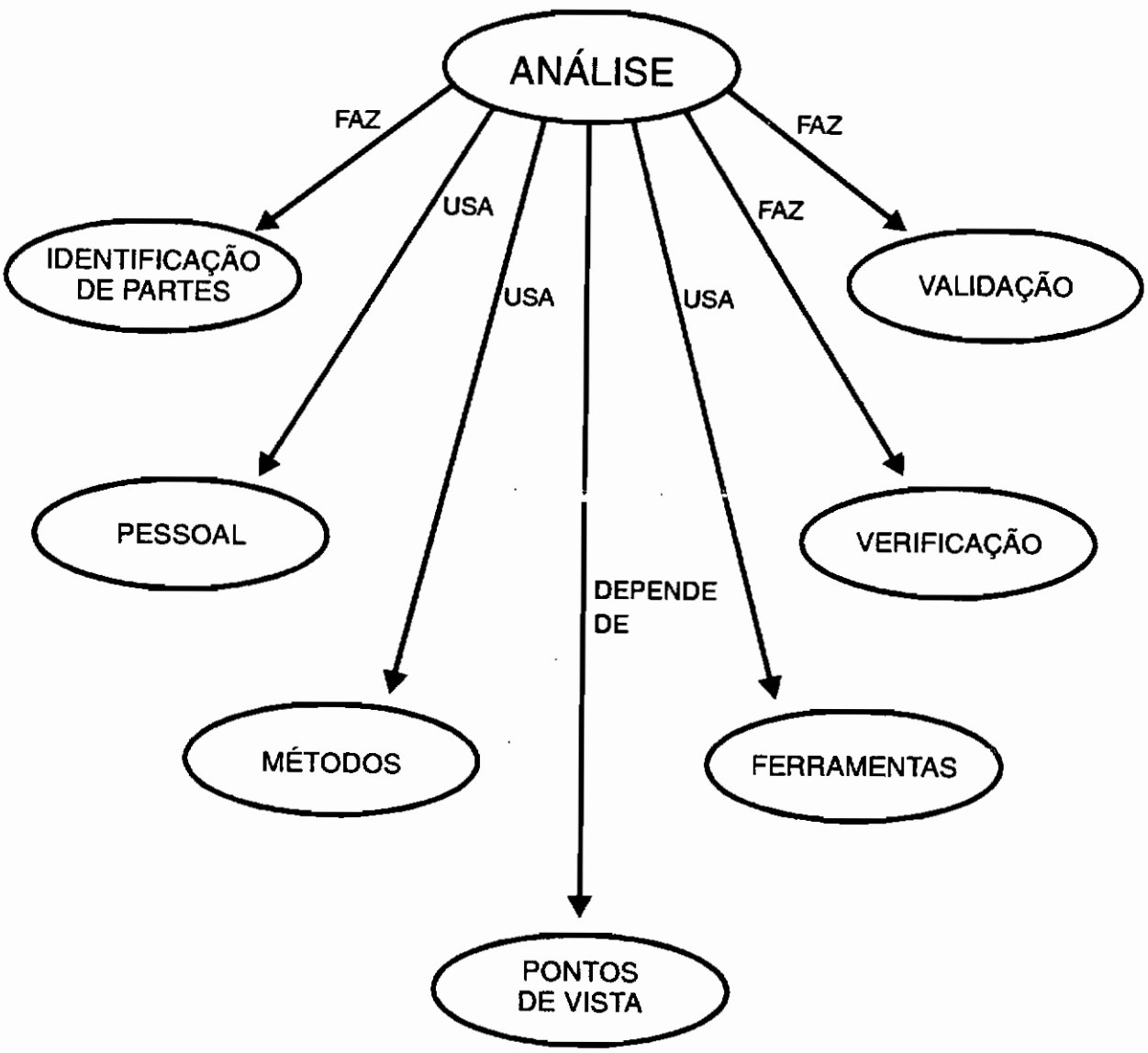


Figure 1: Análise

