INTEGRAÇÃO DE CONHECIMENTO EM UM AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE

Ricardo de Almeida Falbo

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

Aprovada por:

Profa. Ana Regina Cavalcanti da Rocha, D.Sc.

Prof. Crediné Silva de Menezes, D.Sc.

Profa. Cláudia Maria Lima Werner, D.Sc.

Prof. Daltro José Nunes, D.Sc.

Prof. Júlio César S. do Prado Leite, Ph.D.

Prof. Paulo César Masiero, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

DEZEMBRO DE 1998

FALBO, RICARDO DE ALMEIDA

Integração de Conhecimento em um Ambiente de Desenvolvimento de Software [Rio de Janeiro] 1998

X, 215 p., 29,7 cm (COPPE/UFRJ, D.Sc., Engenharia de Sistemas e Computação, 1998)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Ambientes de Desenvolvimento de Software

2. Integração de Conhecimento

3. Ontologias

I. COPPE/UFRJ II. Título (série)

"Se muito vale o já feito

Mais vale o que será."

(Milton Nascimento / Fernando Brant)

A Jô, por simplesmente tudo!

A minha mãe Wilma, por todo apoio e carinho;

A meu irmão Miguel, pela guarida e incentivo;

A meu pai Paschoal, pelo apoio;

A minha irmã Giselle, pelo incentivo.

Agradecimentos

A minha "professora", Ana Regina, por ter acreditado em minha capacidade e pelo prazer de sua orientação;

A meu "cacique", Crediné, pelas infindáveis discussões e pelos ensinamentos de como "voar" e ir mais longe;

A D. Nelsan, Sr. Florentino, Rita e Mario, pelo grande incentivo e apoio nos pequenos detalhes da vida;

Aos professores da COPPE, e em particular ao professor Guilherme Travassos, pelos ensinamentos;

A minha "irmã ontológica", Káthia, pelas discussões e grande troca de idéias;

Às irmãs Ana Paula e Cláudia, pelo suporte administrativo durante esta jornada;

Aos colegas do Departamento de Informática da UFES, pelo incentivo e apoio constantes;

Aos colegas de curso, Rosa, Maxim, Marco Antônio, Marlon, entre tantos outros, pelo incentivo e apoio;

A todos aqueles que, de forma direta ou indireta, colaboraram para a conclusão deste trabalho;

À CAPES, pelo apoio financeiro.

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

INTEGRAÇÃO DE CONHECIMENTO EM UM AMBIENTE DE DESENVOLVIMENTO DE SOFTWARE

Ricardo de Almeida Falbo

Dezembro/1998

Orientadores: Ana Regina Cavalcanti da Rocha
Crediné Silva de Menezes

Programa: Engenharia de Sistemas e Computação

A integração em Ambientes de Desenvolvimento de Software (ADSs) têm sido considerada uma questão de três dimensões: dados, controle e apresentação. Entretanto, à medida que cresce o número de ferramentas baseadas em conhecimento em um ADS, uma quarta dimensão passa a ter de ser considerada: a integração de conhecimento. Este trabalho defende o uso de Servidores de Conhecimento para promover a integração de conhecimento em um ADS.

Um Servidor de Conhecimento é uma infra-estrutura de conhecimento que torna disponíveis componentes de conhecimento sobre um universo de discurso, para serem reutilizados e compartilhados entre ferramentas. Sua arquitetura é projetada com base em ontologias e modelos de tarefa.

Um Servidor de Conhecimento de Processo foi definido no contexto do Projeto TABA da COPPE/UFRJ e um protótipo construído. Para estudar a integração de conhecimento na Estação TABA, foi desenvolvido um assistente inteligente para apoiar a definição de processos de software, utilizando o Servidor de Conhecimento definido.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

KNOWLEDGE INTEGRATION IN A SOFTWARE ENGINEERING ENVIRONMENT

Ricardo de Almeida Falbo

December/1998

Advisors: Ana Regina Cavalcanti da Rocha
Crediné Silva de Menezes

Departament: Engenharia de Sistemas e Computação

One recurring problem in Software Engineering Environments (SEEs) is the integration of tools. Three dimensions have been considered in this context: data, control and presentation. However, as knowledge-based support in SEEs grows, another dimension must be considered: knowledge integration. In this work we advocate the use of Knowledge Servers to promote knowledge integration in SEEs.

Knowledge Servers can improve integration by offering domain knowledge components to be reused and shared among tools. The Knowledge Server architecture is designed based on domain ontologies and task models.

A Software Process Knowledge Server has been defined in the context of TABA Project of COPPE/UFRJ and a prototype developed. To check the knowledge integration in the TABA Workstation, we developed an intelligent assistant to support the definition of new software processes. This assistant was built using the knowledge infrastructure offered by the Knowledge Server defined.

ÍNDICE

Capítulo 1 - Introdução **1**

1.1 - Contexto e Objetivo da Tese 2

1.2 - Metodologia 5

1.3 - Contribuições 7

1.4 - Trabalhos Correlatos 7

1.5 - Organização da Tese 9

Capítulo 2 - Suporte Baseado em Conhecimento ao Processo de Software **10**

2.1 - Suporte de Ambientes de Desenvolvimento ao Processo de Software 12

2.2 - Suporte Baseado em Conhecimento ao Processo de Software: O Estado Corrente 17

2.2.1 - Assistentes Inteligentes 17

2.2.2 - ADSs com Suporte Baseado em Conhecimento 22

2.2.3 - Problemas Inerentes à Abordagem Corrente 25

2.3 - Integração de Conhecimento em ADSs 26

2.4 - Conclusões do Capítulo 31

Capítulo 3 - Modelagem de Conhecimento **33**

3.1 - Representação de Conhecimento e seus Níveis 35

3.2 - Abordagens para a Modelagem de Conhecimento 41

3.3 - Ontologias 46

3.3.1 - O Uso de Ontologias no Desenvolvimento de SBCs 49

3.3.2 - Usando Ontologias para Obter Integração no Nível de Conhecimento 53

3.3.3 - A Construção de Ontologias 54

3.3.4 - Métodos para a Construção de Ontologias 59

3.3.5 - Vantagens e Problemas no Uso de Ontologias 63

3.4 - Um Passo à Frente na Construção e Uso de Ontologias 65
3.4.1 - LINGO - LINguagem Gráfica para descrever Ontologias 66
3.4.2 - Engenharia de Ontologias - Uma Abordagem Sistemática 72
3.4.3 - Engenharia de Conhecimento Baseada em Ontologias 79
3.5 - Conclusões do Capítulo 83

Capítulo 4 - Servidores de Conhecimento **86**
4.1 - A Estação TABA 86
4.2 - O Conhecimento da Estação TABA 90
4.3 - Servidores de Conhecimento e a Integração 96
4.4 - Um Servidor de Conhecimento de Processo de Software 100
4.5 - Conclusões do Capítulo 105

Capítulo 5 - Uma Ontologia de Processo de Software **107**
5.1 - Ontologia de Processo de Software: Especificação de Requisitos 108
5.2 - Ontologia de Atividade 114
5.3 - Ontologia de Procedimento 124
5.4 - Ontologia de Recurso 132
5.5 - Ontologia de Processo de Software 139
5.6 - A Instanciação da Ontologia de Processo 145
5.7 - Conclusões do Capítulo 153

Capítulo 6 - O Servidor de Conhecimento de Processo **155**
6.1 - Modelos de Tarefa 155
6.2 - O Protótipo do Servidor de Conhecimento de Processo 162
6.3 - *Assist-Pro*: Assistente Inteligente para a Definição de Processos de Software 168
6.3.1 - Definição de Processos de Software 168
6.3.2 - Usando o Servidor de Conhecimento de Processo na Construção de Assist-Pro 170
6.3.3 - A Implementação de Assist-Pro 179
6.4 - Conclusões do Capítulo 186

Capítulo 7 - Considerações Finais **189**

7.1 - Conclusões 189

7.2 - Perspectivas Futuras 193

Referências Bibliográficas **196**

Anexo A - Modelo Completo da Ontologia de Processo de Software **208**

Anexo B - Modelo de Objetos do Componente Conhecimento **214**

Capítulo 1

Introdução

O desenvolvimento de software é uma atividade em franca expansão. A cada dia, novos domínios de aplicação são incorporados à área de atuação da Engenharia de Software e, neste contexto, a qualidade tem despontado como um fator essencial em projetos de software.

Entretanto, ainda hoje, o desenvolvimento de software segue, muitas vezes, uma abordagem quase artesanal, fortemente baseada em habilidades individuais, o que tem gerado sérios problemas. Um primeiro e importante passo no tratamento desses problemas é considerar a tarefa global de desenvolvimento como um processo que pode ser controlado, medido e melhorado. Pesquisas, como a realizada no âmbito do Projeto Esprit (BAZZANA et al., 1993), evidenciaram a profunda relação entre a qualidade do produto (software) e a qualidade do processo de desenvolvimento.

Na tentativa de se aperfeiçoar o desenvolvimento de software, em busca de um meio mais racional de produção, duas abordagens têm sido enfocadas:

- melhoria do processo através do qual o software é desenvolvido, e,
- uso de tecnologia para apoiar, automatizar e, portanto, em alguma extensão, reduzir o nível de habilidade requerido para o desenvolvimento de software.

A primeira abordagem envolve a definição, execução, acompanhamento e melhoria de processos e está fortemente associada a modelos e padrões para a qualidade do processo. Contudo, à medida que os processos tornam-se mais complexos, o volume de pessoas envolvidas e o número de tarefas a serem realizadas

crescem em grandes proporções. Assim, gerenciar tais processos, sem ferramentas de apoio, torna-se inviável.

O conceito de Ambientes de Desenvolvimento de Software (ADSs) surgiu com o objetivo de prover uma infra-estrutura para apoiar o engenheiro de software na construção, avaliação e gerência do desenvolvimento de produtos de software, através da combinação de técnicas, métodos e ferramentas (TRAVASSOS, 1994). Um ADS, segundo a definição adotada no Projeto TABA, contém (ROCHA et al., 1987):

- um ciclo de vida, definindo as etapas do processo de desenvolvimento e as atividades a serem realizadas em cada etapa,
- um conjunto de métodos, usados para organizar o pensamento e o trabalho do desenvolvedor ao longo do processo de desenvolvimento, e,
- um conjunto de ferramentas que automatizam os métodos.

Ambientes de Desenvolvimento de Software (ADSs) provêem serviços de infra-estrutura, permitindo integrar ferramentas individuais ao longo de três dimensões principais: controle, dados e interface com usuário. Desta forma, o processo de desenvolvimento pode ser conduzido de modo uniforme e consistente. Além disso, dada a complexidade das tarefas realizadas no desenvolvimento de software, é importante que o ADS ofereça algum tipo de suporte baseado em conhecimento para o desenvolvedor. Idealmente, o conhecimento não deve ficar embutido em uma ferramenta, mas, ao contrário, deve estar integrado ao ambiente para que possa ser compartilhado e reutilizado por diversas ferramentas. Assim, é necessário considerar mais uma dimensão na questão da integração em ADSs: a integração de conhecimento (TRAVASSOS, 1994). Este é o tema central desta tese.

1.1 - Contexto e Objetivo da Tese

Desenvolvimento e manutenção de software são atividades que requerem inteligência, conhecimento e disciplina. O acesso ao conhecimento apropriado e o uso de heurísticas e eficientes algoritmos de busca são cenários típicos em todas as fases do processo de desenvolvimento de software. Assim, a aplicação de técnicas de Inteligência Artificial (IA) na resolução de problemas de Engenharia de Software

pode prover soluções úteis e eficientes, especialmente por sua adaptabilidade (KONTOGIANNIS, 1995).

As primeiras iniciativas de aplicar técnicas de IA à Engenharia de Software tinham por objetivo a programação automática. Entretanto, rapidamente percebeu-se que, ao invés de afastar o ser humano do processo de software, a direção a ser tomada deveria ser no sentido de colocar o computador como uma ferramenta dentro deste processo. Com o aumento da complexidade dos processos de desenvolvimento, faz-se necessário oferecer suporte baseado em conhecimento às suas várias atividades. Desta forma, cada vez mais, técnicas de IA têm sido aplicadas na construção de assistentes inteligentes para apoiar os engenheiros de software na realização de suas tarefas. Entretanto, de modo geral, cada uma destas aplicações traz o conhecimento embutido e, portanto, não disponível para o ambiente de desenvolvimento como um todo. O objetivo deste trabalho é propor um modelo de integração de conhecimento para ADSs, tendo como referencial a Estação TABA (ROCHA et al, 1990).

O Projeto TABA (ROCHA et al, 1990), em desenvolvimento no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, desde 1987, visa a construção de uma Estação de Trabalho configurável para apoiar o engenheiro de software no desenvolvimento de produtos de software. A motivação para este projeto está na constatação de que domínios de aplicação e projetos diferentes têm características diferentes e que estas devem incidir nos ambientes de desenvolvimento através dos quais os engenheiros de software desenvolverão suas aplicações. Somente assim, tais ambientes podem efetivamente apoiar o desenvolvimento. Para atingir seus objetivos, a Estação TABA contém um meta-ambiente capaz de especificar o ADS mais adequado a um projeto em um domínio de aplicação, instanciar e tornar operacional o ambiente especificado (TRAVASSOS, 1994).

Inicialmente, a especificação de ambientes na Estação TABA era apoiada por XAMÃ (AGUIAR, 1992), um assistente baseado em conhecimento, que fornecia heurísticas para a escolha de modelos de ciclo de vida, métodos e ferramentas. Estes elementos eram considerados os componentes a partir dos quais um ADS era formado. Entretanto, XAMÃ não tinha a preocupação com a definição de um processo de desenvolvimento de software. Além disso, a questão da integração de

conhecimento não foi considerada e, sendo assim, o conhecimento de XAMÃ não estava disponível para o ambiente TABA como um todo.

Com o reconhecimento da importância de se ter um processo de software para nortear o desenvolvimento e da inexistência de um processo único adequado às particularidades de todo e qualquer desenvolvimento de software, foi desenvolvida uma ferramenta para definição de processos de software na Estação TABA (ARAÚJO, 1998). Esta ferramenta gera uma descrição de processo que serve de entrada para o instanciador de ambientes. Entretanto, tal ferramenta não oferece nenhum tipo de suporte baseado em conhecimento para o engenheiro de software nesta tarefa que, sem dúvida, é bastante complexa e requer uma gama variada de conhecimento. Além disso, outras ferramentas para execução, alteração e acompanhamento de processo foram desenvolvidas (ARAÚJO, 1998). Assim, escolhemos o universo de discurso de processos de desenvolvimento de software para materializar as discussões sobre integração de conhecimento na Estação TABA.

A meta é prover uma infra-estrutura de conhecimento sobre processos de software, o *Servidor de Conhecimento de Processo*, capaz de apoiar a construção de ferramentas baseadas em conhecimento de processo na Estação TABA. Um Servidor de Conhecimento não é uma ferramenta projetada para atuar como um agente resolvedor de problemas a ser usado pelo engenheiro de software em um processo de desenvolvimento. Sua funcionalidade consiste em prover uma infra-estrutura comum para o desenvolvimento de um conjunto de agentes dessa natureza *em um domínio de interesse*, i.e., um Servidor de Conhecimento torna disponível uma infra-estrutura de conhecimento sobre um domínio de interesse.

Para que o conhecimento possa ser integrado ao ambiente, sua modelagem não pode mais ser feita tendo em vista uma única aplicação, mas deve, sim, considerar que o conhecimento será compartilhado por várias aplicações. Assim, o conhecimento deve ser modelado *para reuso* e o ADS deve ser capaz de comportar modelos e bases de conhecimento reutilizáveis, tanto no nível do meta-ambiente, quanto no nível dos ADSs instanciados.

Nossa abordagem defende o uso de ontologias como o aspecto central para se atingir esta meta. Uma *ontologia* é a especificação de uma conceituação (GRUBER,

1995), isto é, uma descrição de conceitos e relações que podem existir para um agente ou uma comunidade de agentes. Ontologias têm por objetivo firmar um acordo sobre o vocabulário do domínio de interesse, a ser compartilhado por agentes que nele atuam. Basicamente, uma ontologia consiste de conceitos e relações, suas definições, propriedades e restrições, descritos na forma de axiomas.

O uso de ontologias tem um forte impacto na aquisição de conhecimento. Enquanto na Engenharia de Conhecimento tradicional, para cada nova aplicação a ser construída, uma nova conceituação é elaborada, em uma abordagem baseada em ontologias, o conhecimento geral do domínio, relevante para uma grande variedade de sistemas, é capturado e especificado na forma de ontologias. Estas, por sua vez, são usadas para guiar a aquisição do conhecimento específico de uma aplicação. Desta forma, uma mesma ontologia pode ser usada para guiar o desenvolvimento de várias aplicações, diluindo os custos da aquisição e permitindo reutilização e compartilhamento de conhecimento.

Usando ontologias para capturar o conhecimento sobre um universo de discurso e tornando este conhecimento disponível, na forma de Servidores de Conhecimento, para o ADS e suas várias ferramentas, é possível integrar conhecimento em ADSs. Basicamente, um Servidor de Conhecimento para um domínio específico deve oferecer:

- um conjunto de bases de conhecimento modulares e reutilizáveis, construídas a partir de ontologias do domínio e,
- um conjunto de máquinas de inferência customizadas para os tipos de problema mais comumente encontrados neste domínio.

Para mostrar como um Servidor de Conhecimento pode apoiar a construção de ferramentas baseadas em conhecimento, utilizamos o Servidor de Conhecimento de Processo para adaptar a ferramenta de definição de processo existente, tornando-a um assistente inteligente.

1.2 - Contribuições

Os ADSs atuais consideram a integração de conhecimento como uma questão de três dimensões: dados, controle e apresentação. Travassos (TRAVASSOS, 1994) identificou a necessidade de se considerar mais uma dimensão: a integração de conhecimento. Entretanto, não há na literatura nenhuma proposta para se integrar conhecimento em ADSs. Assim, a principal contribuição deste trabalho é propor uma abordagem prática para a integração de conhecimento em ADSs. Esta abordagem advoga o uso de Servidores de Conhecimento como agentes de integração de conhecimento em ADSs. Defendemos, ainda, que tais servidores têm de ser apoiados por ontologias para fixar a semântica da informação tratada. De fato, a construção de Servidores de Conhecimento está essencialmente fundamentada no uso de ontologias.

Uma vez que o uso de ontologias é o ponto-chave de nossa proposta para a integração de conhecimento em ADSs e a construção de ontologias é ainda um campo de estudos em aberto, foi proposta também uma abordagem para a construção de ontologias. A ontologia de processo de software, desenvolvida como base para o Servidor de Conhecimento de Processo, foi construída usando esta abordagem.

Além disso, tendo em vista que um Servidor de Conhecimento é, de fato, uma infra-estrutura para apoiar a construção de ferramentas baseadas em conhecimento no ADS, faz-se necessário estabelecer como utilizar esta infra-estrutura no desenvolvimento de assistentes inteligentes. Assim, uma outra contribuição deste trabalho é a proposta de uma abordagem de Engenharia de Conhecimento, a ser utilizada sempre que houver uma infra-estrutura de componentes reutilizáveis de conhecimento, como são os Servidores de Conhecimento. Esta abordagem foi aplicada no desenvolvimento do assistente para definição de processo, utilizando o Servidor de Conhecimento de Processo.

1.3 - Trabalhos Correlatos

Pesquisadores em Engenharia de Software têm utilizado técnicas de IA no processo de software, geralmente na forma de assistentes inteligentes com o conhecimento totalmente embutido e otimizado para a resolução de um problema específico. Até o presente momento, não se tem dado a devida atenção à integração de conhecimento em ADSs.

No âmbito da Inteligência Artificial, no entanto, existem vários trabalhos explorando o reuso e o compartilhamento de conhecimento, cujos resultados podem ser transpostos para a integração de conhecimento em ADSs.

A idéia de um Servidor de Conhecimento, considerando ontologias e modelos de tarefa, foi explorada inicialmente em (VALENTE, 1995). Valente propôs um conjunto integrado de ferramentas para auxiliar ou parcialmente automatizar três tarefas básicas de um profissional de Direito: modelar um caso ou uma lei, analisar as conseqüências legais de um caso e planejar um curso de ações em resposta a uma reação do Sistema Legal. Nosso trabalho dá um passo à frente: a proposta é tornar disponível o conhecimento sobre um domínio de interesse, sem especificar o tipo de ferramenta a ser construída. Ao invés de propor um conjunto de ferramentas, estamos propondo uma infra-estrutura para auxiliar a construção de ferramentas baseadas em conhecimento.

A abordagem para a construção de ontologias foi elaborada tendo por base outros trabalhos na área, tais como (USCHOLD et al., 1995), (GRÜNINGER et al., 1995) e (VALENTE, 1995), sendo enriquecido com experiências próprias adquiridas com o desenvolvimento de ontologias para este trabalho, produzindo, assim, uma abordagem sistemática para a construção de ontologias.

Em uma abordagem bastante afinada com a nossa proposta de uma abordagem de Engenharia de Conhecimento, NECHES et al. (1991) idealizaram um processo de desenvolvimento de SBCs que começaria pela montagem de componentes reutilizáveis, incluindo porções de bases de conhecimento e resolvedores de problemas de propósito especial. Algum esforço seria necessário para conectar essas partes, contudo, a maior parte seria despendido na modelagem do conhecimento especializado e na construção de resolvedores de problema para as tarefas específicas

do sistema em questão. Entretanto, ainda que em (NECHES et al., 1991) vários aspectos enfocando a reutilização de conhecimento tenham sido discutidos, os autores não propõem um método de Engenharia de Conhecimento baseado nestas idéias. Assim, a abordagem aqui proposta é um avanço nesta área.