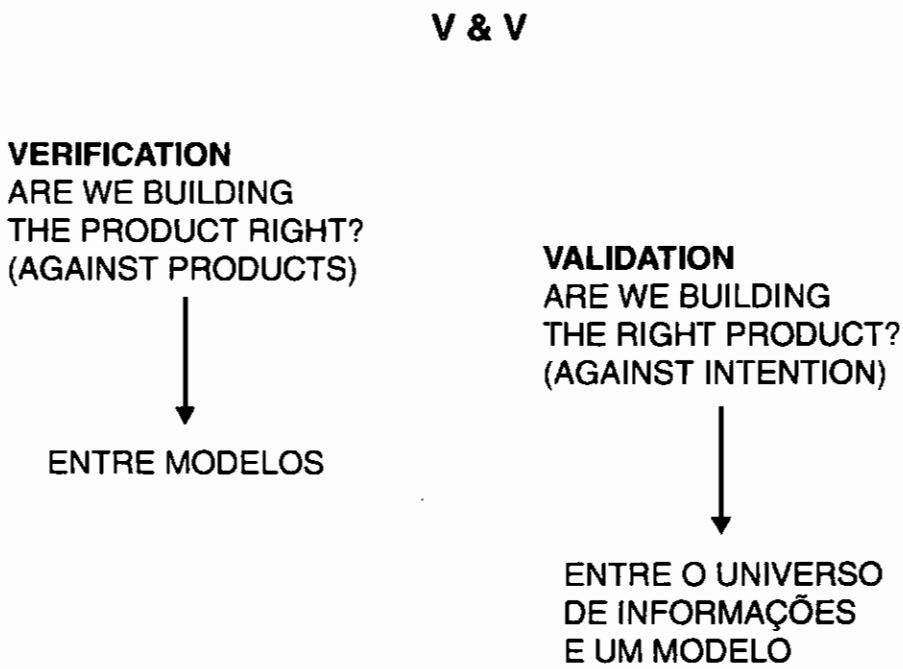
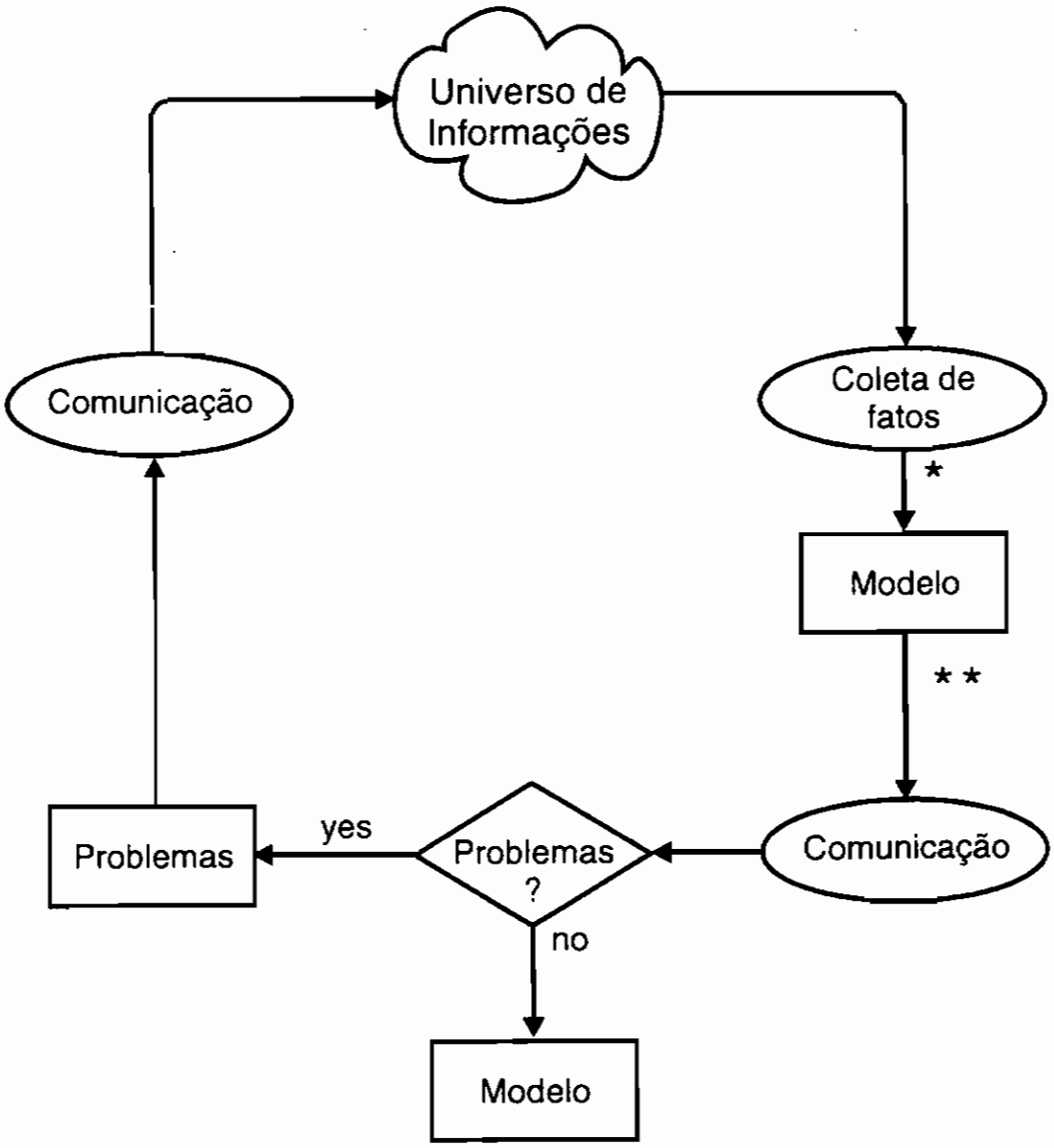

V & V
VERIFICATION
ARE WE BUILDING
THE PRODUCT RIGHT?
(AGAINST PRODUCTS)
ENTRE MODELOS
VALIDATION
ARE WE BUILDING
THE RIGHT PRODUCT?
(AGAINST INTENTION)
ENTRE O UNIVERSO
DE INFORMAÇÕES
E UM MODELO

Figure 2: V & V



* de coleta de fatos para modelo, existe processo de modelagem.
** identificação de partes

Figure 3: Loop da Análise

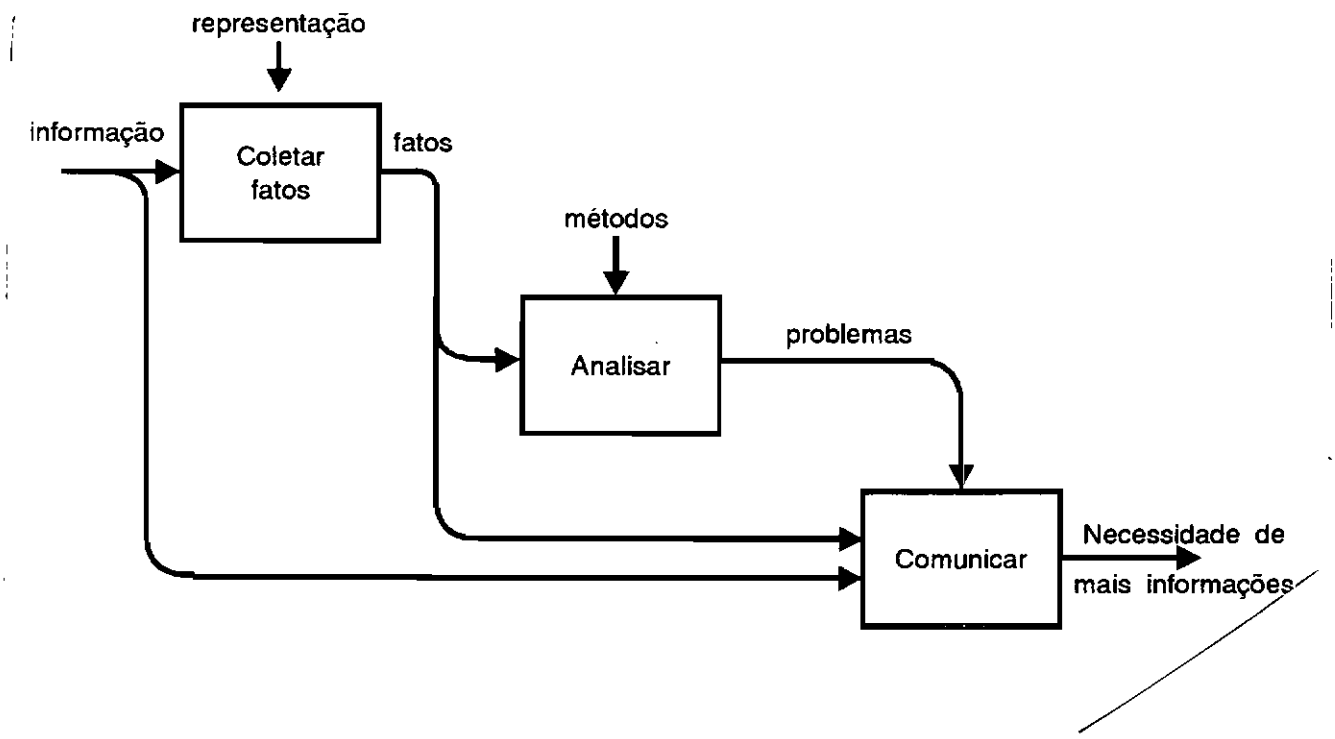


Figure 4: Computação Delta

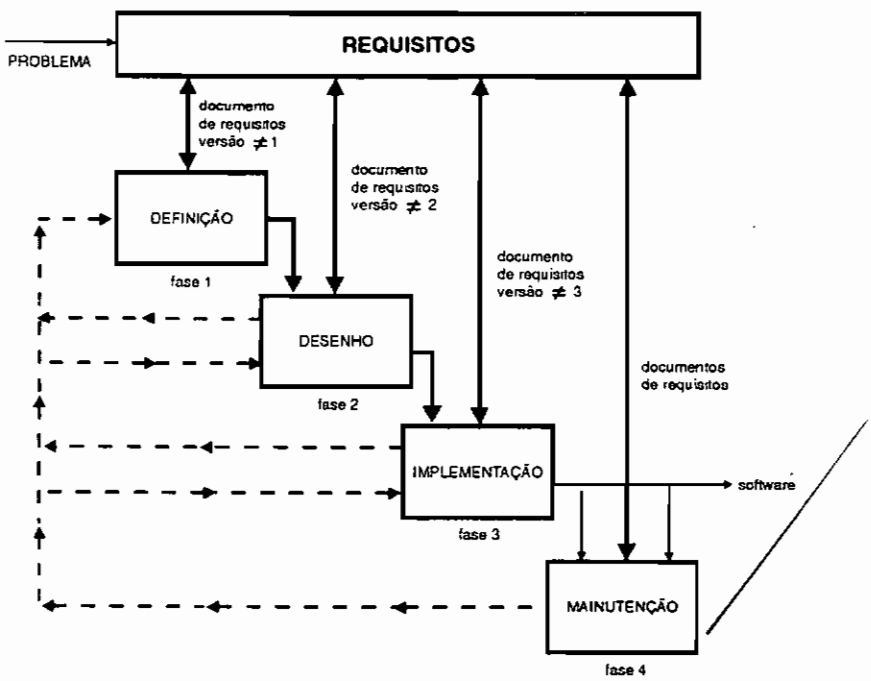