1.4 - Organização da Tese

Este trabalho contém, além desta Introdução, mais seis capítulos.

O capítulo 2 - Suporte Baseado em Conhecimento ao Processo de Software - discute como um ADS pode apoiar um processo de software, enfatizando o suporte baseado em conhecimento. Este capítulo procura, ainda, dar uma visão geral de como a Inteligência Artificial tem sido aplicada no apoio a processos de software, apresentando sucintamente alguns assistentes inteligentes e ADSs com suporte baseado em conhecimento. O enfoque principal, no entanto, recai sobre a questão da integração de conhecimento em ADSs.

O capítulo 3 - Modelagem de Conhecimento - explora algumas facetas da modelagem de conhecimento, procurando identificar elementos diretamente relacionados com o reuso e o compartilhamento de conhecimento, já que estas questões são cruciais para a integração de conhecimento em ADSs. São discutidos aspectos de representação de conhecimento e abordagens para a modelagem de conhecimento. Além disso, são definidas uma abordagem sistemática para a construção de ontologias e uma abordagem de Engenharia de Conhecimento a ser aplicada no desenvolvimento de SBCs apoiado por Servidores de Conhecimento.

No capítulo 4 - Servidores de Conhecimento - a ênfase recai sobre os aspectos arquiteturais de um Servidor de Conhecimento, procurando mostrar como esta infra-estrutura é capaz de promover a integração de conhecimento em ADSs.

O capítulo 5 - Uma Ontologia de Processo de Software - discute a construção da ontologia de processo de desenvolvimento de software em todas as suas fases. Esta ontologia é um dos elementos principais do Servidor de Conhecimento de Processo, uma vez que é parte integrante de sua base de conhecimento, além de fixar o vocabulário a ser utilizado em outros módulos desta mesma base.

No capítulo 6 - O Servidor de Conhecimento de Processo - apresenta o protótipo de um Servidor de Conhecimento de Processo (SCP) construído para a Estação TABA. Uma vez que a arquitetura geral dos Servidores de Conhecimento contempla tanto conhecimento de domínio, dado pelas ontologias e suas instanciações, quanto conhecimento de tarefa, este capítulo discute, também, a construção de modelos de tarefa para problemas típicos encontrados no desenvolvimento de software. Nesta etapa, a biblioteca de modelos de tarefa de CommonKADS (VALENTE et al., 1994) foi utilizada como base. Modelos genéricos para problemas de planejamento e designação foram selecionados e adaptados para o universo do desenvolvimento de software. Finalmente, para mostrar como Servidores de Conhecimento podem facilitar a construção de ferramentas baseadas em conhecimento no universo de discurso suportado, um assistente inteligente para apoiar a definição de processos de software foi construído usando os componentes disponibilizados pelo Servidor de Conhecimento de Processo, seguindo a abordagem de Engenharia de Conhecimento descrita no capítulo 4.

O capítulo 7 - Conclusões e Perspectivas Futuras - contém as conclusões deste trabalho, evidenciando suas contribuições e perspectivas de futuros trabalhos.

Capítulo 2

Suporte Baseado em Conhecimento ao Processo de Software

O aumento da demanda por sistemas de software, associado à importância do papel por estes desempenhado na sociedade atual, tem levado a uma preocupação constante com a produtividade no desenvolvimento e a qualidade dos produtos gerados.

Após duas décadas de promessas, não cumpridas, sobre ganhos de qualidade e produtividade através da aplicação de novas tecnologias e métodos de engenharia de software, profissionais de software têm se dado conta de que um de seus principais problemas é a falta de habilidade para gerenciar o processo de desenvolvimento (PAULK et al., 1993).

Um processo de software pode ser visto como o conjunto de atividades, métodos, práticas e transformações que guiam pessoas na produção de software. Um processo eficaz deve, claramente, considerar as relações entre as atividades, as ferramentas e os procedimentos necessários e a habilidade, o treinamento e a motivação do pessoal envolvido (OSTERWEIL, 1987) (OSTERWEIL, 1997).

À medida que os sistemas de software crescem em complexidade, o processo de desenvolvimento também torna-se mais complexo e passa a ser imprescindível a utilização de ferramentas automatizadas para apoiar a realização das várias tarefas do desenvolvimento, de modo que este se dê com a qualidade e a produtividade desejadas. Inicialmente, os esforços da comunidade de Engenharia de Software

concentraram-se na produção de ferramentas individuais, o que resultou na produção de um grande número de ferramentas CASE (*Computer Aided Software Engineering*). De modo geral, no entanto, tais ferramentas não foram projetadas para se trabalhar cooperativamente e apoiavam apenas algumas fases do processo de desenvolvimento de software.

Uma vez que o uso de ferramentas isoladas podia oferecer apenas soluções parciais, surgiu o conceito de Ambiente de Desenvolvimento de Software (ADS), buscando combinar técnicas, métodos e ferramentas para apoiar desenvolvedores na construção de produtos de software. ADSs provêem serviços de infra-estrutura, permitindo integrar ferramentas individuais ao longo de três dimensões principais - controle, dados e apresentação - garantindo um tratamento uniforme do produto ao longo de seu ciclo de vida. Desta forma, apoiado por um ADS, o processo de desenvolvimento pode ser conduzido de modo uniforme e consistente.

A necessidade de se aperfeiçoar o processo de software levou a uma aproximação entre as áreas de Engenharia de Software e Inteligência Artificial (IA). Um crescente número de pesquisadores tem usado técnicas de IA, sobretudo técnicas de Sistemas Baseados em Conhecimento (SBCs), para apoiar engenheiros de software na realização de suas tarefas na forma de assistentes inteligentes e ADSs com suporte baseado em conhecimento.

A Inteligência Artificial oferece conceitos apropriados para modelar processos, freqüentemente na forma de agentes executando tarefas para atingir objetivos. Desta forma, abordagens, técnicas e ferramentas de IA podem ser adotadas ou adaptadas na formulação de soluções para problemas no processo de desenvolvimento de software (FICKAS et al., 1994). Formalismos e técnicas para representar conhecimento podem melhorar o processo de engenharia de software através da automatização ou aplicação de soluções heurísticas (KONTOGIANNIS, 1995). Entretanto, com a introdução destes elementos, passa a ser necessário considerar a integração de ferramentas como uma questão de quatro dimensões, incluindo, além da integração de dados, controle e apresentação, a *integração de conhecimento* (TRAVASSOS, 1994).

Este capítulo procura dar uma visão geral de como ADSs podem apoiar o desenvolvimento de software, dando ênfase ao suporte baseado em conhecimento para

as atividades do processo de software. A seção 2.1 discute como as várias atividades do processo podem ser apoiadas por um ADS. Na seção 2.2, o enfoque é direcionado para o suporte baseado em conhecimento ao processo de software, dando uma visão geral do estado atual da aplicação de técnicas de IA a problemas no desenvolvimento de software. São apresentados alguns assistentes inteligentes e ADSs com suporte baseado em conhecimento e são discutidos os problemas oriundos da abordagem corrente. A seção 2.3 discute a importância de se considerar a integração de conhecimento em ADSs. Finalmente, a seção 2.4 apresenta as conclusões deste capítulo.

2.1 - Suporte de Ambientes de Desenvolvimento ao Processo de Software

O processo de desenvolvimento de software não pode ser definido de forma universal. Para ser eficaz e conduzir à construção de produtos de boa qualidade, um processo deve ser adequado ao domínio da aplicação e ao projeto específico. Deste modo, processos devem ser definidos, caso a caso, considerando-se as especificidades da aplicação, a tecnologia a ser adotada na sua construção, a organização onde o produto será desenvolvido e o grupo de desenvolvimento.

Ainda que diferentes projetos requeiram processos com características específicas para contemplar suas peculiaridades, é possível estabelecer um conjunto de atividades básicas que deve ser considerado em qualquer definição de processo de software. Padrões e modelos de processo de software, tais como a ISO 9000-3 (1991), o CMM (PAULK et al., 1993) e o SPICE (DORLING, 1993), têm procurado definir diretrizes para o estabelecimento deste conjunto. Com base na ISO 9000-3, por exemplo, um processo de software deve incluir, entre outras, as seguintes atividades relacionadas à infra-estrutura:

- definição e documentação de objetivos, políticas e compromissos com a qualidade,
- estabelecimento das responsabilidades da gerência,
- estabelecimento, documentação e manutenção de um sistema da qualidade,

- revisões do sistema de qualidade adotado para garantir sua contínua adequabilidade e eficiência,
- auditorias internas do sistema da qualidade,
- estabelecimento, documentação e manutenção de procedimentos para tratar problemas e iniciar ações corretivas e preventivas.

Como atividades relacionadas com o ciclo de vida do software, a ISO 9000-3 enumera:

- definição de um modelo de ciclo de vida para o desenvolvimento,
- revisão de contrato,
- planejamento do desenvolvimento,
- identificação de recursos necessários,
- definição de procedimentos e ferramentas apropriadas para a construção, gerência e controle da qualidade,
- definição e documentação das entradas e saídas de cada uma das fases do desenvolvimento,
- planejamento da qualidade,
- definição da documentação,
- especificação de requisitos,
- projeto,
- implementação,
- revisões,
- testes e validação,
- aceitação,
- entrega,
- instalação, e,
- manutenção do software.

Como atividades de apoio, são definidas na norma:

- gerência de configuração,
- controle da documentação,

- estabelecimento e manutenção de procedimentos para identificação, coleta, indexação, armazenamento, manutenção e disposição de registros da qualidade,
- medições do produto e do processo,
- controle de produtos desenvolvidos por terceiros, e,
- treinamento.

Um processo que leve em conta todas estas atividades é, certamente, um processo complexo e sua utilização torna-se inviável sem o apoio de ferramentas automatizadas. Desta forma, a automatização do processo deve ser considerada, caso contrário, torna-se inviável gerenciar e executar o grande número de atividades envolvidas.

Uma vez que processos precisam ser definidos caso a caso, em função das peculiaridades de cada projeto, ADSs também devem ser definidos caso a caso. Não existe um ADS que seja adequado a qualquer situação de desenvolvimento e ADSs devem ser instanciados com base nos processos que (semi-)automatizam.

Neste contexto, meta-ambientes assumem um importante papel. Com base nas características de um projeto, um processo é definido. Uma vez definido o processo, o meta-ambiente instancia um ADS apropriado para o desenvolvimento em questão. Assim, consideramos dois tipos de suporte: o suporte oferecido por um meta-ADS para a definição de um processo adequado ao projeto e a instanciação do ADS correspondente, e o suporte oferecido por um ADS instanciado a um projeto específico.

Suporte de Meta-ADSs

A definição de processos pode ser bastante complexa e, portanto, é fundamental que o meta-ADS ofereça suporte a esta tarefa. Basicamente, este suporte pode ser oferecido de duas formas: através de bibliotecas de componentes de processo e por meio de suporte baseado em conhecimento. As atividades a serem apoiadas são:

- *definição ou seleção do modelo de ciclo de vida apropriado:* o suporte a esta atividade pode ser oferecido através de uma biblioteca de modelos de ciclo de vida e suporte baseado em conhecimento, auxiliando o engenheiro de software a selecionar o modelo mais adequado em função de

características do projeto, tais como escopo, magnitude, complexidade, conhecimento da equipe sobre o domínio da aplicação, prazos, disponibilidade financeira e tecnologia a ser utilizada no desenvolvimento;

- *definição de procedimentos e ferramentas apropriadas para a construção, gerência e controle da qualidade:* um conjunto de facilidades e ferramentas para apoiar o uso de procedimentos e métodos deve ser disponibilizado, cabendo ao meta-ADS apoiar a seleção do melhor conjunto para um projeto específico;
- *definição da documentação:* o meta-ADS pode auxiliar na definição da documentação necessária, tornando disponível conhecimento e experiências em projetos anteriores, na forma de roteiros e normas para a documentação.

Suporte de ADSs Instanciados às Atividades do Ciclo de Vida:

Tomando por base as atividades de um processo definido em conformidade com a ISO 9000-3, um ADS pode apoiar diversas atividades, entre elas, as enumeradas a seguir:

- *planejamento do desenvolvimento e da qualidade*: o ADS pode oferecer suporte baseado em conhecimento, roteiros e exemplos de planos anteriores em projetos similares para auxiliar a elaboração dos Planos do Projeto e da Qualidade. Com base na natureza das diversas atividades do processo, o ADS pode apoiar a identificação dos recursos necessários, a definição das interdependências entre as várias atividades e a definição das entradas e saídas de cada uma das fases do desenvolvimento. Com base no Plano do Projeto, o ADS pode apoiar a identificação dos marcos e pontos de controle, sugerindo o tipo de avaliação a ser realizada;
- *implementação e execução do Plano do Projeto*: o ADS deve prover orientação para a condução do processo de construção, envolvendo as atividades de análise de requisitos, projeto, implementação, integração, teste e instalação do software, podendo, inclusive, impor alguma seqüência de realização. Deve, ainda, oferecer o suporte de ferramentas adequadas para estas atividades;

- *manutenção do software:* a manutenção, de fato, envolve um novo processo em pequena escala e, deste modo, um novo ADS deve ser instanciado. Vários tipos de suporte podem ser oferecidos, tais como, o registro de solicitações e o auxílio de ferramentas para facilitar a detecção dos erros, gerência da configuração e registro de resultados.

Suporte de ADSs Instanciados às Atividades de Apoio:

Finalmente, no tocante às atividades de apoio enumeradas pela ISO 9000-3, um ADS instanciado pode apoiar as seguintes atividades:

- *monitoramento e controle do progresso e da qualidade dos produtos:* o ADS pode auxiliar no acompanhamento dos custos, prazos e funcionalidades, através da visibilidade do andamento do desenvolvimento, mostrando prontamente o estado corrente ou estados prévios do processo e identificando problemas no cumprimento de compromissos. Além disso, pode também auxiliar e orientar a coleta e análise de dados de métricas, permitindo coleta automática e consistente de dados relacionados a ferramentas e registro em uma base de dados central corporativa. Deve, ainda, orientar e apoiar a execução consistente das atividades do sistema da qualidade;
- *gerência de configuração:* o ADS deve identificar os artefatos de software durante as várias fases, registrar responsabilidades, atividades e ferramentas que manipulam estes artefatos. A documentação gerada no desenvolvimento pode ser controlada e disponibilizada para reuso em outros projetos;
- *controle de produtos desenvolvidos por terceiros:* o ADS pode registrar prazos, requisitos e planos de testes, manter controle de configuração e auxiliar nos testes de aceitação;
- *treinamento:* um ADS pode auxiliar no treinamento, provendo tutores inteligentes, assistentes e orientação on-line para as diversas atividades;

Avaliando os vários tipos de suporte que um ADS pode oferecer ao processo de desenvolvimento de software, é possível notar que muitas das atividades anteriormente relacionadas podem ser apoiadas através de suporte baseado em

conhecimento. Esta abordagem, de fato, tem sido cada vez mais utilizada, como veremos na seção que se segue.

2.2 - Suporte Baseado em Conhecimento ao Processo de Software: O Estado Corrente

O desenvolvimento de software é uma atividade de conhecimento intenso e, portanto, precisa evoluir de sua abordagem mais tradicional, centrada em bases de dados, para uma abordagem centrada em bases de conhecimento (BAILOR, 1992). Atualmente, abordagens baseadas em conhecimento têm sido utilizadas para:

- formalizar os produtos do desenvolvimento de software e as atividades de Engenharia de Software que os produzem;
- empregar técnicas de representação de conhecimento para registrar, organizar e recuperar o conhecimento embutido em decisões de projeto que resultam em sistemas de software bem-sucedidos;
- produzir assistentes baseados em conhecimento para controlar o processo de desenvolvimento de software e para auxiliar o engenheiro de software na realização de suas tarefas.

Idealmente, a meta de uma abordagem baseada em conhecimento para apoiar processos de software deve ser prover ADSs baseados em conhecimento, capazes de assistir desenvolvedores e gerentes de projeto no cumprimento de suas tarefas ao longo de todo o ciclo de vida. No entanto, as pesquisas atuais têm se concentrado basicamente no desenvolvimento de assistentes inteligentes e de facilidades baseadas em conhecimento em ADSs para atividades específicas, como veremos a seguir.

2.2.1 - Assistentes Inteligentes

Assistentes inteligentes são ferramentas que oferecem suporte baseado em conhecimento para atividades do processo de software. Várias dessas atividades são complexas e podem ser mais facilmente realizadas se algum tipo de suporte inteligente for oferecido.

É grande o número de assistentes inteligentes descritos na literatura, cobrindo um amplo espectro de atividades do processo de desenvolvimento, tais como reutilização de software, análise de requisitos, projeto de sistema e gerência de riscos, entre outros.

DRUMMOND et al. (1993), por exemplo, utilizaram regras de produção em uma ferramenta para navegação ativa de bibliotecas de componentes reutilizáveis, onde a meta de busca é inferida a partir de uma seqüência de movimentos feita pelo usuário no curso normal da navegação. A partir da meta inferida, o sistema sugere itens específicos da biblioteca.

O sistema *AIRS* (*AI-based Reuse System*) (OSTERTAG et al., 1992) oferece assistência inteligente à busca de componentes, selecionando componentes candidatos em uma biblioteca com base no grau de similaridade entre eles e a descrição do componente desejado, fornecida pelo usuário.

GONZÁLEZ et al. (1997) utilizaram uma abordagem baseada em conhecimento, mais especificamente raciocínio baseado em caso, para apoiar a seleção de componentes em um repositório de componentes orientados a objetos.

BORGO et al. (1997) desenvolveram *OntoSeek*, uma ferramenta para recuperação de informações baseada em ontologia. Esta ferramenta utiliza *Sensus* (SWARTOUT et al., 1997), uma ontologia lingüística, para realizar o casamento entre consultas e dados na recuperação de componentes orientados a objeto. *OntoSeek* possui uma linguagem de representação simples, mas semanticamente rigorosa, que possibilita uma checagem semântica guiada pela ontologia.

FISCHER et al. (1992) desenvolveram um ambiente integrado de projeto baseado em conhecimento, cuja arquitetura suporta três processos básicos: localização, compreensão e modificação de objetos de projeto. O ambiente é composto de três ferramentas: *CATALOGEXPLORER*, *EXPLAINER* e *MODIFIER*.

CATALOGEXPLORER provê suporte à localização de exemplos relevantes de programas, inferindo a tarefa em mãos com base em especificações apenas parciais. Tais exemplos são apresentados em várias perspectivas, incluindo código, diagramas, figuras e texto. *EXPLAINER* é responsável pela apresentação destas perspectivas, guiando e orientando o projetista na compreensão do objeto recuperado. *MODIFIER*,

por sua vez, provê suporte à modificação do objeto de projeto, visando adaptá-lo às necessidades do projeto corrente.

FREITAS et al. (1998) desenvolveram um assistente de projeto para auxiliar um projetista a diagnosticar e verificar possíveis alterações de projeto, com o intuito de melhorar a qualidade do mesmo. Nesta ferramenta, regras heurísticas são usadas para orientar os trabalhos de modelagem e projeto orientados a objetos.

BOLCER (1995) desenvolveu um assistente inteligente para apoiar o projeto de interfaces gráficas com o usuário. A ferramenta *UIDA (User Interface Design Assistant)* trata problemas de estilo e consistência para a integração de interfaces ao longo do processo de desenvolvimento e auxilia na avaliação de interfaces gráficas de um sistema, com base em regras e exemplos de projetos específicos.

O sistema *UIDA* utiliza regras para satisfazer princípios de projeto, sendo que sua base de conhecimento contém representações de decisões do usuário, do histórico de aplicação das regras, e de objetos de interface e seus atributos, entre outras. A base de regras utiliza este conhecimento para iniciar ações e tomar decisões inteligentes sobre o projeto de interfaces com o usuário.

MADACHY (1995) propôs uma extensão ao COCOMO (*Constructive Cost Model*) (BOEHM, 1981) para auxiliar o planejamento de projetos, através da identificação, classificação, quantificação e estabelecimento de prioridades de riscos do projeto. COCOMO é um modelo de custos amplamente utilizado, que incorpora o uso de "direcionadores de custo" para ajustar cálculos de esforço. Na extensão proposta por Madachy, estes direcionadores de custo são usados para a avaliação de riscos. Além de computar, a partir das entradas dos usuários, os resultados intermediários de COCOMO, riscos são identificados, anomalias nas entradas do usuário detectadas e é provida assistência. Este método está encapsulado na ferramenta *ExpertCOCOMO*.

Assim como MADACHY (1995), TOTH (1995) desenvolveu uma ferramenta para prover assistência especializada para as atividades de gerência de riscos, *STRA (Software Technology Risk Advisor)*. *STRA* provê assistência para a identificação, estabelecimento de prioridades e redução de riscos. Enquanto *ExpertCOCOMO* utiliza conhecimento de situações de risco, tendo por base fatores de custo para identificar e

quantificar riscos, *STRA* utiliza uma base de conhecimento de produtos de software e necessidades de processo, satisfazendo fatores de capacidade e maturidade[1]. Desta forma, *STRA* enfoca riscos técnicos de produto, enquanto *ExpertCOCOMO* enfoca riscos de custos e prazos.

SHEPPERD et al. (1997) desenvolveram *ANGEL* (*ANaloGy Estimation Tool*), uma ferramenta para suportar estimativas de esforço em um projeto de software usando analogia. Dados de projetos concluídos são armazenados na forma de casos e um problema de estimativa passa a ser tratado como o problema de encontrar casos similares àquele para o qual se quer realizar uma estimativa.

FINNIE et al. (1997) concluíram, a partir de um estudo comparativo usando pontos-por-função como uma medida do tamanho de um sistema, que modelos de inteligência artificial usando raciocínio baseado em caso e redes neurais são capazes de prover modelos de estimativa bastante efetivos.

SELBY et al. (1991) especificaram um sistema de medição e análise empírica de resultados, denominado *Amadeus*, que permite a engenheiros de software integrar medição e mecanismos de re-alimentação empíricos aos processos de desenvolvimento e manutenção de software. Seu objetivo é facilitar o acesso aos níveis mais altos do CMM (níveis 4 e 5), através da integração da medição ao processo.

GRESSE et al. (1997) descreveram tipos de conhecimento passíveis de reuso, estratégias para reuso, e tipos de suporte baseado em conhecimento necessários para permitir um planejamento efetivo e eficiente de programas de medição baseados no paradigma GQM (*Goal-Question-Metric*).

LIU (1998) utilizou lógica *fuzzy* para estabelecer prioridades para requisitos de qualidade de software. Em sua abordagem, requisitos de qualidade de software são classificados em dois grupos, um representando requisitos de qualidade que têm de ser satisfeitos obrigatoriamente, outro representando requisitos de qualidade que podem ser satisfeitos em um determinado grau. Uma vez que, muitas vezes, requisitos de qualidade são conflitantes, uma abordagem de lógica *fuzzy* foi utilizada para estabelecer prioridades dentro do segundo grupo.

---

[1] fatores de maturidade identificam a maturidade relativa de uma tecnologia de software no momento atual.

IBRAHIM et al. (1997) ofereceram assistência inteligente para a porção de análise do processo de prototipagem, onde os requisitos de um sistema são firmados e/ou alterados para satisfazer às necessidades reais dos usuários. Regras são utilizadas no suporte à decisão de quais alterações nos requisitos considerar, tendo por base as reações dos usuários ao comportamento demonstrado por um protótipo.

COLOR-X CASE (BURG et al., 1997) é uma ferramenta CASE inteligente para a Engenharia de Requisitos. O objetivo desta ferramenta é permitir a criação de modelos mutuamente consistentes e que correspondam à intenção dos requisitos, como descritos pelos usuários em um documento de requisitos. Dentre seus principais componentes, COLOR-X CASE possui: um repositório com informações sobre os modelos criados; um léxico contendo informações léxicas sobre os conceitos usados nos modelos; uma biblioteca de componentes reutilizáveis; e um sistema de base de conhecimento que contém objetos ativos que, juntos, simulam o comportamento do sistema que se está modelando.

No contexto do projeto ComProLab (FRANCH et al., 1997), foram definidas linguagens e ferramentas para Programação com Componentes. A Programação com Componentes é uma estratégia para desenvolver sistemas de software como uma combinação de componentes de software individuais. Dentre as ferramentas desenvolvidas, há um catálogo de tarefas relevantes para o desenvolvimento de sistemas usando Programação com Componentes, e um assistente de modelo de processo. Este assistente faz uso de um conjunto de regras de precedência para apoiar a definição de um modelo de processo de software, a partir de combinações válidas das tarefas catalogadas. A cada instante, o assistente permite apenas a execução daquelas tarefas que não violam as regras de precedência.

HURLEY et al. (1997) utilizaram tutores inteligentes para apoiar o treinamento de desenvolvedores em novas metodologias de desenvolvimento de software.

No contexto do projeto RE^2 (CANFORA et al., 1998), foi desenvolvido um ambiente integrado para reengenharia de sistemas escritos em C. A meta é extrair e fazer a reengenharia de módulos reutilizáveis a partir de sistemas existentes, capturando o conhecimento existente nestes módulos.

A integração de ferramentas neste ambiente é obtida através de uma representação única do sistema, que é produzida por ferramentas de análise de código e implementada na forma de uma base de fatos Prolog. Esta base de fatos é compartilhada por diferentes ferramentas de engenharia reversa com o objetivo de abstrair diferentes módulos candidatos ao reuso.

2.2.2 - ADSs com Suporte Baseado em Conhecimento

O uso de técnicas de IA em ADSs, com o intuito de oferecer suporte baseado em conhecimento a atividades do processo de software, tem se tornado cada vez mais uma realidade. Embora esta seja uma tarefa complexa, alguns ambientes têm experimentado esta abordagem, como veremos a seguir.

HyperCASE (CYBULSKY et al., 1992) é uma infra-estrutura para integração de ferramentas CASE, combinando uma interface com o usuário baseada em hipertexto, com um repositório de documentos baseado em conhecimento. O sistema provê um ambiente de engenharia de software visual, integrado e customizável, consistindo de ferramentas livremente acopladas.

A arquitetura de *HyperCASE* é composta por três subsistemas. *HyperEdit* integra ferramentas em uma interface gráfica com o usuário, permitindo a criação, modificação e apresentação de documentos de software. É composto de um gerenciador de interface, um sistema de autoria e um gerenciador de eventos. *HyperBase* é um repositório hipermídia de documentos de software baseado em conhecimento. Organiza uma sofisticada base de conhecimento sobre documentos de software reutilizáveis e seus componentes, além de prover heurísticas para verificar consistência e ajudar na execução e testes. Por fim, *HyperDict* é um dicionário de dados comum a todos os documentos em *HyperCASE*.

Inscape (PERRY et al., 1993) é um ADS baseado em especificação, integrado pelo uso construtivo de especificações formais de interfaces. As dependências semânticas entre os vários componentes em um sistema de software são explicitadas e utilizadas para assistir na construção e evolução de sistemas. Um conjunto de predicados de primeira ordem descreve a semântica dos objetos. Estes predicados

descrevem as propriedades dos objetos de dados e as pré e pós-condições de operação de módulos.

Inscape oferece, ainda, uma máquina de inferência para ser usada por ferramentas internas. *Inquire*, por exemplo, um navegador e mecanismo de busca baseado em predicados, utiliza esta máquina com o propósito de auxiliar o ambiente, ou um usuário, na busca de componentes.

KBSEE (*Knowledge-Based Software Engineering Environment*) (GOMAA et al., 1996) é um ambiente desenvolvido para suportar o desenvolvimento de modelos de domínio e a geração de especificações de sistemas a partir desses modelos. O ambiente consiste de um conjunto integrado de ferramentas de software, incluindo ferramentas comercialmente disponíveis e ferramentas desenvolvidas para o ambiente.

Uma das principais ferramentas de *KBSEE* é a ferramenta de elicitação de requisitos baseada em conhecimento (*Knowledge Based Requirements Elicitation Tool - KBRET*), usada para assistir a geração de especificações de sistemas a partir de modelos de domínio. *KBRET* conduz um diálogo com o engenheiro de software oferecendo características para o sistema em desenvolvimento. O engenheiro de software seleciona as características desejadas e *KBRET* realiza inferências para verificar se elas são consistentes. A partir dessas características, *KBRET* determina tipos de objetos a serem incluídos no sistema em desenvolvimento. Esta informação é usada para adaptar o modelo de domínio e gerar a especificação do sistema.

Os principais componentes de *KBRET* são: (i) uma base de conhecimento dependente de domínio, (ii) uma base de conhecimento independente de domínio e (iii) um gerenciador de interface com o usuário. A base de conhecimento dependente de domínio é derivada a partir do repositório de objetos do ambiente através de um extrator de base de conhecimento dependente de domínio. Esta ferramenta extrai informações contidas no repositório de objetos para modelos de domínio e mapeia-as para a base de conhecimento dependente de domínio, criando fatos para cada tipo de objeto, entre outros. Diferentes bases de conhecimento para diferentes domínios podem ser geradas e, desta forma, o ambiente é configurado para gerar especificações de sistemas nos diferentes domínios. Com esta abordagem, é possível reutilizar a

porção de conhecimento independente de domínio, casando-a com porções dependentes de domínio, dando grande flexibilidade ao ambiente.

MONTERO et al. (1997) propuseram o uso de uma arquitetura de transformação baseada em regras, para suportar a transferência de informação entre ferramentas apoiando diferentes fases do ciclo de vida. A motivação para esta abordagem está no fato da informação ter de fluir por diversas ferramentas ao longo do processo de software. Cada uma destas ferramentas possui seu próprio meta-modelo, tornando difícil a integração. A arquitetura de transformação é responsável pela conversão de informações entre as várias ferramentas e seus meta-modelos, permitindo, assim, o fluxo da informação ao longo do processo.

ZHANG et al. (1997) utilizaram uma abordagem baseada em um framework de linguagens visuais para a construção e integração de ferramentas em um ADS. A motivação para este trabalho advém do fato de ferramentas CASE oferecerem uma grande variedade de linguagens visuais diagramáticas para análise e projeto de sistemas que, em última instância, refletem diferentes meta-modelos. Assim, as ferramentas de VisPro, uma infra-estrutura para construção de ADSs integrados, servem, basicamente, ao mesmo propósito da arquitetura de transformação de (MONTERO et al., 1997). A diferença básica está na forma de resolução do problema da integração, que neste caso é dada através de linguagens visuais. Cada linguagem visual tem sua gramática especificada como uma gramática de grafos sensível ao contexto, usando regras.

DEMIRÖRS (1997) propôs o uso de uma arquitetura *blackboard* como o modelo base de *TeamPro*, um protótipo de um ADS com suporte baseado em conhecimento. A ênfase principal de *TeamPro* é a comunicação entre os membros de uma equipe. A arquitetura *blackboard* provê uma base comum para a integração da equipe, através do conhecimento que ela mantém sobre o sistema e as responsabilidades de cada membro da equipe.

Alguns ADSs Centrados em Processo provêem suporte inteligente à modelagem de processos, entre eles *EPOS* e *Smart*. *EPOS* (JACCHERI et al., 1993) (NGUYEN et al., 1997) possui um planejador inteligente que é responsável pelo detalhamento de tarefas, gerando, automaticamente uma nova rede de subtarefas para

cada tarefa composta. O planejador começa com uma tarefa composta e uma declaração da meta a ser atingida e aplica técnicas de encadeamento progressivo e decomposição hierárquica, juntamente com conhecimento específico do domínio para construir uma rede de subtarefas apropriada.

Smart (GARG et al., 1994), por sua vez, utiliza uma ferramenta baseada em conhecimento, *Articulator*, para modelar, analisar e simular processos organizacionais complexos. *Smart* possui uma biblioteca baseada em conhecimento, que apoia a organização, acesso e reuso de componentes de processo de software. Possui, ainda, uma base de conhecimento sobre os componentes de processo e provê um conjunto de operações que suportam navegação, busca, composição e abstração de modelos de processo.

Smart auxilia a construção de modelos de processo de três formas: primeiro, pode representar e armazenar padrões de desenvolvimento, metodologias e políticas organizacionais na forma de componentes de processo; segundo, funções de busca baseadas em conhecimento podem ser usadas para recuperar componentes necessários, com base nos requisitos do usuário; terceiro e mais importante, operações baseadas em conhecimento podem ser usadas para compor e adaptar os componentes de processo selecionados para construir o modelo de processo desejado.

2.2.3 - Problemas Inerentes à Abordagem Corrente

Ao analisar os sistemas anteriormente apresentados, entre outros (FALBO et al., 1995), percebe-se claramente que o conhecimento aparece sempre embutido em alguma ferramenta ou assistente. É importante destacar que isto ocorre mesmo nos ADSs. Esta tem sido a tônica do suporte baseado em conhecimento ao processo de software: o conhecimento é adquirido e modelado para um propósito específico e embutido em um assistente. Entretanto, esta abordagem apresenta uma série de problemas, entre eles:

1. *Dificuldade de compartilhar e reutilizar conhecimento*: uma vez que o conhecimento é adquirido e modelado para um propósito específico, torna-se difícil separar as porções do conhecimento que representam conhecimento comum a uma gama de aplicações daquelas que representam

conhecimento heurístico específico da aplicação corrente. Conseqüentemente, dificilmente o conhecimento pode ser compartilhado ou reutilizado.

2. *Dificuldade de comunicação entre as ferramentas*: uma vez que cada ferramenta é construída com base em uma conceituação específica e utiliza um vocabulário próprio, torna-se difícil a comunicação entre ferramentas, no que diz respeito ao conhecimento.
3. *Redundância e inconsistência do conhecimento no ADS*: muito do conhecimento descrito em uma ferramenta pode ser comum a várias outras ferramentas do ADS, o que leva à redundância e, talvez, à inconsistência do conhecimento descrito no ADS como um todo.
4. *Baixa produtividade na construção de ferramentas baseadas em conhecimento para o ADS*: sempre que uma nova ferramenta baseada em conhecimento tem de ser construída, uma aquisição de conhecimento é realizada a partir do nada, o que aumenta os custos e diminui a produtividade no desenvolvimento de tais ferramentas.

Estes problemas são análogos aos problemas encontrados quando a integração de dados não é considerada. Assim, esta tendência precisa ser revertida para que possamos construir ambientes verdadeiramente integrados. Somente com a *integração de conhecimento* devidamente considerada, torna-se viável a construção de ambientes plenamente integrados. Sem contemplar esta dimensão da integração, provavelmente, cada ferramenta estará baseada em uma conceituação, utilizará um vocabulário próprio e o conhecimento nela embutido não poderá ser reutilizado ou compartilhado.

2.3 - Integração de Conhecimento em ADSs

Ferramentas CASE isoladas tem se demonstrado inadequadas (VESSEY et al., 1992). Muito tempo é gasto em seu aprendizado e elas não são capazes de garantir a consistência e a qualidade do produto ao longo do desenvolvimento. Faz-se necessário, portanto, integrar ferramentas, garantindo um tratamento uniforme do produto ao longo de seu ciclo de vida. É importante observar, ainda, que integração não é uma propriedade de uma única ferramenta, mas de suas relações com outros

elementos do ambiente (THOMAS et al., 1992). Estes relacionamentos mostram como as ferramentas se integram ao processo de desenvolvimento e à plataforma de execução (TRAVASSOS, 1994).

Uma das principais metas a serem perseguidas em ADSs é exatamente a integração de ferramentas e, desta forma, a arquitetura de um ambiente deve permitir que ferramentas possam cooperar umas com as outras. Ferramentas em ADSs devem suportar algum método de desenvolvimento e, ainda, possibilitar que as informações a elas associadas estejam disponíveis para o ambiente como um todo (TRAVASSOS, 1994).

A maioria dos ADSs trata a integração de ferramentas como uma questão de três dimensões: controle, dados e apresentação (TRAVASSOS, 1994). A integração de apresentação é responsável por proporcionar ao usuário a sensação de integração no ambiente. É através dela que as formas de apresentação da informação e as técnicas de interação com o usuário tornam-se homogêneas. Assim, a dimensão apresentação busca promover a integração no que se refere à interação com o usuário. A integração de controle é responsável por prover serviços e funcionalidades básicas às ferramentas e ao ambiente, permitindo o seu funcionamento de forma organizada. Por fim, a integração de dados estabelece a forma como as ferramentas realizarão o tratamento das informações, através da provisão de serviços básicos de armazenamento e gerenciamento de estruturas de informação.

Entretanto, com o aumento da complexidade dos processos de software, faz-se necessário considerar informações de natureza semântica. Sistemas de informação suportando tarefas e domínios complexos, tipicamente, não possuem o corpo de conhecimento necessário para gerar interpretações adequadas. Ao contrário, assumem que o usuário possui este conhecimento. Entretanto, suporte inteligente implica que esta carga deve ser deslocada de volta ao sistema, tanto quanto possível. Neste contexto, é necessário considerar uma visão mais abrangente de integração, observando, além da integração de dados, controle e apresentação, a *integração de conhecimento* (TRAVASSOS, 1994).

A integração de conhecimento possibilita às ferramentas um melhor entendimento da semântica das informações e dos métodos existentes para tratar a

informação. Este conhecimento, assim como os dados, deve estar disponível para o ambiente, de forma a poder ser compartilhado por todas as ferramentas que dele necessitarem. A integração de conhecimento torna disponíveis os serviços básicos de armazenamento, gerenciamento e utilização do conhecimento descrito e adquirido ao longo do processo de desenvolvimento (TRAVASSOS, 1994).

Em um ADS provendo suporte inteligente a diversas tarefas, as várias ferramentas ou assistentes baseados em conhecimento podem ser vistos como agentes inteligentes. Neste sentido, o ADS torna-se um sistema multi-agente. O principal desafio dos sistemas multi-agentes é a questão de como utilizar, de forma conjunta, conceitos provenientes de diferentes descrições e conceituações (TAKAAI et al., 1997). Quando dois agentes possuindo interpretações diferentes para um mesmo conceito tentam se comunicar, estas diferenças provocam distúrbios na comunicação. Assim, é fundamental que as ferramentas em um ADS compartilhem um vocabulário e uma mesma interpretação para os termos empregados. Esta deve ser a principal meta da integração de conhecimento: fixar a semântica das informações trocadas entre as diversas ferramentas.

Analisando os assistentes e ambientes apresentados nas seções anteriores, fica evidente que a questão da integração de conhecimento tem sido negligenciada. Claramente, estes sistemas têm muito conhecimento em comum. Um assistente para análise de riscos, assim como um assistente para modelagem de processos, tem de ter algum conhecimento sobre atividades, recursos humanos, métodos e ferramentas de software. Idealmente, dentro de um ADS, estas ferramentas deveriam ser capazes de compartilhar este conhecimento. A tabela 2.1 mostra algumas interseções observadas nos sistemas discutidos.

Tabela 2.1 - Tipos de Conhecimento Utilizados em Sistemas Inteligentes de Suporte a Atividades do Processo de Software.

Assistentes para	Conhecimento sobre					
	Atividade	**Artefato**	**Ferramenta**	**Método**	**Recurso Humano**	**Domínio**
Modelagem de Processo	x	x	x	x	x	
Análise de Requisitos	x	x		x	x	x
Projeto	x	x		x	x	x
Projeto de Interface		x	x	x		
Análise de Riscos	x	x	x	x	x	x
Avaliação Qualidade	x	x	x	x	x	

As interseções mostradas na tabela 2.1 não significam que o conhecimento descrito em cada ferramenta, para cada tipo, é o mesmo. Ao contrário, muitas vezes, as perspectivas são bastante diferentes. Nos assistentes para análise de requisitos e projeto, por exemplo, o conhecimento sobre método diz respeito ao(s) método(s) suportado(s) pelas ferramentas, estando descrito com muito mais detalhes do que em um assistente para análise de riscos, onde há informações gerais sobre vários métodos. O conhecimento de atividade utilizado em um assistente de análise de requisitos ou projeto, por sua vez, diz respeito às atividades que devem ser realizadas em uma análise de requisitos, ou projeto, respectivamente. Já em um assistente para modelagem de processo ou para análise de riscos, tem-se um conhecimento mais amplo sobre as atividades do desenvolvimento de maneira geral. Entretanto, ainda que diferentes perspectivas e níveis de detalhes sejam utilizados, há muito conhecimento em comum. No caso do conhecimento sobre atividade, aspectos relacionados à decomposição de atividades e precedência entre elas estão presentes em todos os assistentes que manipulam este tipo de conhecimento.

Entretanto, para compartilhar conhecimento em um ADS, é preciso mudar a forma como são construídas suas ferramentas. Atualmente, o conhecimento em um ambiente é dado pelo conhecimento embutido em cada uma de suas ferramentas e, portanto, há muita redundância e inconsistência. Como mostra a figura 2.1(a), cabe às ferramentas oferecer o suporte baseado em conhecimento ao ambiente. Entretanto, o que se faz necessário é a construção de um modelo de conhecimento para o ambiente e seu uso por cada uma das ferramentas, como mostra a figura 2.1(b). Neste caso, cabe ao ambiente oferecer o suporte baseado em conhecimento para suas ferramentas. Somente com esta abordagem será possível obter ambientes verdadeiramente integrados.

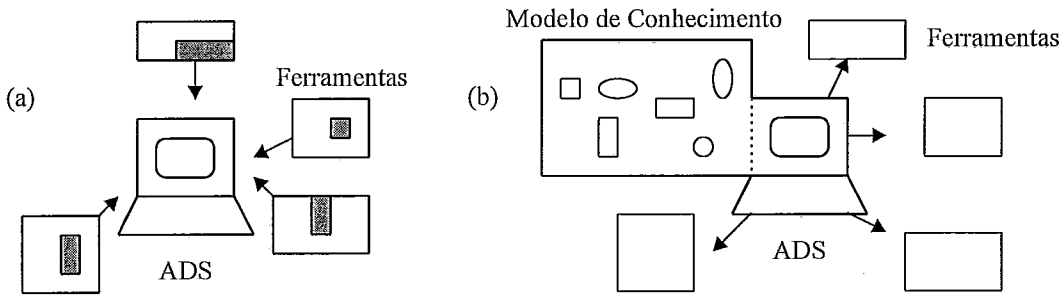


Figura 2.1 - (a) Ambientes Atuais e (b) a Integração de Conhecimento em um ADS.