Figure 5: Ligando as Partes ao UdI

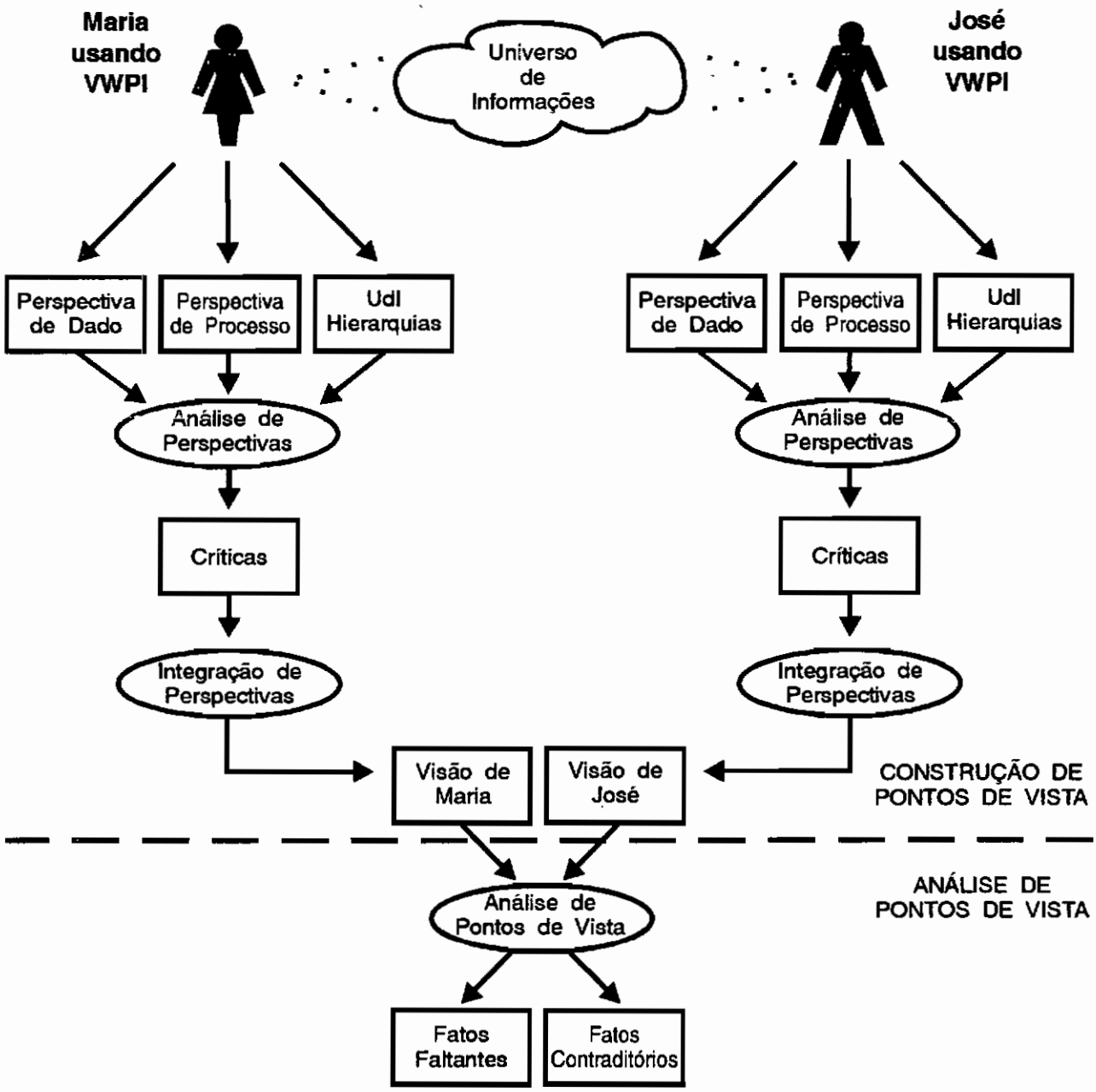


Figure 6: Análise de Pontos de Vista

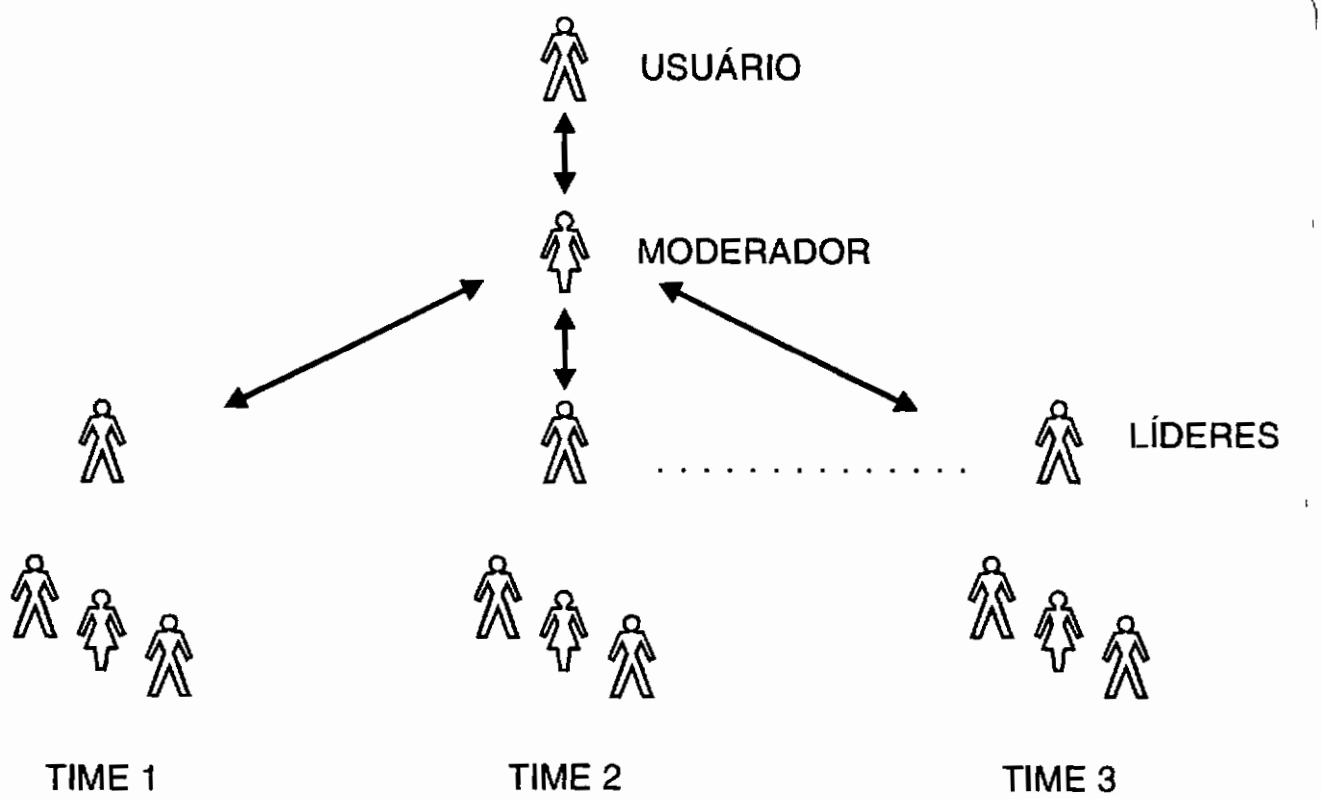


Figure 7: Inspeções N-Fold

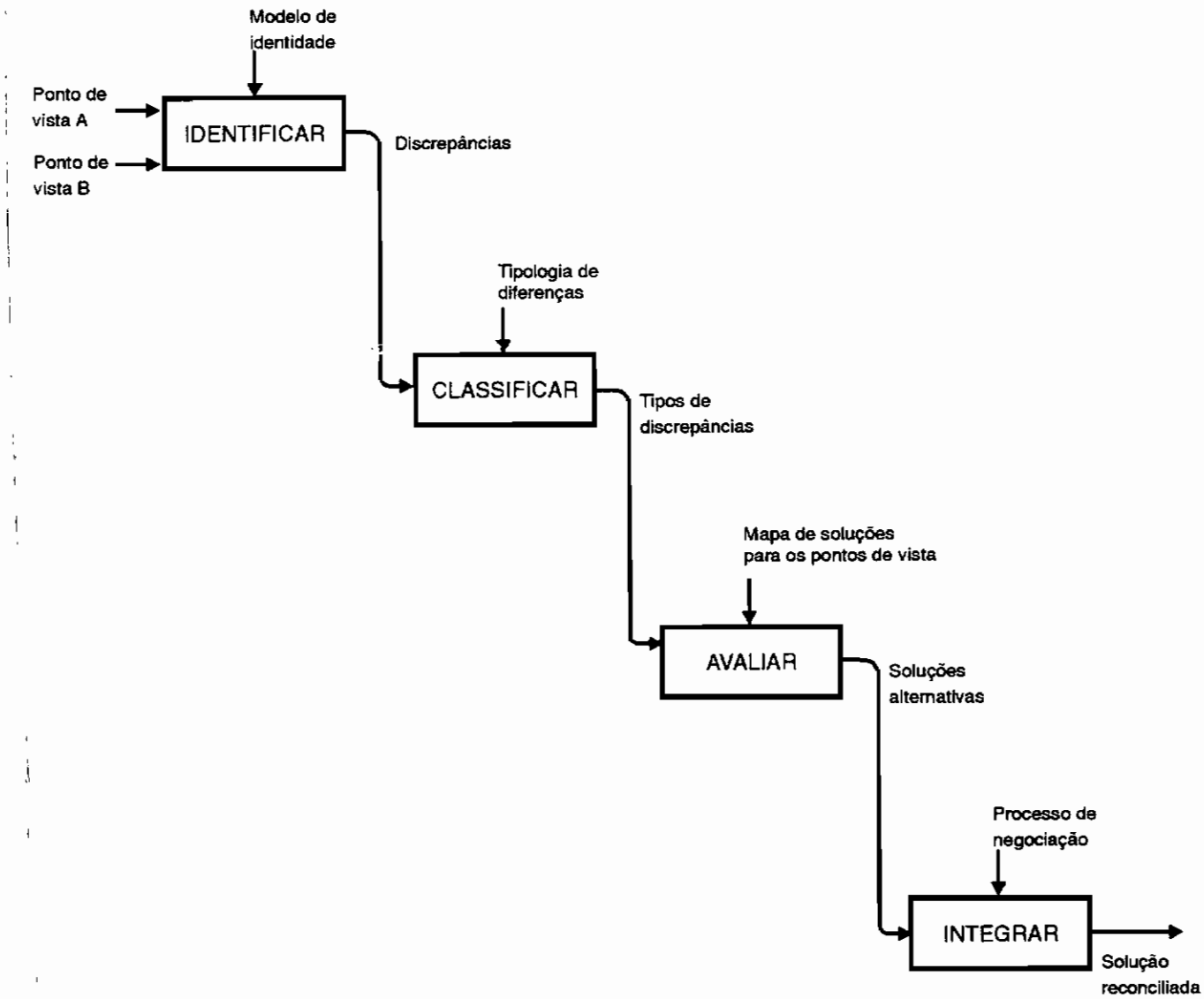


Figure 8: Sub-Processos da Resolução de Pontos de Vista

```
        "Um oficial do navio tem, como um de seus deveres,
         a tarefa de limpar o deck do navio, e ele
         pode usar diferentes tipos de vassouras"

REGRAS:
                    Perspectiva de Ator

     (10 ((oficial =patente =nome) (iate =numero-de-registro))
         -->
         (($delete-from wm (oficial =patente =nome))
          ($add-to wm (limpar-o-deck =numero-de-registro))))

; O agente ''oficial'' (entrada) executa a acao ''limpar-o-deck'' (saida)
; no objeto ''iate'' (invariante).

                    Perspectiva de Dado

      (20 ((oficial =nome) (barco =numero-de-registro)
           (vassoura =tipo))
          -->
          (($delete-from wm (vassoura =tipo))
           ($add-to wm (limpar-o-deck-com-vassoura =tipo =numero-de-registro))))

; O objeto ''vassoura'' (entrada) e usado pelo agente ''oficial''
; (invariante) para  executar a acao ''limpar-o-deck-com-vassoura'' (saida)
; no objeto ''barco'' (invariante)

                    Hierarquias
       (is-a (2 (navio iate barco)))
       (parts-of (2 (navio deck)))
```