Para ilustrar esta concepção, consideremos o seguinte cenário típico no contexto de ADSs com suporte baseado em conhecimento.

Cenário: Desenvolvimento de ferramentas inteligentes para um ADS.

Em um ADS, deseja-se oferecer suporte baseado em conhecimento para o acompanhamento de projetos e para a avaliação da qualidade. Utilizando a abordagem dos sistemas atuais (figura 2.1(a)), dois assistentes seriam desenvolvidos embutindo o conhecimento necessário. Certamente, o assistente para acompanhamento de projetos iria manipular conhecimento sobre *atividade*, *recurso* e *produto*. Provavelmente, parte deste conhecimento seria necessário também para o assistente de avaliação da qualidade. Entretanto, não haveria garantia de que este conhecimento seria tratado de forma uniforme nos dois assistentes. Por exemplo, ao invés de empregar os termos *atividade*, *recurso* e *produto*, o assistente de avaliação da qualidade poderia adotar

termos como *tarefa*, *insumo* e *artefato*, criando problemas na troca de informações entre estas ferramentas.

Além disso, sem um modelo de conhecimento fundamentando a construção de bases de conhecimento, perde-se a possibilidade de relacionar conceitos, e as regras de inferência tornam-se elementos isolados, dissociados uns dos outros. Assim, a base de conhecimento passa a ser um conjunto de regras isoladas, em um nível puramente simbólico, como mostra a figura 2.2. Nesta figura, ainda que aparentemente haja uma interseção do conhecimento representado nas ferramentas, não há meios de se capturá-la. Mesmo dentro de uma ferramenta (no exemplo a ferramenta N), as relações, quando existentes, são puramente simbólicas, e não conceituais.

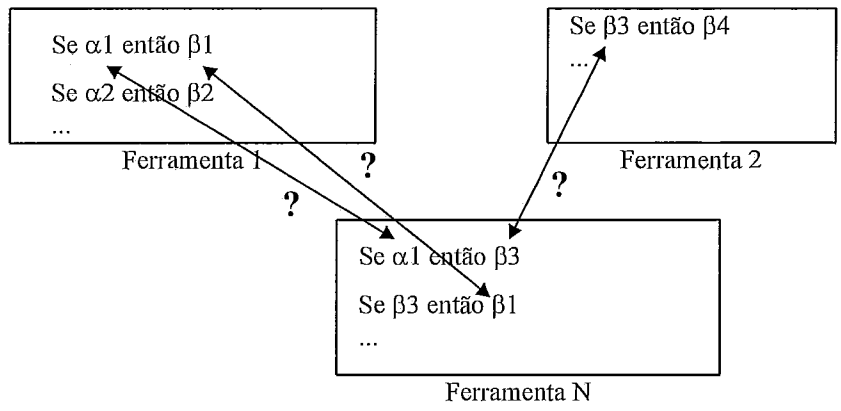


Figura 2.2 - O Conhecimento nos Sistemas Atuais.

Construindo um modelo de conhecimento para o ADS (figura 2.1(b)), estes problemas podem ser minimizados, ou até mesmo eliminados. Um vocabulário básico é estabelecido e, portanto, todas as ferramentas empregam os mesmos termos para referenciar os conceitos correspondentes do mundo real. Uma vez que estes conceitos estão amparados em um modelo de conhecimento global para o ADS, suas relações ganham semântica, deixando de ser puramente simbólicas.

2.4 - Conclusões

Um crescente número de pesquisadores tem usado técnicas de Inteligência Artificial para assistir engenheiros de software na realização de suas tarefas. Neste

capítulo, discutimos como a tecnologia de sistemas baseados em conhecimento tem sido utilizada para oferecer suporte inteligente a atividades do processo de software. Alguns assistentes inteligentes e ADSs com suporte baseado em conhecimento foram brevemente apresentados.

Ao analisar estes sistemas, percebe-se claramente que, mesmo nos ADSs, a abordagem corrente consiste em adquirir e modelar o conhecimento para um propósito específico e, então, embuti-lo em um assistente. Entretanto, esta abordagem apresenta uma série de problemas e esta tendência precisa ser revertida para que possamos construir ambientes integrados.

Na medida em que prolifera o número de assistentes baseados em conhecimento em um ADS, passa a ser necessário considerar a integração de ferramentas como uma questão de quatro dimensões, incluindo, além da integração de dados, controle e apresentação, a integração de conhecimento. Somente com esta dimensão devidamente observada, torna-se viável a construção de ambientes plenamente integrados. Sem considerar a integração de conhecimento, provavelmente, cada ferramenta estará baseada em uma conceituação, utilizará um vocabulário próprio e o conhecimento nela embutido não poderá ser reutilizado ou compartilhado. O conhecimento, da mesma forma que os dados, deve estar disponível para o ambiente como um todo, podendo ser acessado pelas diversas ferramentas, evitando-se redundâncias e inconsistências.

No entanto, a integração de conhecimento não é obtida de forma espontânea. Ao contrário, ela impõe uma mudança na forma de se desenvolver ferramentas baseadas em conhecimento. Ao invés de construirmos assistentes com conhecimento totalmente embutido e otimizado para uma tarefa específica, devemos buscar uma abordagem onde estes sejam capazes de compartilhar corpos de conhecimento comum em um domínio de aplicação. Assim, o enfoque deve ser o desenvolvimento *para* reuso, neste contexto, com uma perspectiva mais desafiadora: o reuso de conhecimento. Contudo, desenvolver Sistemas Baseados em Conhecimento com este enfoque requer uma abordagem diferente para o processo de modelagem do conhecimento. Este é o tema de discussão do próximo capítulo.

Capítulo 3

Modelagem de Conhecimento

Nos últimos anos, tem havido um crescente interesse em descrever sistemas complexos de processamento de informação em termos do conhecimento por eles manipulado. A construção de protótipos de sistemas especialistas em âmbito acadêmico tem dado lugar ao desenvolvimento de Sistemas Baseados em Conhecimento (SBCs) aplicados a negócios.

Inicialmente, a maior parte dos esforços para incorporar conhecimento aos sistemas concentrava-se na construção de mecanismos uniformes e gerais de representação. O uso de formalismos e mecanismos de inferência uniformes (p. ex., regras e seus mecanismos de encadeamento progressivo ou regressivo) eram considerados a base para o desenvolvimento de SBCs. Esta uniformidade, contudo, teve como conseqüência direta a incapacidade de separar os conhecimentos do domínio da aplicação e da tarefa a ser realizada. Uma vez que a máquina de inferência era de propósito geral, a estratégia para resolver um problema era embutida como parte da base de conhecimento. Com isso, a reutilização de conhecimento tornou-se praticamente inviável. Do ponto de vista do domínio, o reuso era problemático dado que o conhecimento era adquirido para uma tarefa específica. Do ponto de vista da tarefa, apesar de tarefas como diagnose e planejamento, mesmo em diferentes domínios ou aplicações, apresentarem vários aspectos em comum, estes não eram explicitamente representados e, portanto, o reuso também não era possível.

De fato, esta incapacidade de se perceber as diferentes facetas do conhecimento estava intrinsecamente relacionada à forma como era conduzido o processo de aquisição de conhecimento. A aquisição de conhecimento era vista como um processo de extrair o conhecimento de um especialista humano e transferi-lo para um sistema. Assumia-se que o especialista era uma "mina" de conhecimento, e que o papel do engenheiro de conhecimento era "explorar" essa mina (VALENTE, 1995). Especialistas utilizam um grande número de heurísticas, o que lhes permite evitar caminhos de raciocínio pouco eficientes. Todavia, exatamente pelo fato do conhecimento estar compilado, a separação dos diferentes tipos de conhecimento não é trivial. Além disso, a ênfase exclusivamente no conhecimento do especialista é por si só uma falha. Podem existir várias outras fontes de conhecimento igualmente importantes. Ao relegar estas fontes, a estratégia de transferência do conhecimento do especialista para o sistema não apenas tornava a tarefa de aquisição mais difícil, como também reforçava o problema da superficialidade. Outro problema proveniente desta abordagem refere-se ao fato dela não levar em conta o fato do conhecimento sobre a natureza do problema a ser resolvido poder ser usado para guiar a aquisição.

CLANCEY (1993) propôs mudar tal perspectiva, argumentando que o foco da Engenharia de Conhecimento deve ser a modelagem de sistemas e não a tentativa de reproduzir a maneira como pensam os especialistas. CLANCEY (1993) é um dos defensores da *visão de modelagem* para a Engenharia de Conhecimento, de acordo com a qual, uma base de conhecimento não é um repositório de conhecimento extraído da mente de um especialista, mas o resultado de uma atividade de modelagem, cujo objeto é o comportamento observado de um agente inteligente atuando em seu ambiente.

Este capítulo explora algumas facetas da modelagem de conhecimento, procurando identificar elementos diretamente relacionados à reutilização e ao compartilhamento de conhecimento que, em última instância, são as metas da integração de conhecimento em ADSs. A seção 3.1 discute a questão da representação de conhecimento e seus níveis. A seção 3.2 apresenta, sucintamente, algumas abordagens para a modelagem de conhecimento. Na seção 3.3, o enfoque recai sobre o uso de ontologias no desenvolvimento de SBCs. Na seção 3.4, procuramos avançar

neste campo de estudos, apresentando uma linguagem gráfica para a descrição de ontologias, um método para a construção de ontologias e uma abordagem de Engenharia de Conhecimento baseada em Ontologias. Finalmente, na seção 3.5, são apresentadas as conclusões deste capítulo.

Vale a pena realçar que não é nossa intenção discutir o processo de desenvolvimento de SBCs como um todo, mas apenas seus aspectos relacionados à modelagem de conhecimento. Para uma discussão mais ampla sobre o processo de desenvolvimento, métodos, técnicas e ferramentas para a construção de SBCs, vide (WERNECK, 1995).

3.1 - Representação de Conhecimento e seus Níveis

Ao observarmos uma entidade inteligente raciocinando sobre o mundo, deparamo-nos com um fato importante e inevitável: raciocínio é um processo interno, mas a maioria das coisas sobre as quais se raciocina existe apenas externamente. Assim, o aspecto primordial em qualquer sistema inteligente é a conceituação da porção do mundo sobre a qual se raciocina. Uma conceituação é um conjunto de conceitos que se assume existir em um domínio de interesse, bem como a interação entre eles.

É impossível representar o mundo real, ou mesmo uma parte dele, em sua completa riqueza de detalhes. Para representar um certo fenômeno ou uma porção do mundo, a que chamamos domínio, é necessário concentrar a atenção em um número limitado de conceitos, suficientes e relevantes para se criar uma abstração do fenômeno em estudo. Desta forma, várias conceituações diferentes são possíveis para um mesmo domínio e, para capturar a conceituação de uma porção do mundo, é necessário utilizar alguma *representação*.

Segundo DAVIS et al. (1993), a noção de representação de conhecimento pode ser melhor compreendida em termos de cinco papéis distintos que esta desempenha:

1. Uma representação de conhecimento é um mecanismo usado para se raciocinar sobre o mundo ao invés de agir diretamente sobre ele. Neste sentido, ela é, fundamentalmente, um *substituto* para aquilo que representa. Este papel conduz, naturalmente, a dois importantes aspectos. O primeiro diz respeito à sua

identidade projetada. Deve haver alguma forma de correspondência especificada entre o substituto e seu referente planejado no mundo. Esta correspondência é a semântica da representação. O segundo aspecto é a fidelidade. Fidelidade perfeita é, em geral, impossível, tanto na prática quanto em princípio. A única representação completamente precisa de um objeto é o objeto em si. Qualquer outra representação é imprecisa e, inevitavelmente, contém simplificações.

2. Uma representação de conhecimento é uma resposta à pergunta "Em que termos devo pensar sobre o mundo?", isto é, um conjunto de compromissos ontológicos. Uma vez que toda representação é uma aproximação imperfeita da realidade, ao selecionar uma representação, estamos tomando um conjunto de decisões sobre como e o que ver no mundo. Ou seja, selecionar uma representação significa fazer um conjunto de compromissos ontológicos. Esses compromissos determinam o que pode ser visto, enfocando alguma parte do mundo em detrimento de outras. Esta forma de ver o mundo não é apenas um efeito colateral da escolha de uma representação; ao contrário, o efeito focalizador é parte essencial do que a representação oferece, já que a complexidade do mundo real é esmagadora. Assim, o comprometimento ontológico feito por uma representação pode ser uma de suas mais importantes contribuições.
3. Uma representação de conhecimento é uma teoria fragmentária de raciocínio que especifica que inferências são válidas e quais são recomendadas. Uma representação é motivada por alguma percepção de como as pessoas argumentam ou por alguma crença sobre o que significa raciocinar de forma inteligente. A teoria de raciocínio inteligente embutida em uma representação é geralmente implícita, mas pode se tornar mais evidente pelo exame de três componentes: a concepção de inferência inteligente, o conjunto de inferências que a representação sanciona e o conjunto de inferências que ela recomenda. Enquanto as inferências sancionadas indicam o que pode ser inferido, as inferências recomendadas dizem respeito ao que deve ser inferido. Esta orientação é necessária, pois o conjunto de inferências sancionadas é tipicamente muito grande para ser usado indiscriminadamente. Estes

componentes podem ser vistos, também, como as respostas da representação a três questões fundamentais: (i) O que significa raciocinar de forma inteligente? (ii) O que podemos inferir a partir do que conhecemos? e (iii) O que devemos inferir a partir do que conhecemos?

4. Uma representação de conhecimento é um meio de computação pragmaticamente eficiente. Na realidade, esta questão aborda a utilidade prática da representação. Se ela torna coisas possíveis mas não facilmente computáveis, então, a representação pode não ser de muita valia para o problema em mãos.
5. Uma representação de conhecimento é um meio de expressão, i.e., uma linguagem na qual se pode dizer coisas sobre o mundo. Esta é, também, uma questão de utilidade prática da representação. Se ela permite expressar certas situações, mas isto não é feito facilmente, então, como usuários, muitas vezes não podemos saber se a representação não é capaz de expressar alguma coisa que gostaríamos de dizer ou simplesmente não sabemos como usá-la. Uma representação é a linguagem na qual nos comunicamos e, assim, devemos ser capazes de falar sem esforço heróico.

Estes cinco papéis têm impacto nas várias fases da Engenharia de Conhecimento. Mais especificamente, afora o papel de número 4, os demais devem ser atentamente observados na modelagem de conhecimento.

Devemos frisar que a concepção de representação de conhecimento adotada neste trabalho é a mesma que em (DAVIS et al., 1993): uma forma utilizada para representar conhecimento. O conjunto familiar de ferramentas básicas de representação, tais como lógica, regras, *frames* e redes semânticas, é referenciado aqui como tecnologias de representação de conhecimento. É uma prática bastante comum construir representações de conhecimento em múltiplos níveis de linguagens, tipicamente, com uma das tecnologias de representação de conhecimento no nível inferior.

Inicialmente, a Engenharia do Conhecimento adotava uma abordagem de natureza puramente computacional, isto é, a ênfase era a tecnologia de representação, em detrimento da semântica do conhecimento. É verdade que, em algum ponto do processo de desenvolvimento, temos de nos defrontar com decisões sobre qual meio

de implementação utilizar; entretanto, a resposta computacional é apenas parcialmente satisfatória. Há uma grande distância conceitual entre o nível de implementação e o conhecimento de resolução de problema observado no mundo real. Assim, faz-se necessário um outro nível de discurso onde o conhecimento e a resolução de problema sejam tratados independentemente de suas possíveis implementações.

NEWELL (1982) propôs a adoção de uma perspectiva de nível de conhecimento, em adição ao nível simbólico. No nível de conhecimento, agentes de resolução de problema podem ser caracterizados em termos das ações que podem executar, do conhecimento que possuem e de suas metas. Um formalismo é apenas um sistema simbólico que codifica um corpo de conhecimento. Contudo, apenas esta diferenciação é ainda insuficiente. De fato, é possível identificar níveis mais refinados de representação de conhecimento.

O nível simbólico proposto por NEWELL (1982) pode ser decomposto em dois níveis: o nível de estruturas de dados e o nível de tecnologia de representação. No *nível de estrutura de dados* ou *de implementação*, o conhecimento do sistema é representado em termos de estruturas de dados, tais como grafos, listas ou células de memória. O *nível de tecnologia de representação*, por sua vez, já envolve alguns compromissos ontológicos. Lógica, regras, *frames* e quaisquer outras tecnologias de representação, incorporam uma visão sobre os tipos de coisas que são importantes no mundo. Na lógica, por exemplo, o mundo é visto em termos de entidades individuais e relações entre elas.

Cada tecnologia de representação adota um modo próprio de observação do mundo e, portanto, selecionar uma tecnologia de representação envolve um certo grau de comprometimento ontológico. De fato, os compromissos ontológicos de uma representação acumulam-se em camadas, sendo o nível de tecnologia de representação a camada inferior.

Ainda que as tecnologias de representação provejam um meio de ver as coisas no mundo, elas não indicam como instanciar esta visão. Na verdade, este é o objetivo da modelagem de conhecimento: elaborar uma conceituação da porção do mundo em estudo. Durante esta fase, as discussões devem se dar totalmente no nível de

conhecimento proposto por Newell. Entretanto, este nível pode ser desmembrado em outros dois: nível epistemológico e nível ontológico.

No *nível epistemológico,* especifica-se a estrutura dos conceitos e seus inter-relacionamentos. Modelos de objetos e de dados são exemplos de representações neste nível. Contudo, para representar conhecimento, tais representações são ainda pobres. Esses formalismos estabelecem apenas significados particulares de estruturação. É necessário, portanto, introduzir a noção de *nível ontológico* (GUARINO, 1993). Enquanto o nível epistemológico é o nível de estruturação, o nível ontológico é o nível de *significação*. No nível ontológico, as primitivas de conhecimento satisfazem postulados formais de significação, que restringem a interpretação de uma teoria com base em uma ontologia formal. Assim, uma ontologia é uma especificação formal de uma conceituação.

Para melhor ilustrar estes dois níveis, observemos o pequeno extrato de um modelo de entidades e relacionamentos de um sistema acadêmico para uma universidade, mostrado na figura 3.1. Esta representação de nível epistemológico mostra que existem conceitos no mundo de aluno, disciplina e turma, e que há relacionamentos de matrícula, histórico e oferta entre instâncias desses conceitos. Pode-se perceber que, neste nível, só é possível representar a estrutura das coisas no mundo. Entretanto, algumas questões importantes permanecem em aberto, entre elas: Que elementos do conjunto de entidades *aluno* podem estar relacionados com uma instância específica de *turma*? O que é exatamente uma instância de *turma*? Que subconjunto do produto cartesiano das instâncias de *aluno* e *disciplina* caracteriza precisamente o relacionamento *histórico*? Tais questões não são respondidas por esse modelo, já que ele é um modelo do nível epistemológico. A solução normalmente adotada nestes casos consiste em embutir este conhecimento dentro do código de um programa, o que dificulta o seu reuso e compartilhamento. Construindo-se ontologias formais, é possível estabelecer significados formais para alguns termos do vocabulário do domínio, assim como pode-se restringir a interpretação da teoria com base na axiomatização da ontologia.

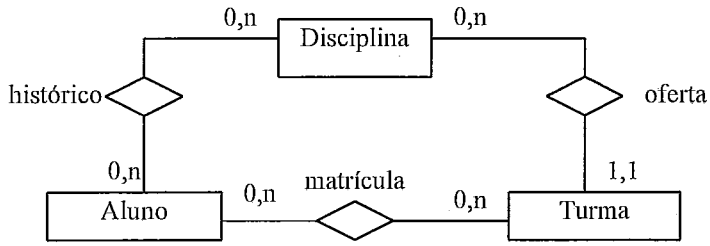


Figura 3.1 - Extrato de um Modelo ER para um Sistema Acadêmico.

No caso do sistema acadêmico, axiomas da ontologia poderiam, por exemplo, impor condicionantes para o estabelecimento do relacionamento *matrícula*, tal como: se *a* é um aluno e existe um relacionamento *histórico(a,d)*, indicando que este aluno já cursou a disciplina *d* tendo sido aprovado, então não é possível haver o relacionamento *matrícula(a,t)*, se a turma *t* for uma oferta da disciplina *d*, (isto é *oferta(t,d)*) já cursada pelo aluno. A seguinte sentença, em lógica de primeira ordem, é a formalização deste axioma:

$$(\forall\ a,d,t)\ (aluno(a) \wedge disciplina(d) \wedge turma(t) \wedge$$
$$histórico(a,d,Aprovado) \wedge oferta(t,d) \rightarrow \neg\ matrícula(a,t))$$

Desta forma, uma ontologia formal pode ser usada para estabelecer um acordo sobre o vocabulário e seus axiomas podem ser utilizados para verificar se um determinado uso é consistente, o que facilita a reutilização do conhecimento em contextos não previstos antecipadamente.

A formalização de uma ontologia requer uma linguagem formal, tal como a linguagem de predicados de primeira ordem. Entretanto, é bom frisar que tal linguagem é apenas um formalismo do nível simbólico. Sentenças em lógica de primeira ordem são apenas cadeias de símbolos satisfazendo condições sintáticas. A lógica é ontologicamente neutra e, precisamente por essa razão, é utilizada na construção de teorias ontológicas.

3.2 - Abordagens para a Modelagem de Conhecimento

A modelagem é o aspecto central da Engenharia de Conhecimento e a construção de um SBC deve ser vista como a construção de um modelo do comportamento desejado. Não se busca criar um modelo que, no geral, simule os processos cognitivos de um especialista, mas sim criar um modelo que produza resultados similares na resolução de problemas na área de interesse. Um SBC, portanto, dificilmente será o equivalente funcional e comportamental de um especialista. Há várias razões para tal. Primeiro, a introdução da tecnologia de informação freqüentemente envolve novas distribuições de funções e papéis dos agentes e, assim, um SBC pode executar funções que não fazem parte do repertório do especialista. Segundo, geralmente não se consegue tornar completamente explícito o processo de raciocínio que fundamenta um especialista. Por fim, há diferenças inerentes entre as capacidades das máquinas e das pessoas.

Desta forma, a Engenharia de Conhecimento não deve ser vista como um processo de transferência de conhecimento para uma representação computacional, mas como a construção de um modelo de conhecimento. Na visão de modelagem, a aquisição de conhecimento é essencialmente um processo construtivo no qual o engenheiro de conhecimento usa todos os tipos de informação sobre o comportamento do especialista e estabelece as decisões finais de modelagem (WIELINGA et al., 1992).

Há muitas possíveis visões sobre o comportamento de uma solução de problema, e para cada visão pode corresponder um modelo diferente. Além disso, os procedimentos usados para modelar o conhecimento variam de proposta a proposta.

A abordagem de *Tarefas Genéricas* (CHANDRASEKARAN et al., 1993), por exemplo, utiliza as noções de tarefa, método, sub-tarefa e estrutura de tarefa como componentes de modelagem, representando uma infra-estrutura uniforme para analisar um sistema sob uma perspectiva de tarefa. O termo "tarefa" é usado como sinônimo de tipos de problema, tais como diagnose e planejamento. Métodos, por sua vez, são usados para realizar tarefas. Uma estrutura de tarefa associa tarefas aos métodos que podem ser usados para executá-las e aos tipos de conhecimento requeridos pelos métodos. Esta estrutura é uma árvore de tarefas, métodos e sub-tarefas aplicadas

recursivamente, até se obter tarefas passíveis de execução direta usando o conhecimento disponível.

A abordagem de *Componentes de Especialidade* (CoE) (STEELS, 1990), assim como as Tarefas Genéricas, parte de uma análise detalhada da tarefa da aplicação. Tarefas têm estrutura interna e podem ser decompostas em sub-tarefas. Para cada tarefa ou sub-tarefa são construídos um ou mais *modelos de caso*. O conhecimento de domínio requerido pelos vários tipos de modelos de caso é descrito em *modelos de domínio*. Um método de resolução de problema é responsável por aplicar o conhecimento de domínio a uma tarefa, podendo consultar modelos de domínio, criar ou alterar estruturas intermediárias, executar ações para obter mais dados (p.ex., interagindo com um usuário) ou expandir um modelo de caso, adicionando ou modificando fatos.

KADS (SCHREIBER et al., 1993) é o resultado de uma pesquisa realizada em um projeto ESPRIT, visando o desenvolvimento de uma metodologia ampla para a construção de SBCs. KADS está fundamentada em cinco princípios básicos: múltiplos modelos, modelagem de conhecimento em níveis, reutilização de componentes, refinamento de modelos e projeto preservando a estrutura.

CommonKADS (BREUKER et al., 1994) é a mais recente versão da metodologia KADS, prevendo gerência de projeto, análise organizacional, engenharia de conhecimento e engenharia de software para SBCs. Em CommonKADS, o desenvolvimento de um SBC é visto como a construção de modelos de comportamento de resolução de problema em seu contexto organizacional concreto. CommonKADS provê um conjunto de modelos, dito o *modelo do produto*, que permite expressar as várias perspectivas na situação de resolução de problema. Os modelos de organização, tarefa, agente e comunicação capturam o contexto da atividade de resolução de problema de interesse para o projeto de SBC. O modelo de especialidade descreve o conhecimento e o raciocínio envolvidos na execução das tarefas. Finalmente, o modelo de projeto descreve a sua realização computacional.

Considerações sobre as Abordagens Apresentadas

A aquisição de conhecimento é, sem dúvida, o aspecto crucial no desenvolvimento de SBCs e, portanto, faz-se necessário o uso de uma abordagem sistemática, preferencialmente, apoiada em alguma forma de reuso. Uma idéia chave, introduzida para aumentar a reusabilidade do conhecimento e facilitar a manutenção dos sistemas resultantes, consiste em separar os conhecimentos de domínio e de tarefa. Esta separação mostra que há dois problemas basicamente distintos, apesar de relacionados, no projeto de um SBC e que estes podem ser, pelo menos até certo ponto, investigados e resolvidos separadamente. Desta forma, podem existir modelos estruturados separados para representar o domínio e a tarefa.

Analisando as abordagens anteriormente citadas, é possível notar que elas compartilham a visão de que o conhecimento deve ser modelado tendo por base uma tarefa. De maneira geral, seus principais componentes de modelagem são tarefas, sub-tarefas e métodos de resolução de problemas, sendo que o conhecimento sobre o domínio da aplicação é normalmente capturado com base na tarefa em mãos. Assim, implícita ou explicitamente, defendem que o conhecimento do domínio deve ser adaptado à tarefa em mãos.

O enfoque centrado em tarefas é importante. Do ponto de vista teórico, ele oferece um meio eficaz para construir teorias de conhecimento que fazem significativas generalizações empíricas acerca da resolução de tipos genéricos de problemas. Uma teoria desta natureza identifica um conjunto de tarefas genéricas e lista, para cada uma delas, os tipos de métodos de resolução de problema e modelos de domínio esperados. Uma vez que tal teoria exista, tem-se modelos poderosos para auxiliar a aquisição de conhecimento para uma aplicação específica.

Os defensores das abordagens centradas em tarefas argumentam que o conhecimento do domínio depende fortemente da tarefa específica que se tem em mãos. Esta posição é por eles justificada com base no *problema da interação*, apresentado inicialmente por Bylander e Chandrasekaran (HEIJST et al., 1997a):

> *"A representação de conhecimento para o propósito de resolver um dado problema é fortemente afetada pela natureza do problema e a estratégia de inferência a ser aplicada."*

Segundo HEIJST et al. (1997a), são duas as principais razões para o problema da interação: primeiro, a tarefa da aplicação determina em grande parte que tipos de conhecimento devem ser codificados; segundo, o conhecimento deve ser codificado de modo que a estratégia de inferência usada possa funcionar eficientemente.

Por outro lado, no contexto da Engenharia de Conhecimento, os defensores da abordagem baseada em domínio argumentam que o conhecimento é de utilidade geral, ou seja, um mesmo corpo de conhecimento pode ser utilizado em diferentes contextos e problemas e, portanto, não deve ser amarrado a uma tarefa.

Recentemente, o uso de ontologias tem se tornado a base para as abordagens baseadas em domínio. Ontologias são indicadas para se estabelecer um acordo sobre o conhecimento a ser trocado entre agentes baseados em conhecimento, removendo diferenças arbitrárias no nível de conhecimento. Muitas vezes, tais diferenças são introduzidas apenas para tratar particularidades da tarefa em questão. Neste caso, o problema tem origem na abordagem centrada nas tarefas, uma vez que a dificuldade de se integrar conhecimento advém do fato do conhecimento de domínio ter sido capturado segundo a perspectiva específica de uma tarefa. Desta forma, uma abordagem mais equilibrada deve preocupar-se com o conhecimento do domínio de forma independente das tarefas que irão manipulá-lo.

Esta dualidade tarefa-domínio tem sido alvo de constantes debates na área de SBCs. GUARINO (1997), por exemplo, argumenta que o problema da interação, como colocado anteriormente, retrata um problema de *tecnologia de representação* e, portanto, está no nível simbólico de NEWELL (1982). Segundo ele, no nível de conhecimento, certamente diminui a importância do problema da interação, que poderia ser re-escrito da seguinte forma:

> *"O conhecimento requerido para resolver um dado problema é fortemente afetado pela natureza do problema."*

Colocado desta forma, o problema da interação pode ser visto como uma relação de relevância entre o conhecimento e o problema. Indubitavelmente, uma porção específica de conhecimento pode ser mais ou menos *relevante* para uma tarefa particular, mas isto não quer dizer que este conhecimento é *específico* da tarefa. Assim, GUARINO (1997) defende a tese da independência do conhecimento de

domínio, reforçando que a reusabilidade ao longo de múltiplas tarefas ou métodos pode e deve ser sistematicamente perseguida.

Alimentando o debate, HEIJST et al. (1997b) alegam que a diferença de opinião em relação a GUARINO (1997) tem origem em diferentes visões sobre a natureza do nível de conhecimento. A visão de GUARINO (1997) corresponde de perto à formulação original de NEWELL (1982), onde, no nível de conhecimento, um agente é descrito em termos de seu comportamento racional, independentemente da tecnologia de sua representação. Contudo, eles argumentam que é difícil imaginar como o nível de conhecimento de NEWELL (1982) poderia ser usado na prática, sem qualquer estrutura ou representação. Assim, embora inspirados nas idéias de NEWELL (1982), eles propõem uma interpretação mais pragmática do nível de conhecimento. Nesta visão, *modelos* de nível de conhecimento são descrições do conhecimento requerido para resolver algum problema, formuladas em uma linguagem que não restrinja a expressividade em favor da eficiência. Neste sentido, tais modelos são também representações e, portanto, é necessário encarar o problema da interação também no nível de conhecimento (HEIJST et al., 1997b).

De fato, há vários níveis de representação, como discutido na seção 3.1, e modelos de conhecimento são sempre representações. Mas, assim como GUARINO (1997) , acreditamos que, na aquisição de conhecimento, a ênfase deve ser a relevância do conhecimento.

Inegavelmente, tarefa e domínio estão fortemente relacionados e o conhecimento sobre uma perspectiva pode auxiliar a aquisição do conhecimento da outra. O que devemos buscar é uma abordagem conciliadora que procure tirar vantagem de ambas as perspectivas. Nesta abordagem, a aquisição de conhecimento deve se dar em três etapas. Primeiro, procede-se uma análise do conhecimento do domínio da aplicação. A seguir, o foco passa a ser a tarefa genérica que melhor reflete o problema a ser resolvido. Finalmente, na terceira e última etapa, os modelos genéricos de domínio e de tarefa são utilizados para guiar a aquisição do conhecimento específico da aplicação em desenvolvimento.

Uma abordagem desta natureza tem duas vantagens diretas. Primeiro, é possível reutilizar tanto conhecimento de domínio quanto conhecimento de tarefa, na

forma de ontologias de domínio e modelos de tarefa, respectivamente. Segundo, e mais importante, estes modelos podem ser usados para guiar a aquisição do conhecimento específico da aplicação. Uma abordagem nesta linha está descrita na seção 3.4. Antes, contudo, faz-se necessário um estudo detalhado de ontologias, um componente essencial para a abordagem proposta.

3.3 - Ontologias

Uma *ontologia* é uma especificação de uma conceituação (GRUBER, 1995), isto é, uma descrição de conceitos e relações que podem existir para um agente ou uma comunidade de agentes. É importante realçar que uma ontologia não descreve apenas conhecimento imediato, isto é, conhecimento factual que pode ser obtido diretamente a partir da observação do domínio, mas também conhecimento derivado, ou seja, conhecimento obtido através de inferência sobre o conhecimento imediato disponível.

O termo ontologia foi adotado da Filosofia. Já há muito tempo, filósofos têm usado ontologias para tentar descrever domínios naturais (as coisas naturais do mundo) e a existência dos seres e coisas em si. Entretanto, na Inteligência Artificial, ontologias têm um caráter um pouco distinto. Na Filosofia, ontologias são usadas com o intuito de desvendar o significado das coisas no mundo, procurando descrever a natureza das coisas. Na IA, por sua vez, ontologias são usadas para descrever domínios já consagrados, como Medicina, Engenharia e Direito, onde é possível saber o significado projetado das coisas. Assim, o que se busca com uma ontologia em IA é firmar um acordo sobre o vocabulário do domínio de interesse, a ser partilhado por agentes que conversam sobre ele.

O uso de ontologias se constitui em uma ferramenta útil para apoiar a especificação e implementação de qualquer sistema de computação complexo. Uma ontologia é desenvolvida para satisfazer a um dos seguintes propósitos (SMITH, 1996):

- permitir que múltiplos agentes compartilhem seu conhecimento;
- ajudar as pessoas a compreender melhor uma certa área de conhecimento;
- ajudar outras pessoas a compreender uma certa área de conhecimento;

- ajudar pessoas a atingir um consenso no seu entendimento sobre uma área de conhecimento.

Auxiliar os desenvolvedores a formalizar e elicitar suas especificações e conhecimento é um importante papel das ontologias. Contudo, ontologias são capazes, também, de apoiar a implementação prática de sistemas. Uma ontologia em um Sistema Baseado em Conhecimento tem um papel similar a um meta-modelo de dados em um sistema de banco de dados (SMITH, 1996).

Na medida em que tem crescido o interesse por ontologias em IA, estas tem sido utilizadas de diferentes maneiras. Com base no seu conteúdo, ontologias podem ser classificadas em (HEIJST et al., 1997a) (GUARINO, 1997) (CHANDRASEKARAN et al., 1997) (GUARINO, 1998):

- *ontologias genéricas*: descrevem conceitos bastante gerais, tais como, espaço, tempo, matéria, objeto, evento, ação, etc., que são independentes de um problema ou domínio particular;
- *ontologias de domínio*: expressam conceituações de domínios particulares, descrevendo o vocabulário relacionado a um domínio genérico, tal como Medicina.
- *ontologias de tarefas*: expressam conceituações sobre a resolução de problemas, independentemente do domínio em que ocorram, isto é, descrevem o vocabulário relacionado a uma atividade ou tarefa genérica, tal como, diagnose ou vendas;
- *ontologias de aplicação*: descrevem conceitos dependentes do domínio e da tarefa particulares. Estes conceitos freqüentemente correspondem a papéis desempenhados por entidades do domínio quando da realização de uma certa atividade;
- *ontologias de representação*: explicam as conceituações que fundamentam os formalismos de representação de conhecimento.

Ontologias de domínio são construídas para serem utilizadas em um micro-mundo. São os tipos mais comumente desenvolvidos, sendo que diversos trabalhos são encontrados na literatura, enfocando áreas como química (GÓMEZ-PÉREZ et al., 1996), manufatura de aeronaves (BARLEY et al., 1997), modelagem de

empreendimento (FOX et al., 1993) (GRÜNINGER et al., 1995), medicina (HEIJST et al., 1997a) (HUMPHREYS et al., 1993) (OLIVEIRA et al., 1998), sistemas físicos (BORST et al., 1997), direito (VALENTE, 1995), biologia e bioquímica (KARP et al., 1993) e ciência dos materiais (VAN DER VET et al., 1993), entre outros.

A pesquisa enfocando ontologias genéricas procura construir teorias básicas do mundo, de caráter bastante abstrato, aplicáveis a qualquer domínio (conhecimento de senso comum). Entre os trabalhos nesta categoria, destacam-se (BUNGE, 1977), (SOWA, 1995), (LENAT et al., 1990), (HOBBS, 1995), (GUARINO, 1995) e (SWARTOUT et al., 1997). Estes trabalhos estão bastante alinhados com o uso de ontologias na Filosofia e procuram descrever a natureza das coisas. Tipicamente, ontologias genéricas definem conceitos tais como coisa, estado, evento, processo, ação, etc., com o intuito de serem especializados na definição de conceitos em uma ontologia de domínio.

Ontologias de representação procuram tornar claros os compromissos ontológicos embutidos em formalismos de representação de conhecimento. Um exemplo desta categoria é a ontologia de *frames*, utilizada em Ontolingua (GRUBER, 1992).

O estudo de ontologias de tarefas é a vertente mais recente do estudo de ontologias. Sua principal motivação é facilitar a integração dos conhecimentos de tarefa e domínio em uma abordagem mais uniforme e consistente, tendo por base o uso de ontologias. Trabalhos nesta categoria incluem (MUSEN et al., 1995) e (CHANDRASEKARAN et al., 1997).

GUARINO (1998) propõe que ontologias sejam construídas segundo seu nível de generalidade, como mostra a figura 3.2. Os conceitos de uma ontologia de domínio ou de tarefa devem ser especializações dos termos introduzidos por uma ontologia genérica. Os conceitos de uma ontologia de aplicação, por sua vez, devem ser especializações dos termos das ontologias de domínio e de tarefa correspondentes.

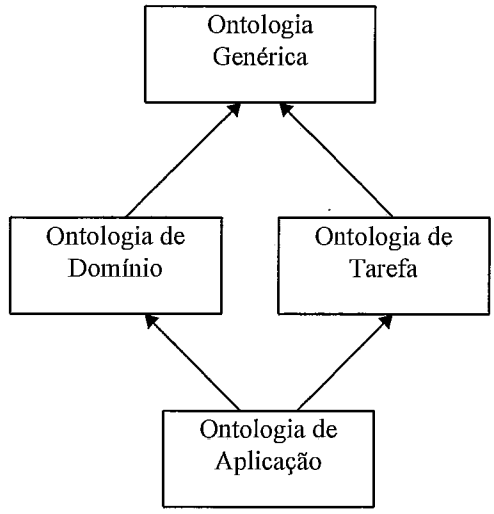


Figura 3.2 - Tipos de Ontologias, segundo seu Nível de Dependência em Relação a uma Tarefa ou Ponto de Vista Particular (GUARINO, 1998).

Uma vez que o uso de ontologias neste trabalho tem por objetivo a integração de conhecimento em ADSs, estamos diretamente interessados em ontologias de domínio e, portanto, a maior parte das discussões que se seguem aplica-se a este tipo de ontologia, ainda que possa ser estendida, com algumas restrições, para os demais tipos.

3.3.1 - O Uso de Ontologias no Desenvolvimento de SBCs

Como visto na seção 3.1, é impossível representar o mundo real, ou mesmo uma parte dele, em sua completa riqueza de detalhes. Para representar um certo fenômeno ou parte do mundo, a que chamamos domínio, é necessário concentrar a atenção em um número limitado de conceitos, suficientes e relevantes, para criar uma abstração do fenômeno em questão. Assim, um aspecto central de qualquer atividade de modelagem consiste em elaborar uma *conceituação*: um conjunto de regras informais que restringem a estrutura de uma parte da realidade, utilizado por um agente para isolar e organizar conceitos e relações relevantes (GUARINO, 1997).

Uma vez que uma ontologia é a especificação explícita de uma conceituação de um domínio, a primeira questão a ser colocada é: *O que uma ontologia deve conter?* Uma ontologia de domínio é uma representação formal e declarativa que

estabelece o vocabulário (conceitos) para os elementos no domínio, as declarações lógicas que descrevem o que são estes elementos, e como estes elementos podem, ou não podem, ser relacionados uns com os outros (SMITH, 1996). Assim, basicamente, uma ontologia consiste de um *vocabulário* específico usado para descrever uma certa realidade, adicionado a um conjunto de suposições explícitas dizendo respeito ao *significado planejado* dos termos do vocabulário. Este conjunto de suposições tem, usualmente, a forma de uma teoria lógica de primeira ordem, onde os termos do vocabulário aparecem como nomes de predicados unários ou binários, representando conceitos e relações. No caso mais simples, uma ontologia descreve uma hierarquia de conceitos relacionados por associações de especialização / generalização; em casos mais complexos, axiomas apropriados são adicionados para expressar outros relacionamentos entre conceitos ou para restringir sua interpretação (GUARINO, 1998).

Em suma, uma ontologia consiste de conceitos e relações - o vocabulário - e suas definições, propriedades e restrições, descritas na forma de axiomas. Conceitos são normalmente organizados em taxonomias. Alguns conceitos, ditos primitivos, não podem ser expressos em termos de outros conceitos e, desta forma, necessitam ser definidos textualmente, ou explicados através de exemplos. Entretanto, simplesmente propor um conjunto de termos básicos pode ser pouco para uma ontologia. Axiomas devem ser providos para definir a semântica dos termos. Axiomas são um meio de definir e representar outras informações sobre conceitos e suas relações, incluindo restrições sobre propriedades e seus valores. Idealmente, uma ontologia não deve ser uma simples hierarquia de termos, mas uma infra-estrutura teórica que verse sobre o domínio.

Ontologias de domínio não devem ter como objetivo a especificação dos termos mais comuns, mas a especificação das *categorias básicas* de conhecimento do domínio. A escolha dos termos a serem incluídos em uma ontologia não deve ser arbitrária ou puramente prática. Muitas vezes, termos comumente usados têm significados contraditórios e estão mais relacionados à comunicação do que à essência do conhecimento usado no domínio. Categorias básicas não devem ser vistas como

termos de nível mais alto em uma hierarquia de abstrações, mas ao contrário, como tipos de conhecimento (VALENTE et al., 1997).