Figure 9: Exemplo

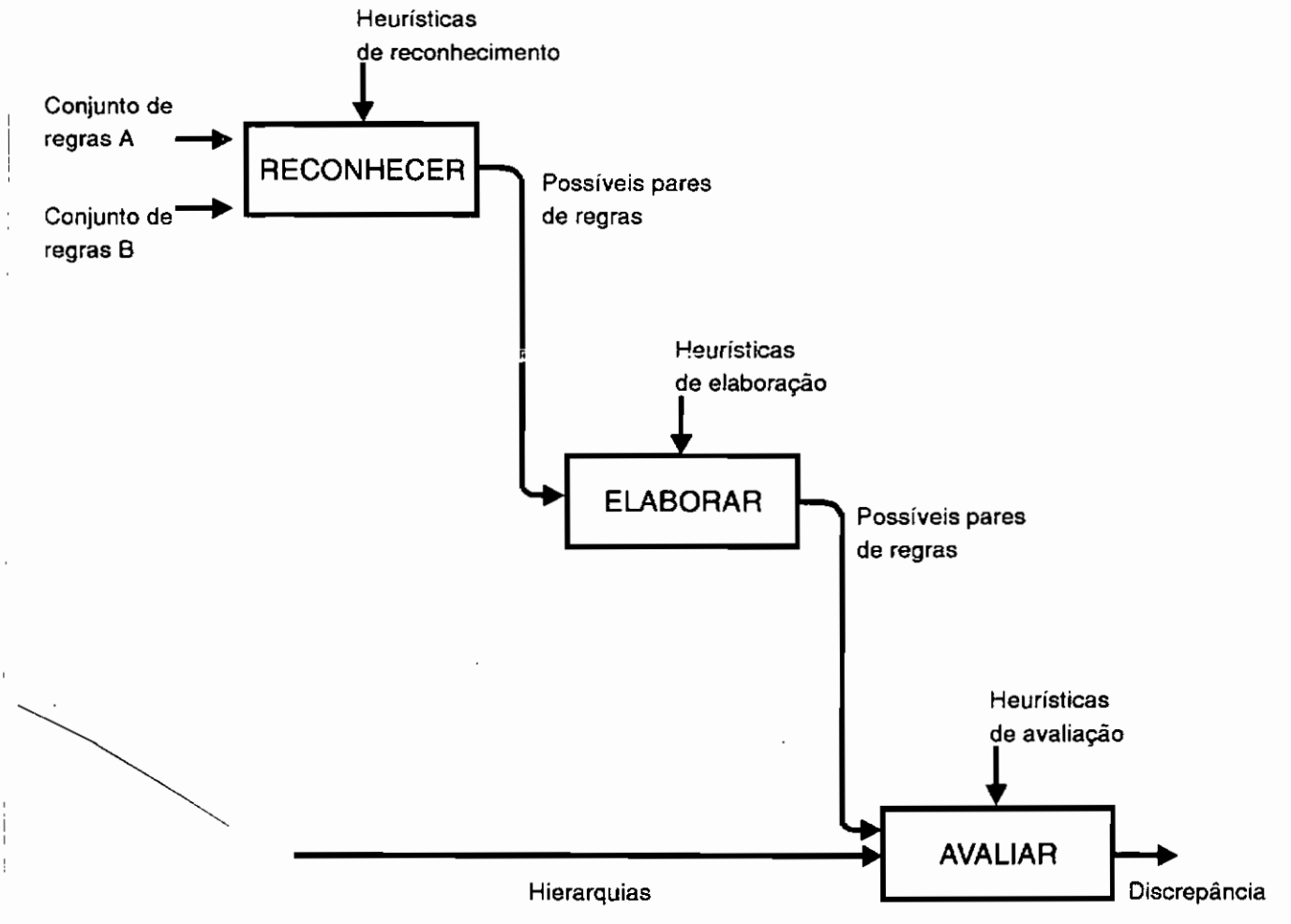


Figure 10: Comparação de Pares de Regras

References

[Abelson 85] Abelson, H., Sussman, G., and Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1985.

[Arango 89] Arango, G. Domain Analysis: From Art Form to Engineering Discipline. *In 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, 1989), IEEE Computer Society Press, pp. 152-159.

[Fagan 86] Fagan, M.E., Advances in Software Inspections, *IEEE Trans. on Software Engineering,* vol. 12, n. 7, Jul., 1986, pp. 744-751.

[Fickas 87] Fickas, S. Automating the Analysis Process: An Example. *In 4th International Workshop on Software Specification and Design* (Monterey, CA, 1987), IEEE Computer Society Press, pp. 58-67.

[Gerson 86] Gerson,E. and Star, S. Analyzing Due Process in the Workplace. *ACM Trans. on Office Inf. systems 4,* 3 (Jul.1986), 257-270.

[Greenspan 84] Greenspan, S. *Requirements Modelling: A Knowledge Representation Approach to Software Requirements Definition.* PhD thesis, Computer Research Group, University of Toronto, Mar. 1984.

[Hall 88] Hall, R. Computational Approaches to Analogical Reasoning: A Comparative Analysis. *Artificial Intelligence 21,* 1 (Jan. 1988), 241-250.

[Kling 80] Kling, R. Social Analyses of Computing: Theorectical Perspective in Recent Empirical Research. *Computer Surveys,* pp.61-110, March, 1980.

[Langley 86] S. Ohlsson and P. Langley; *PRISM Tutorial and Manual.* Tech. Rep. 86-02, University of California, Irvine - Dept. of Computer Science, Feb. 1986.

[Martin 90] Martin, J. and Tsai, W.T. N-fold Inspection: A requirements analysis technique, *Communications of the ACM,* v. 33, n. 2, Feb, 1990.

[Mullery 79] Mullery, G. CORE - A Method for Controlled Requirement Specification. In *Proc. 4th Int. Conf. on Softw. Eng.* (1979), IEEE Computer Society Press, pp. 126-135.

[Reubenstein 89] Reubenstein, H. and Waters, R. The Requirements Apprentice: An Initial Scenario. *In 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, 1989), IEEE Computer Society Press, pp.