Não é objetivo de uma ontologia de domínio descrever todo o conhecimento a ser codificado em uma base de conhecimento. Algum conhecimento empírico, compilado ou prático, que é dependente da tarefa ou aplicação particular, pode encontrar lugar apenas em uma ontologia de aplicação. O conhecimento desempenha papéis na resolução de problemas e, uma vez que muitos destes papéis são dinamicamente atribuídos, não podem fazer parte de uma ontologia de domínio. Assim, não cabe à ontologia de domínio estabelecer interações entre conhecimento de domínio e conhecimento geral de resolução de problema. Mesmo uma ontologia de aplicação não necessariamente contempla todo conhecimento a ser codificado em uma base de conhecimento. Uma base de conhecimento genérica, ao contrário, pode descrever também fatos e declarações relacionadas a um estado de coisas particular. Assim, dentro de uma base de conhecimento genérica, é possível distinguir dois componentes: a ontologia (contendo informação independente de estado) e o conhecimento específico (contendo informação dependente de estado) (GUARINO, 1998).

De fato, um dos principais benefícios do uso de ontologias na especificação de software é a possibilidade de se adotar uma estratégia mais produtiva para a aquisição de conhecimento. Na Engenharia de Conhecimento tradicional, para cada nova aplicação a ser construída, uma nova conceituação é elaborada. Isto se reflete na forma como é conduzida a aquisição de conhecimento: para cada novo SBC, uma fase de aquisição é realizada, quase sempre a partir do nada, enfocando todas as particularidades do sistema em questão. Esta estratégia é extremamente custosa, haja visto que a aquisição de conhecimento é a fase que demanda maiores esforços no desenvolvimento de SBCs. Especialistas são recursos escassos, de alto custo, mas essenciais para a aquisição de conhecimento e, portanto, devem ser melhor aproveitados. Assim, é importante reutilizar e compartilhar o conhecimento capturado. Na abordagem tradicional, porém, como a aquisição de conhecimento considera todas as particularidades da aplicação, seus resultados não são facilmente reutilizáveis, uma vez que é muito difícil separar os vários tipos de conhecimento utilizados.

Ontologias provêem um eficiente mecanismo para aumentar a produtividade na fase de aquisição de conhecimento. Em uma abordagem baseada em ontologias, a aquisição de conhecimento pode ser realizada em duas etapas. Primeiro, o conhecimento geral do domínio, relevante para uma variedade de sistemas no domínio, é capturado e especificado na forma de ontologias. Estas, por sua vez, são usadas para guiar a segunda etapa da aquisição de conhecimento, onde as particularidades de uma aplicação específica são consideradas. Desta forma, uma mesma ontologia pode ser usada para guiar o desenvolvimento de várias aplicações, diluindo os custos da primeira etapa da aquisição e permitindo a reutilização e compartilhamento de conhecimento. A figura 3.3 esboça uma comparação entre as abordagens tradicional e baseada em ontologias para a aquisição de conhecimento.

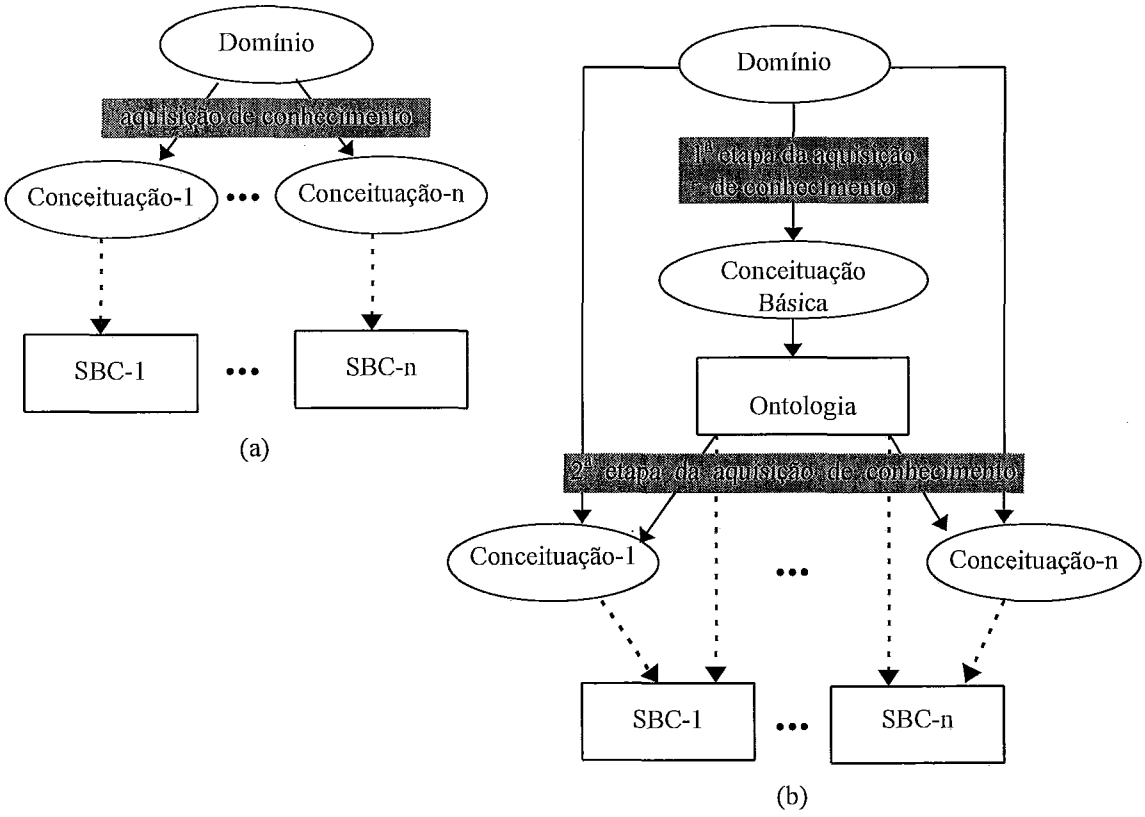


Figura 3.3 - (a) Abordagem Tradicional de Aquisição de Conhecimento.
(b) Abordagem Baseada em Ontologias.

Na abordagem tradicional (figura 3.3a), para cada novo SBC a ser construído, uma nova conceituação é elaborada a partir do domínio. Na abordagem baseada em

ontologias (figura 3.3b), uma conceituação básica do domínio é capturada na forma de ontologias, descrevendo conceitos, relações, propriedades e restrições válidos para uma comunidade de agentes atuando no domínio de discurso. Estas ontologias formam um vocabulário básico para se falar sobre o domínio e são a base para a elaboração das conceituações específicas das aplicações, onde conhecimento empírico ou prático é capturado e codificado em um SBC.

3.3.2 - Usando Ontologias para Obter Integração no Nível de Conhecimento

Uma vez que o principal obstáculo à integração de conhecimento advém da falta de uma conceituação básica do domínio sobre a qual as várias ferramentas possam ser construídas, o uso de ontologias emerge como o ponto-chave na busca da integração. Uma ontologia de domínio deve ser desenvolvida e tornada disponível no ADS, na forma de uma base de conhecimento modular, para ser utilizada para guiar a aquisição do conhecimento específico de cada ferramenta baseada em conhecimento a ser construída, permitindo também o reuso e o compartilhamento de conhecimento.

Para ilustrar esta concepção, vamos voltar ao cenário descrito na seção 2.3, agora considerando uma abordagem baseada em ontologias.

Cenário Revisitado: Usando Ontologias na Construção de Ferramentas

Retornando ao cenário apresentado na seção 2.3, se o ADS possuir um modelo de conhecimento, dado por um conjunto de ontologias, o processo de construção das duas ferramentas seria bastante simplificado. Em primeiro lugar, uma vez que ambas as ferramentas teriam por base o mesmo modelo de conhecimento (figura 2.1(b)), os termos empregados seriam os mesmos, facilitando o intercâmbio de informações. Além disso, o modelo de conhecimento descreve uma teoria e, portanto, ainda que as bases de conhecimento sejam implementadas em um nível simbólico, estas têm uma contraparte no nível de conhecimento, dada pela teoria sobre a qual estão fundamentadas. Desta forma, os conceitos utilizados estão relacionados entre si e é possível manipular o conhecimento de forma explícita, dissociado de sua implementação. Finalmente, uma vez que há um modelo de conhecimento

fundamentando a construção das bases de conhecimento, é possível compartilhar e reutilizar este conhecimento.

3.3.3 - A Construção de Ontologias

De maneira geral, qualquer que seja o domínio, a complexidade envolvida na construção de ontologias é grande e, portanto, algum mecanismo de decomposição deve ser usado para facilitar este processo de construção. Uma abordagem potencialmente interessante é considerar *ontologias em níveis*, isto é, construir ontologias básicas, centrais para o domínio em estudo e, a partir destas, construir ontologias de nível mais alto, adicionando novos elementos e, portanto, estendendo as ontologias dos níveis inferiores.

Esta abordagem pode ser vista como uma adaptação da abordagem de ontologias em níveis de CommonKADS (BREUKER et al., 1994). CommonKADS propõe que ontologias sejam construídas em três níveis, onde cada nível é formulado em termos do nível inferior, mais amplamente aplicável. As *ontologias de base* correspondem a meta-ontologias, isto é, ontologias definindo os conceitos a serem utilizados na definição de ontologias. CommonKADS propõe sua própria meta-ontologia, definindo atributos, valores, expressões, relações de valor, conceitos, instâncias, relações e estruturas. As *ontologias intermediárias* são ontologias de aplicação e as *ontologias orientadas a tarefa* não são ontologias propriamente ditas, mas os papéis que o conhecimento pode desempenhar na realização de uma tarefa específica. Usando uma metáfora de programação, ontologias de base são vistas como tipos primitivos de dados, ontologias intermediárias como tipos abstratos de dados e ontologias orientadas a tarefa como tipos de dados de propósito específico.

Na abordagem defendida neste trabalho, não há um número pré-definido de níveis, nem tão pouco características rígidas para cada nível. O nível mais baixo ainda corresponde à meta-ontologia, mas, acima dele, um número qualquer de níveis de ontologias pode ser utilizado. Se ontologias genéricas forem utilizadas, elas deverão estar localizadas nos níveis inferiores. Ontologias centrais, básicas para um domínio amplo, devem estar em níveis intermediários. Nos níveis mais altos, devem aparecer as ontologias de aplicação.

Uma outra ferramenta poderosa no desenvolvimento de ontologias é o uso de micro-teorias (LENAT et al., 1990). Uma micro-teoria pode ser vista, também, como uma extensão de uma ontologia, mas sua conotação é bastante diferente das ontologias em níveis. Enquanto uma ontologia de nível mais alto estende uma ontologia de nível inferior adicionando novos conceitos, relações e axiomas, uma micro-teoria adiciona apenas axiomas a uma ontologia. Assim, uma micro-teoria, na realidade, restringe uma ontologia, como mostra a figura 3.4. Em linhas gerais, uma micro-teoria consiste de: (i) relações explícitas com uma ontologia, (ii) uma semântica localmente consistente, e, (iii) axiomas descrevendo detalhes ou facilitando a resolução de problema dentro do domínio. Contudo, é necessário assegurar que o modo como os termos são usados e os seus significados nos axiomas da micro-teoria sejam consistentes com o uso projetado pela ontologia (KIM et al., 1994).

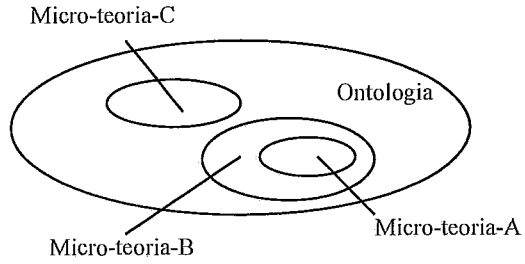


Figura 3.4 - Ontologia e Micro-teorias Associadas.

Uma vez que ontologias são projetadas, a escolha de seus compromissos ontológicos é, de fato, uma decisão de projeto. Para guiar e avaliar o projeto de ontologias, são necessários critérios de qualidade objetivos, fundamentados no propósito do artefato resultante. GRUBER (1995) enumerou um conjunto de critérios para avaliar a qualidade de ontologias. Estes critérios, relacionados a seguir, devem nortear o processo de construção de uma ontologia em todas as suas etapas.

- *Clareza*: Uma ontologia deve comunicar efetivamente o significado projetado dos termos definidos e, assim, suas definições devem ser objetivas. Onde for possível, uma definição completa é preferida em relação a uma definição parcial e todas as definições devem ser documentadas em linguagem natural, de modo a reforçar a clareza.

- *Coerência*: Uma ontologia deve ser coerente, isto é, deve comportar apenas inferências consistentes com as definições. Coerência deve ser observada, também, em relação a conceitos definidos informalmente. Se uma sentença passível de ser inferida a partir dos axiomas da ontologia contradiz uma definição ou exemplo dado informalmente, então a ontologia é incoerente.
- *Extensibilidade*: Uma ontologia deve ser projetada para antecipar usos do vocabulário compartilhado e, portanto, sua representação deve poder ser estendida e especializada. Em outras palavras, deve ser possível definir novos termos para usos especiais, com base no vocabulário existente, sem haver necessidade de rever definições existentes.
- *Compromissos de codificação mínimos*: A conceituação deve ser especificada no nível de conhecimento sem depender de uma tecnologia particular de representação de conhecimento. Uma tendência de codificação surge quando escolhas de representação são feitas puramente para a conveniência de notação ou implementação. Assim, essa tendência deve ser minimizada, já que agentes compartilhando conhecimento podem ser implementados em diferentes sistemas e paradigmas de representação.
- *Compromissos ontológicos mínimos*: O conjunto de compromissos ontológicos de uma ontologia deve ser o menor possível, capaz de suportar as atividades planejadas de compartilhamento de conhecimento. Uma ontologia deve fazer tão poucas imposições quanto possível sobre o mundo que está sendo modelado, permitindo que as partes comprometidas com a ontologia fiquem livres para especializar e instanciar a ontologia na medida do necessário. Uma vez que compromissos ontológicos são baseados no uso consistente de um vocabulário, eles podem ser minimizados através da especificação de uma teoria mais fraca (que admita um maior número de modelos), contendo definições restritas apenas para os termos essenciais à comunicação consistente do conhecimento da teoria.

No contexto do Projeto TOVE (*TOronto Virtual Enterprise*), foram propostos outros critérios, com ênfase sobre a *competência* da ontologia (FOX et al., 1993). A competência diz respeito a quão bem a ontologia apoia a resolução de problemas, isto é, que questões a ontologia pode responder ou que tarefas ela pode suportar. Questões

de competência devem ser definidas na fase de especificação da ontologia e utilizadas para avaliar se a ontologia responde às questões para as quais está sendo projetada. Este critério é especialmente importante, pois permite realizar uma avaliação formal de uma ontologia.

A escolha deste conjunto de critérios de qualidade para o projeto de ontologias é bastante interessante, dada a sua ortogonalidade. Ao contrário do que acontece normalmente com a maioria dos conjuntos de critérios de projeto, estes critérios não estão inerentemente em desacordo. Ao tentarmos melhorar a qualidade da ontologia com base em um critério, não estamos prejudicando-a com base nos demais critérios.

Ao buscar clareza, p.ex., definições devem ser completas e objetivas, de modo a restringir as possíveis interpretações dos termos. Por outro lado, minimizar compromissos ontológicos significa especificar uma teoria fraca, admitindo muitos modelos. Apesar de parecer à primeira vista, estas duas metas não estão em oposição. O critério de clareza versa sobre as definições de termos, enquanto os compromissos ontológicos dizem respeito a uma conceituação que está sendo descrita. Uma vez decidido que uma distinção é uma caracterização valiosa, deve-se dar à mesma a definição mais completa possível.

Vale a pena ressaltar a importância de se ter compromissos ontológicos mínimos, já que este critério é a base para o compartilhamento e reuso de conhecimento. Quanto menos compromissos uma ontologia embutir, mais passível de reutilização ela será e, neste contexto, as abordagens de ontologias em níveis e micro-teorias são ferramentas importantes. Inicialmente, constrói-se ontologias centrais, abrangendo apenas os compromissos ontológicos básicos e, na seqüência, uma dissertação mais formal e restrita do domínio pode ser definida em uma ontologia de nível mais alto ou em uma micro-teoria. Contudo, deve-se tomar cuidado para não sub-comprometer a ontologia, pois este outro extremo diminui o suporte e a orientação providos. De fato, deve-se buscar o equilíbrio entre super e sub-comprometimento. Neste sentido, uma biblioteca de ontologias com uma organização modular é fundamental. Com ela, compromissos ontológicos podem ser adicionados incrementalmente em vários níveis de ontologias.

Um último aspecto a ser considerado sobre a construção de ontologias diz respeito à escolha de uma linguagem para expressá-las. A princípio, qualquer linguagem de representação de conhecimento formal, ou até mesmo informal, pode ser usada para expressar ontologias. Na prática, no entanto, apenas algumas poucas linguagens têm sido utilizadas para esse fim, entre elas (VALENTE, 1995):

- Lógica de Primeira Ordem: é comumente usada por ser uma linguagem geral, bem conhecida e expressiva, e por adicionar relativamente poucos compromissos ontológicos. Uma ontologia expressa em lógica é a declaração de uma teoria lógica.
- KIF (*Knowledge Interchange Format*) (GRUBER, 1992): é uma linguagem formal construída para trabalhar como um meio de comunicação de conhecimento entre bases construídas usando diferentes linguagens. KIF é basicamente uma notação prefixa para lógica de predicados de primeira ordem com termos funcionais e igualdade, em cima da qual várias ontologias adicionais (de conjuntos, números, seqüências, etc.) foram construídas.
- Ontolingua (GRUBER, 1992): é uma linguagem formal e um sistema projetado para o propósito específico de expressar ontologias. Ontolingua foi construída sobre KIF, adicionando mecanismos para expressar classes, relações e hierarquias de classe.
- CML (*Conceptual Modelling Language*) (BREUKER et al., 1994): é uma linguagem semi-formal proposta como um formalismo de representação dentro de CommonKADS. CML é largamente inspirada em KL-ONE (BRACHMAN et al., 1985), com construções adicionais para expressar tarefas, inferências e conhecimento de resolução de problema, de acordo com a infra-estrutura epistemológica adotada por KADS.
- *Description Logic* (RUSSELL et al., 1995): é uma lógica projetada para enfocar categorias e suas definições. Seus principais mecanismos de inferência visam verificar se uma categoria é um sub-conjunto de outra, ou se um objeto pertence a uma categoria.

Cada uma dessas linguagens embute um número de compromissos ontológicos e a escolha de uma particular linguagem deve ser feita com base na sua adequação aos propósitos de representação da ontologia.

3.3.4 - Métodos para a Construção de Ontologias

Até o momento, uma grande quantidade de ontologias foi desenvolvida por diferentes grupos, sob diferentes abordagens, e usando diferentes métodos e técnicas. Entretanto, poucos trabalhos foram publicados sobre como proceder, mostrando as práticas, critérios de projeto, atividades, métodos e ferramentas usadas para construir ontologias. A conseqüência é clara: a ausência de atividades padronizadas, ciclos de vida e métodos sistemáticos, assim como de um conjunto de critérios de qualidade, técnicas e ferramentas, tornam o desenvolvimento de ontologias uma arte ao invés de uma atividade de engenharia. A arte só tornar-se-á engenharia quando houver uma definição e padronização de um ciclo de vida, abrangendo desde a definição de requisitos até a manutenção do produto acabado, bem como métodos e técnicas que conduzam o desenvolvimento (FERNÁNDEZ et al., 1997).

Poder-se-ia pensar que os métodos adotados na construção de SBCs poderiam ser facilmente adaptados para a construção de ontologias. Entretanto, isto não é verdade. Um dos principais problemas que os engenheiros de conhecimento enfrentam na construção de SBCs é a dificuldade de obter um conjunto de requisitos para o sistema. Uma vez que especialistas geralmente não são capazes de descrever de forma concreta e completa como se comportam no domínio de aplicação, é difícil para os engenheiros de conhecimento especificar o futuro comportamento do SBC. Assim, SBCs são geralmente construídos incrementalmente, usando protótipos evolutivos, com as deficiências do produto final de cada ciclo sendo utilizadas como a especificação para o próximo protótipo. Em ontologias, o mesmo não ocorre. Ontologias são construídas para serem reutilizadas ou compartilhadas. Desta forma, é necessário especificar, pelo menos parcialmente, uma grande porção do vocabulário de domínio que a ontologia se propõe a cobrir. Esta é a principal diferença entre o processo de desenvolvimento de ontologias e de SBCs. Conseqüentemente,

metodologias usadas para construir SBCs não podem ser totalmente aplicadas à construção de ontologias (FERNÁNDEZ et al., 1997).

À medida que ontologias têm merecido maior atenção da comunidade de SBCs, alguns métodos para sua construção têm sido propostos, como em (USCHOLD et al., 1995), (FERNÁNDEZ et al., 1997) e (GRÜNINGER et al., 1995).

USCHOLD et al. (1995) propuseram o que eles mesmos chamaram de uma "metodologia inicial para a construção de ontologias", definindo um pequeno número de estágios necessários para qualquer futura metodologia mais ampla. Segundo eles, uma metodologia para o desenvolvimento de ontologias deve incluir os seguintes estágios, cada um deles associado a um conjunto de técnicas, métodos, princípios e diretrizes para sua realização:

- *Identificação do Propósito*: É importante saber claramente *porque* uma ontologia está sendo construída, quais são seus usos projetados e os seus potenciais usuários.
- *Construção da Ontologia*: Envolve três passos principais: captura, codificação e integração com ontologias existentes. A captura da ontologia envolve a identificação dos conceitos e relações relevantes no domínio de interesse, a geração de definições textuais precisas para estes elementos e o estabelecimento de termos para referenciá-los. Na codificação, a conceituação capturada no estágio anterior é representada em alguma linguagem formal. Durante os passos de captura e codificação, é possível que ontologias existentes sejam reutilizadas e, portanto, é necessário integrá-las.
- *Avaliação*: Uma ontologia deve ser avaliada em termos de questões de competência, especificações de requisitos e/ou do mundo real.
- *Documentação*: Todas as decisões importantes devem ser documentadas, tanto no que tange aos principais conceitos definidos na ontologia, como no que diz respeito às primitivas usadas para expressar definições na ontologia, i.e., a meta-ontologia.

FERNÁNDEZ et al. (1997) propuseram METHONTOLOGY, um método estruturado para a construção de ontologias. O primeiro passo deste método é a *especificação*, cujo objetivo é produzir um documento de especificação da ontologia,

contendo: (i) o propósito da ontologia, incluindo seus usos projetados, cenários de uso e potenciais usuários, (ii) o nível de formalidade da ontologia a ser implementada, e (iii) o escopo, o que inclui um conjunto de termos a serem representados, suas características e granularidade.

O segundo passo, a *conceituação*, é o mais trabalhoso pois envolve a elaboração de várias representações intermediárias utilizadas para conceituar o domínio, tais como, Dicionário de Dados, Árvores de Classificação de Conceitos, Tabela de Constantes, Tabelas de Atributo de Instância e de Classe, Tabelas de Fórmulas, Árvores de Classificação de Atributos e Tabelas de Instâncias.

Com o objetivo de acelerar a construção de uma ontologia, o método propõe que se considere o reuso de definições previamente feitas em outras ontologias, inspecionando-se meta-ontologias e ontologias de biblioteca que provejam definições cuja semântica e implementação sejam coerentes com os termos identificados na conceituação.

A fase que se segue é a *implementação*, cujo resultado é a ontologia codificada em uma linguagem formal. Finalmente, a ontologia deve ser avaliada. A *avaliação* consiste em realizar um julgamento técnico da ontologia com respeito ao documento de especificação de requisitos e, na verdade, deve ocorrer durante cada fase e entre as fases do ciclo de vida.

Uma farta documentação é gerada em METHONTOLOGY, incluindo documento de especificação de requisitos, documento de aquisição de conhecimento, modelo conceitual, documento de formalização, documento de integração, documento de implementação e documento de avaliação.

Dentro do contexto do Projeto TOVE (*TOronto Virtual Enterprise*), definiu-se uma metodologia para projeto e avaliação de ontologias (GRÜNINGER et al., 1994) (GRÜNINGER et al., 1995), contendo, basicamente, os seguintes passos:

- *Descrição dos cenários de motivação*: O desenvolvimento de uma ontologia é sempre motivado por algum conjunto de problemas no mundo real que precisa ser modelado e resolvido. Um cenário de motivação provê um conjunto de soluções intuitivamente possíveis para estes problemas, fornecendo uma

primeira idéia da semântica planejada para os objetos e relações a serem incluídos na ontologia.

- *Definição de questões informais de competência*: Dado o cenário de motivação, um conjunto de *questões de competência* deve ser elaborado. Estas questões são, na realidade, os requisitos da ontologia e devem ser usadas para avaliar se os compromissos da ontologia são necessários e suficientes para caracterizar os problemas dos cenários de motivação e suas soluções.
- *Especificação da terminologia em lógica de primeira ordem*: Uma vez que as questões de competência foram estabelecidas, deve-se especificar a terminologia da ontologia, usando lógica de primeira ordem. Esta descrição formal dos objetos, suas propriedades e relações, provê uma linguagem a ser usada para expressar as definições e restrições nos axiomas.
- *Definição formal das questões de competência*: Uma vez que se tenha as questões informais de competência e a terminologia da ontologia, o próximo passo consiste em definir formalmente as questões de competência.
- *Especificação de axiomas em lógica de primeira ordem*: Nesta etapa, axiomas são definidos como sentenças de primeira ordem usando os predicados da ontologia.
- *Avaliação da ontologia*: Com base nas questões formais de competência, a ontologia deve ser avaliada. Podem existir diferentes formas de se axiomatizar uma ontologia e, portanto, as questões de competência devem ser usadas para avaliar a completeza do conjunto de axiomas em uma particular axiomatização.

Considerações sobre os Métodos Apresentados

Analisando-se os métodos apresentados e outros trabalhos onde ontologias foram construídas, tal como (VALENTE, 1995), é possível delinear uma abordagem sistemática para a engenharia de ontologias. Basicamente, o processo de construção de ontologias envolve os seguintes passos: (i) identificação de propósito e especificação dos requisitos da ontologia, (ii) captura, (iii) formalização, (iv) integração com ontologias existentes, (v) avaliação e (vi) documentação. Estes passos, de maneira geral, estão presentes nos métodos apresentados, ainda que diferentes abordagens e instrumentos sejam utilizados.

METHONTOLOGY emprega diversos instrumentos, tais como Dicionários de Dados, Árvores de Classificação e tabelas diversas. Entretanto, alguns desses instrumentos parecem ser adequados apenas a domínios específicos, tal como a Tabela de Fórmulas, que encontra espaço na construção de ontologias no domínio bioquímico (GÓMEZ-PÉREZ et al., 1996). Além disso, não considera explicitamente a axiomatização da ontologia.

A metodologia de TOVE também apresenta características bastante peculiares ao contexto na qual está inserida, a modelagem de empreendimento. De fato, muitas das idéias sobre ontologias não são incorporadas por ela, o que a torna uma metodologia aplicada e não geral. No entanto, emprega um mecanismo bastante eficaz para avaliação da qualidade, através das questões de competência, que deve ser incorporado a outras propostas.

É desejável reunir em uma única proposta, as melhores características dos métodos apresentados. Com esta finalidade, na seção 3.4, propomos um método para a construção de ontologias, incorporando o que julgamos ser as principais idéias sobre construção de ontologias.

3.3.5 - Vantagens e Problemas no Uso de Ontologias

Os SBCs atuais são monólitos isolados, caracterizados por um alto acoplamento interno e a falta de interfaces externas. O único meio prático de compartilhar ou reutilizar uma base de conhecimento existente é adotar o conjunto completo composto por sua representação e pelo ambiente de programação no qual foi implementada.

Usando ontologias, o esforço da aquisição de conhecimento pode ser dividido em duas partes. Na primeira etapa, especificações explícitas da conceituação básica do domínio são criadas na forma de ontologias, tendo como foco o conhecimento geral sobre o domínio, comum a um conjunto amplo de aplicações. Na segunda etapa, o conhecimento específico de uma aplicação é capturado e codificado em um SBC. Esta fase é fortemente guiada pelas ontologias, já que estas provêem ao engenheiro de conhecimento um vocabulário para se falar sobre o domínio, dado pelos termos das ontologias, e um núcleo de conhecimento, dado por seus axiomas. Esta é a principal

vantagem de se utilizar ontologias no desenvolvimento de SBCs: ao se dividir a aquisição de conhecimento em duas etapas, tem-se um meio prático e objetivo de reutilizar / compartilhar bases de conhecimento e são oferecidas valiosas diretrizes para orientar a aquisição do conhecimento específico de uma aplicação.

Uma ontologia captura o conhecimento relevante em um universo de discurso, incorporando um conjunto de compromissos ontológicos em um formato que permite a construção de aplicações e bases de conhecimento fundamentadas nesses compromissos. Quando os termos definidos em uma ontologia são instanciados em novas aplicações, o esforço da aquisição de conhecimento é distribuído. De fato, em se tratando de reuso de conhecimento, o poder maior não deriva do reuso de bases de conhecimento completas, preenchidas com milhões de fatos sobre um amplo domínio de discurso. SBCs irão sempre requerer conhecimento específico da aplicação e nenhum esforço é capaz de antecipar todo o conteúdo possível. A força advém do uso de ontologias no projeto e construção de bases de conhecimento similares para diferentes aplicações e instituições. Ontologias explícitas e públicas favorecem a integração de ferramentas projetadas para operar sobre conhecimento de domínio, através de um vocabulário harmônico e bem definido (GRUBER, 1995).

Ao se construir pequenas bases modulares de conhecimento, fundamentadas em ontologias, está-se provendo uma abordagem sistemática para reuso e compartilhamento de conhecimento. Se uma aplicação compromete-se com uma ontologia, a base de conhecimento correspondente deve ser incorporada diretamente à base de conhecimento da aplicação.

Finalmente, ontologias são úteis para compreensão e interação entre pessoas, servindo, por exemplo, como referência do consenso obtido por uma comunidade profissional sobre o vocabulário técnico a ser usado em suas interações.

Entretanto, o uso de ontologias também apresenta problemas. O'LEARY (1997), por exemplo, identificou os seguintes impedimentos: (i) A escolha de uma ontologia é um processo político, já que nenhuma ontologia pode ser totalmente adequada a todos os indivíduos ou grupos. (ii) Ontologias não são necessariamente estacionárias, i.e., necessitam evoluir. Poucos trabalhos têm enfocado a evolução de ontologias. (iii) Estender ontologias não é um processo direto. Ontologias são,

geralmente, estruturadas de maneira precisa e, como resultado, são particularmente vulneráveis a questões de extensão, dado o forte relacionamento entre complexidade e precisão das definições. (iv) A noção de bibliotecas de ontologias sugere uma relativa independência entre diferentes ontologias. A interface entre elas constitui, portanto, um impedimento, especialmente porque cada uma delas é desenvolvida no contexto de um processo político. Ontologias desenvolvidas independentemente podem não se integrar efetivamente com outras por vários motivos, desde similaridade de vocabulário até visões conflitantes do mundo.

O formato no qual ontologias são distribuídas representa também um obstáculo para o uso de ontologias. Segundo SWARTOUT et al. (1997), muitas ontologias são distribuídas em um formato de código fonte na linguagem de representação, o que não permite que usuários naveguem através da ontologia para compreender seu escopo, estrutura e conteúdo. Além disso, a falta de tradutores entre linguagens representa outro sério obstáculo.

Finalmente, não há ainda um consenso quanto a avaliação da qualidade de ontologias. Apesar de existirem indicações de critérios a serem adotados, não há métricas e procedimentos estabelecidos para a avaliação da qualidade de ontologias. Assim, torna-se bastante difícil assegurar que uma biblioteca de ontologias é capaz de aumentar a produtividade no desenvolvimento de SBCs.

3.4 - Um Passo à Frente na Construção e Uso de Ontologias

A construção e o uso de ontologias no desenvolvimento de Sistemas Baseados em Conhecimento é uma área de estudos em aberto. Muitos trabalhos têm sido realizados nesta área, mas ainda há muito a ser feito. O objetivo desta seção é avançar no sentido de aprimorar o processo de construção de ontologias e seu uso no desenvolvimento de aplicações. Basicamente, enfocamos três aspectos importantes:

- *o uso de uma notação gráfica para descrever ontologias*: É reconhecido o poder de comunicação das linguagens gráficas. Baseados nesta realidade, na seção 3.4.1, definimos LINGO, uma linguagem gráfica para descrever ontologias;

- *um método para a construção de ontologias*: Embora existam alguns métodos descritos na literatura que apresentam valiosas contribuições, conforme discutimos na seção 3.3.4, julgamos que não há, ainda, uma proposta que possa ser considerada completa e amplamente aplicável. Neste sentido, definimos uma abordagem sistemática para o desenvolvimento de ontologias, incorporando as principais características dos vários métodos existentes (seção 3.4.2);
- *uma abordagem de engenharia de conhecimento*: Uma vez construídas as ontologias, uma importante questão surge: como utilizá-las no desenvolvimento de aplicações? Ainda que algumas metodologias para o desenvolvimento de SBCs possam ser adaptadas ou estendidas para tratar esta questão, optamos por definir uma abordagem de engenharia de conhecimento centrada no uso de ontologias (seção 3.4.3).

3.4.1 - LINGO - LINguagem Gráfica para descrever Ontologias

Na aquisição de conhecimento, o uso de uma representação gráfica é fundamental, pois age como um elemento facilitador da comunicação entre engenheiros de conhecimento e especialistas. Na construção de ontologias, tal representação é basicamente uma linguagem para se falar de ontologias e, portanto, incorpora uma meta-ontologia. Nesta seção definimos LINGO, uma LINguagem Gráfica para descrever Ontologias.

Antes de apresentarmos LINGO, contudo, faz-se necessário responder uma questão: Por que criar uma representação gráfica específica para expressar ontologias, se há representações que poderiam ser adaptadas para este fim, tais como a linguagem de modelagem de especialidade de KADS, uma linguagem de modelagem de objetos ou uma representação de entidades e relacionamentos? A resposta é simples: porque não desejamos incorporar na descrição de uma ontologia a semântica que tais linguagens embutem. Uma vez que as linguagens citadas anteriormente são bastante representativas do universo de linguagens gráficas passíveis de adaptação para descrever ontologias, vamos analisá-las brevemente.

A linguagem gráfica para a modelagem de especialidade de KADS (SCHREIBER et al., 1993) foi proposta para suportar o desenvolvimento de SBCs e, para tal, possui representações que entendemos serem desnecessárias para ontologias, tais como notações para conjuntos e expressões. De fato, ao se utilizar tais representações, está-se embutindo um conjunto de compromissos ontológicos, dados por uma meta-ontologia mais específica.

No contexto do Projeto GAMES (HEIJST, 1995), uma notação gráfica é utilizada para representar ontologias. Contudo, uma vez que Ontolingua é usada para descrever essas ontologias, a notação proposta adota a ontologia de Ontolingua como uma meta-ontologia e, portanto, embute seus compromissos ontológicos.

Em um modelo de objetos, objetos representam abstrações do mundo real que possuem estado (dado por seus atributos), comportamento (dado por seus métodos) e identidade própria. Classes, por sua vez, agrupam objetos que possuem os mesmos atributos e relacionamentos e exibem o mesmo comportamento. Em se tratando de uma ontologia, estamos preocupados em descrever uma conceituação e, portanto, lidamos essencialmente com conceitos e relações. Poder-se-ia argumentar que conceitos poderiam ser descritos como classes em um modelo de objetos, mas há diferenças fundamentais: objetos em uma classe exibem um comportamento dado pelos métodos da classe; para conceitos em uma ontologia, isto não faz sentido; geralmente, classes em um modelo orientado a objetos possuem atributos[1], em ontologias, ainda que conceitos possam apresentar atributos, esta não é uma característica obrigatória. Este aspecto é uma razão também para não adotarmos um modelo de entidades e relacionamentos para descrever ontologias, ainda que alguns métodos, como o proposto no âmbito do Projeto TOVE, o façam.

Além das razões anteriormente expostas, uma vez que um traço marcante de ontologias são os seus axiomas, introduzimos em LINGO apenas aquelas notações capazes de capturar certos axiomas de forma implícita. Ou seja, uma notação gráfica de LINGO deve possuir uma semântica equivalente a um conjunto de axiomas, de modo que, ao utilizar tal notação estejamos, de fato, descrevendo o conjunto de axiomas que ela representa.

[1] Alguns métodos orientados a objetos, como o proposto por COAD et al. (1992), utilizam como critério para inclusão de classes em um modelo, o fato da classe possuir mais do que um atributo.

Em se tratando de ontologias, uma linguagem gráfica deve possuir primitivas básicas capazes de representar a conceituação de um domínio. Neste sentido, em sua forma mais simples, LINGO possui primitivas para representar apenas conceitos e relações, cujas notações estão mostradas na figura 3.5.

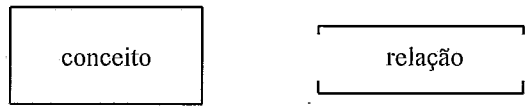


Figura 3.5 - Notações Utilizadas para Conceitos e Relações.

A figura 3.6 mostra um exemplo de uma relação binária entre os conceitos de atividade e produto, em um contexto de manufatura, indicando que atividades geram produtos. Cardinalidades são usadas para mostrar quantas instâncias de um conceito podem participar da relação. Uma vez que a cardinalidade (0..n) não impõe nenhum axioma, ela não é representada.

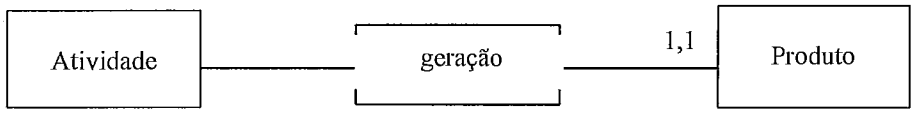


Figura 3.6 - Exemplo de uma Relação Binária entre Conceitos.

O uso da cardinalidade (1,1) na figura 3.6, por exemplo, induz os seguintes axiomas:

$$(\forall\, p)\ (produto(p) \rightarrow (\exists\, a)\ (geração(p,a)) \qquad \text{(A1)}$$

$$(\forall\, p, a_1, a_2)\ (geração(p,a_1) \wedge geração(p,a_2) \rightarrow a_1 = a_2) \qquad \text{(A2)}$$

Axiomas da forma do axioma (A1) são um reflexo da cardinalidade mínima 1. Axiomas da forma de (A2), por outro lado, são reflexo da cardinalidade máxima 1.

Apesar do exemplo da figura 3.6, não se deve pensar que as relações estão restritas a relações binárias. Relações de ordem superior, tais como relações ternárias, são igualmente válidas. Além disso, relações entre instâncias de um mesmo conceito também são válidas. Neste caso, sugerimos o uso de papéis, como mostra a figura 3.7.

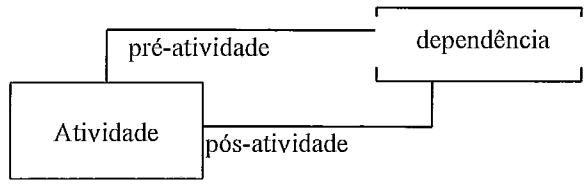


Figura 3.7 - Uso de Papéis.

Neste caso, o seguinte axioma pode ser derivado:

$$(\forall\ a_1,a_2)\ (preatividade(a_1,a_2) \leftrightarrow posatividade(a_2,a_1) \qquad \text{(A3)}$$

Alguns tipos de associações têm uma semântica forte e, na verdade, escondem por detrás uma ontologia genérica, tal como a relação de composição. Para estes tipos de associações, uma notação especializada é proposta. De fato, esta é a característica marcante de LINGO e que a faz diferente de outras representações gráficas: qualquer notação proposta além das notações básicas para conceitos e relações visa incorporar uma teoria. Uma vez que a teoria é incorporada, axiomas podem ser automaticamente gerados. Estes axiomas dizem respeito simplesmente à estruturação dos conceitos e são ditos *axiomas epistemológicos*. Assim, ainda que LINGO seja uma representação de nível epistemológico, ela incorpora um mecanismo de inclusão de teorias no nível ontológico.

Na versão corrente de LINGO, apenas dois tipos especiais de associações foram consideradas: a relação todo-parte e a associação sub-tipo-de. Quaisquer que sejam os elementos envolvidos em uma relação todo-parte, há um conjunto de propriedades que são sempre válidas, entre elas:

$$(\forall\ x,y)\ (partede(x,y) \rightarrow \neg\ partede(y,x)) \qquad \text{(A4)}$$

$$(\forall\ x,y,z)\ (partede(x,y) \wedge partede(y,z) \rightarrow\ partede(x,z)) \qquad \text{(A5)}$$

$$(\forall\ x,y)\ (disjunto(x,y) \leftrightarrow \neg\ (\exists\ z)\ (partede(z,x) \wedge partede(z,y))) \qquad \text{(A6)}$$

$$(\forall\ x)\ (atômico(x) \leftrightarrow \neg\ (\exists y)\ (partede(y,x)) \qquad \text{(A7)}$$

Ao utilizar uma relação todo-parte, estamos importando e aplicando uma teoria abstrata de composição de elementos ao conteúdo da ontologia em desenvolvimento. Assim, é importante atribuir uma notação especial para este tipo de relação, de modo que, ao utilizarmos tal representação, estejamos implicitamente incluindo a ontologia

genérica que o especifica. A figura 3.8 mostra a notação empregada para representar relações de composição.

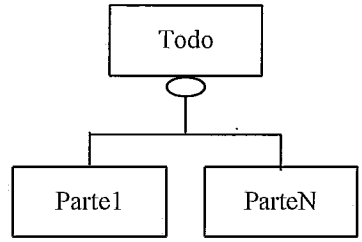


Figura 3.8 - Notação para Composição.

Uma vez que a relação de composição é uma relação entre instâncias de conceitos, cardinalidades devem ser representadas. Para representar este tipo de relação, utilizamos uma linha cheia com uma pequena elipse junto ao todo.

Um tipo de associação que também merece atenção especial é a associação "sub-tipo-de". Ao construirmos uma taxonomia de conceitos, estamos nos comprometendo implicitamente com uma ontologia de sub-tipos, que nos diz, entre outras coisas, que:

$$(\forall\ x, y)\ (subtipo(x, y) \rightarrow supertipo(y, x)) \qquad (A8)$$

Para representar este tipo de associação, usamos a notação mostrada na figura 3.9.

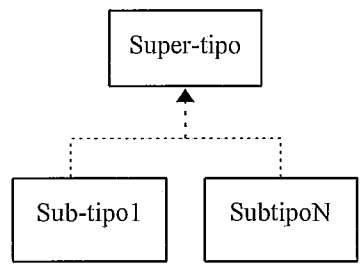


Figura 3.9 - Notação para Hierarquia de Conceitos.

Uma vez que a associação sub-tipo-de se dá entre conceitos e não entre suas instâncias, utilizamos uma linha pontilhada para representá-la, com uma seta apontando para o super-tipo.

O uso de notações especiais para realçar diferentes tipos de associações pode ser uma forma bastante direta de se integrar ontologias genéricas com ontologias de domínio. Um ambiente de desenvolvimento de ontologias que adotasse esta filosofia

embutiria um poderoso mecanismo para inclusão de teorias. Cada tipo de associação especificando uma teoria genérica ganharia sua própria notação e sempre que esta fosse utilizada, estar-se-ia integrando ontologias automaticamente. Neste sentido, LINGO não deve ser considerada uma representação estática e acabada: quando uma teoria versando sobre um determinado tipo de associação for identificada, uma notação específica pode ser incluída. Esta abordagem tem muitos pontos em comum com o uso de *projeções de ontologias* como um mecanismo para facilitar a construção de grandes ontologias a partir de outras menores, proposto por BORST et al. (1997).

Além disso, LINGO pode também ser usada para expressar condicionantes entre relações. Tomemos o seguinte caso: dados três conceitos *A*, *B* e *C* e duas relações *r1* e *r2*, entre instâncias dos conceitos *A* e instâncias dos conceitos *B* e *C*, respectivamente, queremos expressar uma condicionante entre as relações, dizendo que se uma instância de *A* está relacionada com uma instância de *B*, então ela não pode estar relacionada com uma instância de *C*. Para tal, a seguinte notação foi proposta:

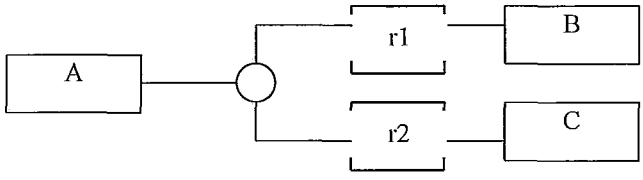


Figura 3.10 - Condicionante Ou-exclusivo entre Relações.

Ao utilizarmos a notação da figura 3.10, estamos assumindo que:

$(\forall\ a)\ ((a \in A) \wedge ((\exists\ b)\ (b \in B) \wedge r1(a,b)) \rightarrow \neg\ ((\exists\ c)\ (c \in C) \wedge r2(a,c)))$

$(\forall\ a)\ ((a \in A) \wedge ((\exists\ c)\ (c \in C) \wedge r2(a,c)) \rightarrow \neg\ ((\exists\ b)\ (b \in B) \wedge r1(a,b)))$

Analogamente, introduzimos a notação da figura 3.11 para estabelecer uma condicionante de obrigatoriedade entre relações: se uma instância de *A* está relacionada com uma instância de *B*, então ela obrigatoriamente tem de estar relacionada com uma instância de *C*:

$(\forall\ a)\ ((a \in A) \wedge ((\exists\ b)\ (b \in B) \wedge r1(a,b)) \rightarrow ((\exists\ c)\ (c \in C) \wedge r2(a,c)))$

$(\forall\ a)\ ((a \in A) \wedge ((\exists\ c)\ (c \in C) \wedge r2(a,c)) \rightarrow ((\exists\ b)\ (b \in B) \wedge r1(a,b)))$

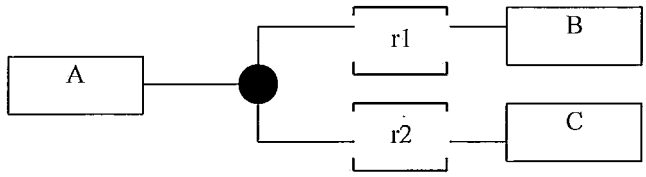


Figura 3.11 - Condicionante de Obrigatoriedade entre Relações.

Notações para outros tipos de condicionantes podem ser incorporadas a LINGO e, novamente, a proposta aqui apresentada não deve ser considerada definitiva. Ao contrário, LINGO deve ser considerada uma linguagem aberta, suscetível a extensões para capturar outras necessidades da modelagem no nível ontológico.

3.4.2 - Engenharia de Ontologias - uma Abordagem Sistemática

O objetivo desta seção é dar um passo à frente no sentido de transformar a construção de ontologias de arte em engenharia. As várias atividades do processo de construção de ontologias são discutidas, sendo apresentadas algumas orientações de como proceder na sua realização. Além disso, é proposto um ciclo de vida, mostrando as interações entre as várias atividades.

Identificação de Propósito e Especificação de Requisitos

A primeira atividade a ser realizada no processo de construção de uma ontologia é identificar claramente o seu propósito e os usos esperados para ela, i.e., a competência da ontologia. A competência de uma representação diz respeito à cobertura de questões que essa representação pode responder ou de tarefas que ela pode suportar (FOX et al., 1993). Ao se estabelecer a competência, temos um meio eficaz de delimitar o que é relevante para a ontologia e o que não é. É útil, também, identificar potenciais usuários e os cenários que motivaram o desenvolvimento da ontologia em questão.

Uma vez definido o propósito, deve-se especificar os requisitos da ontologia. Esses devem contemplar os usos projetados para a ontologia e podem ser expressos em termos de questões de competência: as questões que a ontologia deve ser capaz de responder (GRÜNINGER et al., 1995). Ao se especificar um relacionamento entre as

questões de competência e os cenários de motivação, está se dando uma justificativa para a ontologia e, mais importante, se está provendo um mecanismo para sua avaliação.

É importante realçar que as questões de competência não geram compromissos ontológicos; ao contrário, são usadas para avaliar os compromissos de uma ontologia, avaliando sua expressividade (GRÜNINGER et al., 1995).

Se o domínio de interesse for muito complexo, deve-se utilizar algum mecanismo de decomposição, de modo a melhor distribuir esta complexidade. Uma abordagem potencialmente interessante é considerar ontologias em níveis, conforme discutido na seção 3.3.3. Para se efetuar a decomposição do domínio, é necessário esboçar a ontologia e, portanto, uma aquisição inicial do conhecimento deve ser realizada nesta fase. Uma vez decomposto o domínio, deve-se definir uma ontologia para cada sub-domínio.

Captura da Ontologia

Esta é, sem dúvida, a etapa mais importante no desenvolvimento de uma ontologia. O objetivo é capturar a conceituação do universo de discurso, com base na competência da ontologia. Os conceitos e relações relevantes devem ser identificados e organizados. Um modelo utilizando uma linguagem gráfica, tal como LINGO, pode ser um instrumento-chave para facilitar a comunicação com os especialistas do domínio. Este modelo deve ser acompanhado de um dicionário de termos.

Conceitos primitivos, isto é, aqueles que não são passíveis de uma definição em termos de outros conceitos da ontologia, devem ser definidos utilizando linguagem natural e exemplos, tomando o devido cuidado para se evitar ambigüidades e inconsistências. A escolha dos termos a serem usados para referenciar as categorias de conhecimento deve ser feita cuidadosamente, evitando-se termos com interpretação duvidosa. Conceitos passíveis de descrição em termos de outros conceitos, devem ser definidos com referências claras a estes, com o objetivo de facilitar a formalização. Deve-se ainda construir taxonomias, organizando categorias e sub-categorias interconectadas do conhecimento do domínio de interesse.

Os conceitos e relações formam a base da ontologia. Mas uma característica essencial de ontologias é a definição de axiomas. Simplesmente propor uma

taxonomia ou um conjunto de termos básicos, não constitui uma ontologia. Axiomas devem ser providos para definir a semântica dos termos. Os axiomas especificam definições de termos na ontologia e restrições sobre sua interpretação. Neste momento, não há necessidade de se escrever axiomas formais, mas ao contrário, estes devem ser descritos em linguagem natural, refletindo simplesmente as restrições existentes sobre o universo de discurso.

Os axiomas em uma ontologia podem apresentar duas formas e propósitos diferentes: axiomas de derivação e axiomas de consolidação. Axiomas de derivação são aqueles que permitem explicitar informações a partir do conhecimento previamente existente. Assim, são meios para a dedução e representam conseqüências lógicas neste processo. Axiomas de consolidação, por sua vez, não são utilizados para derivar informação, mas apenas para descrever a coerência das informações existentes. Neste sentido, não representam conseqüências lógicas. Tipicamente, os axiomas de consolidação definem condicionantes para o estabelecimento de uma relação ou para a definição de um objeto como instância de um conceito.

Os axiomas de derivação podem ter origem no significado dos conceitos e relações da ontologia ou na forma como são estruturados. Quando axiomas são descritos para mostrar restrições impostas pela forma de estruturação dos conceitos, eles são ditos axiomas epistemológicos. Quando descrevem restrições de significação impostas no domínio, são ditos axiomas ontológicos.

Esta classificação quanto à natureza dos axiomas é uma boa diretriz para guiar a definição dos axiomas de uma ontologia, ou seja, devemos estar atentos para capturar axiomas que considerem a estruturação dos conceitos e relações (os axiomas epistemológicos), seus significados e restrições (os axiomas ontológicos) e as leis de integridade que os regem (os axiomas de consolidação).

O processo de definição de axiomas é, talvez, o aspecto mais difícil na construção de ontologias. Entretanto, esse processo pode ser guiado pelas questões de competência. Os axiomas devem ser necessários e suficientes para expressar as questões de competência e para caracterizar suas soluções. Além disso, qualquer solução para uma questão de competência deve ser descrita pelos axiomas da ontologia e deve ser consistente com eles. Se os axiomas propostos não forem

suficientes para esse propósito, então, conceitos, relações ou axiomas adicionais devem ser introduzidos na ontologia. Neste sentido, a captura de uma ontologia é um processo iterativo e fortemente ligado à avaliação. Podem existir diferentes formas de se axiomatizar uma ontologia, sendo que as questões de competência devem ser usadas para avaliar a completeza do conjunto de axiomas em uma axiomatização particular (GRÜNINGER et al., 1994).

Os critérios de qualidade, discutidos na seção 3.3.3, devem ser observados, sobretudo: clareza, no que se refere à definição dos termos no dicionário; coerência, entre as definições textuais e os exemplos; e comprometimento ontológico mínimo, para evitar que o campo de ação da ontologia seja restrito. Este último critério está diretamente relacionado à definição dos axiomas.

Formalização da Ontologia

A validação de uma teoria sobre um universo de discurso é, sem dúvida, melhor realizada quando esta é descrita em uma linguagem formal. Nesta linguagem, em contraste com a linguagem natural, tem-se símbolos não ambíguos e formulações exatas e, portanto, a clareza e a correção de uma dedução podem ser testadas com maior facilidade e precisão. Uma dedução em linguagem natural, geralmente, envolve pressuposições implícitas que entram desapercebidas no processo de dedução (CARNAP, 1958).

O tratamento teórico de qualquer domínio consiste em propor sentenças sobre os objetos neste domínio (sentenças atribuindo certas propriedades e relações aos objetos em questão) e em estabelecer regras de acordo com as quais outras sentenças possam ser derivadas a partir das sentenças dadas (CARNAP, 1958).

Nesta etapa, devemos estabelecer um formalismo para representar as categorias de conhecimento da ontologia. Primeiro, devemos olhar entre os formalismos disponíveis se não existe algum que possa ser usado diretamente ou adaptado. Caso não exista, um formalismo adequado deve ser proposto. Uma vez definido o formalismo de representação a ser usado, tem-se uma base para fixar a terminologia da ontologia e, principalmente, a sua semântica.

É importante realçar que uma ontologia formal não é capaz de substituir uma descrição de uma conceituação em linguagem natural, mas deve, sim, ser usada para

suportá-la ou para somar-se a ela, trabalhando como um dispositivo no qual algumas idéias são verificadas em relação a completeza e, talvez, coerência. Ou seja, cada representação cumpre um papel específico.

Em suma, o que se pretende é representar explicitamente a conceituação capturada no estágio anterior em uma linguagem formal, o que envolve o comprometimento com alguma meta-ontologia, a escolha de uma linguagem de representação e a criação da ontologia formal. Quando não for necessário nenhum comprometimento especial com alguma meta-ontologia que comprovadamente mostre-se adequada à ontologia em desenvolvimento, a lógica de primeira ordem tende a ser o formalismo mais adequado, haja visto que é aquele que embute o menor número de compromissos ontológicos adicionais.

Quando a lógica de primeira ordem for o formalismo adotado, o primiero passo da etapa de formalização deve ser a especificação dos símbolos não lógicos da linguagem, ou seja, as *constantes*, denotando indivíduos específicos do universo de discurso, os *símbolos funcionais*, denotando funções e os *predicados*, denotando propriedades e relações declaradas dos indivíduos. Feito isso, é possível formar sentenças sobre os indivíduos do universo de discurso, os axiomas formais.

Integração com Ontologias Existentes

Durante os processos de captura e/ou formalização, pode surgir a necessidade de integrar a ontologia em questão com outras já existentes, visando aproveitar conceituações previamente estabelecidas. De fato, é uma boa prática desenvolver ontologias funcionais modulares, que sejam gerais e mais amplamente reutilizáveis, e, quando necessário, integrá-las, obtendo o resultado desejado. Quando muitos detalhes forem necessários, a ontologia deve incorporar apenas os compromissos ontológicos essenciais, sendo os demais descritos em uma micro-teoria, ou em outra ontologia de nível mais alto. Assim, preserva-se o critério de qualidade de compromisso ontológico mínimo.

Avaliação

Finalmente, a ontologia deve ser avaliada para verificar se satisfaz os requisitos estabelecidos na especificação. Adicionalmente, ela deve ser avaliada em

relação a critérios de qualidade para o projeto de ontologias. O conjunto de critérios apresentado na seção 3.3.3 deve ser usado tanto para guiar o desenvolvimento, quanto para avaliar a qualidade das ontologias construídas.

É importante notar que esta etapa pode (e de fato, deve) ser realizada em paralelo com as etapas de captura e formalização, em um processo iterativo. Para avaliar uma ontologia junto aos especialistas do domínio, o uso de uma representação gráfica é extremamente importante, razão pela qual propôs-se o uso de LINGO. Para avaliar a completeza da ontologia, especialmente no que se refere a seus axiomas, as questões de competência têm um papel fundamental. Em um domínio particular, é possível escrever um número bastante grande de axiomas. Entretanto, surge uma questão: Como não escrever mais axiomas do que o necessário? Para responder a esta questão temos que ter em mente o princípio de comprometimento ontológico mínimo e, em mãos, as questões de competência da ontologia. O conjunto de axiomas deve ser necessário e suficiente para expressar as questões de competência e caracterizar suas soluções, e apenas isto (GRÜNINGER et al., 1995) (FOX et al., 1993). Axiomas redundantes ou que não contribuem para responder a uma questão de competência devem ser eliminados.

Documentação

Todo o desenvolvimento da ontologia deve ser documentado, incluindo propósitos, requisitos e cenários de motivação, as descrições textuais da conceituação, a ontologia formal e os critérios de projeto adotados. Assim, como a avaliação, a documentação é uma etapa que deve ocorrer em paralelo com as demais.

Os termos capturados na conceituação do universo de discurso devem ser descritos em um Dicionário de Termos, considerando dois princípios importantes: o princípio do vocabulário mínimo e o princípio da auto-referência. O princípio do vocabulário mínimo diz respeito ao vocabulário utilizado na definição dos termos da ontologia. Este vocabulário deve ser o menor possível e não deve apresentar ambigüidades. Qualquer termo que não tenha um significado claro e não ambíguo, deve ser definido como uma entrada no Dicionário. O princípio da auto-referência indica que a definição de um termo no Dicionário deve, sempre que possível, ser feita utilizando outros termos do Dicionário. Com base neste princípio, o uso de

hipertextos surge como uma potencial abordagem para a documentação de ontologias. Esta tecnologia mostra-se adequada, tendo em vista que torna natural a definição de novos termos a partir de outros mais primitivos, permitindo navegação entre definições, exemplos e a formalização, incluindo os axiomas.

O Processo de Desenvolvimento de uma Ontologia

O processo de desenvolvimento de uma ontologia deve ser visto como um processo fortemente iterativo, e não como passos seqüenciais. A etapa de captura pode apontar novos requisitos ainda não identificados. Na avaliação, pode-se perceber que os termos descritos são insuficientes para o propósito planejado, impondo um retorno à etapa de captura. Situações semelhantes podem ocorrer na etapa de formalização: incoerências podem ser detectadas, provocando uma revisão das especificações e dos termos definidos na ontologia. Finalmente, quando for necessário integrar uma ontologia com outras existentes, este processo pode ter substancial impacto na definição e formalização dos termos.

As etapas do processo de desenvolvimento de uma ontologia e suas interdependências são ilustradas pela figura 3.12. As linhas tracejadas indicam que há uma interação constante, embora mais fraca, entre as etapas associadas. As linhas cheias mostram o fluxo principal de trabalho no processo de construção de uma ontologia. A linha envolvendo as etapas de captura e formalização da ontologia realça a forte interação e, por conseguinte iteração, que ocorre entre essas etapas.

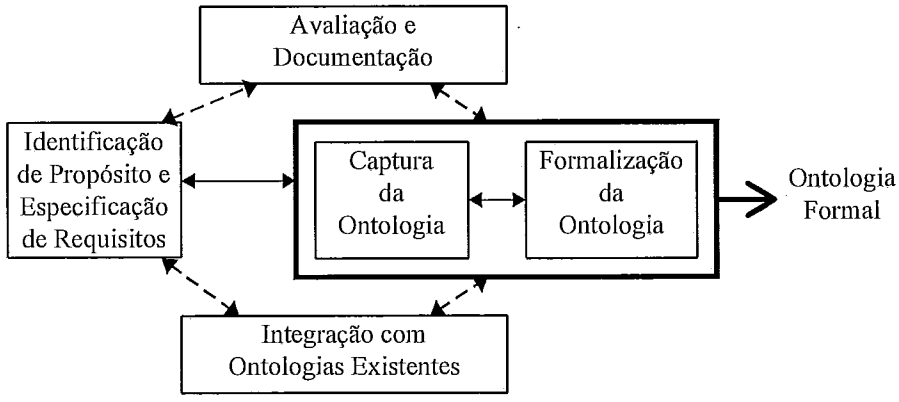


Figura 3.12 - Etapas do Desenvolvimento de uma Ontologia e suas Interdependências.

Uma vez obtida a ontologia formal, muitas vezes é desejável torná-la operacional. Para tal, duas outras atividades devem ser realizadas: projeto e codificação. No projeto, os conceitos, relações e axiomas da ontologia formal devem ser colocados em um formato compatível com a linguagem de implementação. Na codificação, a ontologia é codificada na linguagem escolhida.

3.4.3 - Engenharia de Conhecimento Baseada em Ontologias

Assim como qualquer software reutilizável, bases de conhecimento têm de ser projetadas para o reuso e, neste contexto, ontologias assumem um importante papel. Da mesma forma, modelos de tarefa, como proposto em KADS, CommonKADS e Tarefas Genéricas, são elementos bastante úteis. Assim, as perspectivas de domínio e de tarefa devem ser sempre consideradas, buscando prover um meio eficaz de produção de SBCs, fortemente apoiado no reuso de componentes. Nesta seção propomos uma abordagem de Engenharia de Conhecimento com esta visão, tendo ontologias como o elemento-chave da proposta.

Como qualquer abordagem de Engenharia de Conhecimento para o desenvolvimento de SBCs, a abordagem proposta apresenta atividades de análise de requisitos, modelagem de conhecimento, projeto, implementação e testes. A diferença básica, no entanto, surge na condução do processo de aquisição de conhecimento. A aquisição de conhecimento é uma atividade que permeia todas as fases do desenvolvimento de um SBC, sendo que seus maiores esforços concentram-se na análise de requisitos e, principalmente, na modelagem de conhecimento. Na abordagem baseada em reutilização, o que se busca é um meio sistemático de se conduzir esta atividade, de modo que os resultados sejam passíveis de reuso.

Em um primeiro momento, conhecimento genérico aplicável a vários sistemas, é capturado na forma de ontologias e modelos de tarefa. Estes elementos são, então, disponibilizados em uma biblioteca para reuso. Na segunda etapa da aquisição, os requisitos de conhecimento específicos de uma aplicação particular são considerados.

Basicamente, a abordagem proposta envolve os seguintes passos:

1. *Análise de Requisitos*: O desenvolvimento de um SBC, assim como qualquer software, começa pela especificação de requisitos, onde se procura obter uma visão inicial do tipo de aplicação requerida. Esta fase tem o intuito de gerar uma descrição informal do domínio e da tarefa da aplicação e requer, tipicamente, entrevistas com gerentes e especialistas do domínio, leituras sobre a área, etc.
2. *Seleção/Construção de Ontologias*: Uma vez identificado o domínio da aplicação, deve-se selecionar a(s) ontologia(s) apropriada(s), adequando-a(s) ao desenvolvimento em questão. A ontologia pode ser estendida com novos conceitos e relações no mesmo nível da teoria, ou em uma abordagem de ontologias em níveis. Outros tipos de extensão são possíveis, tais como as micro-teorias e as projeções propostas por BORST et al. (1997). Entretanto, uma vez que a maioria dos engenheiros de conhecimento ainda não tem à sua disposição uma biblioteca de ontologias, é necessário considerar a construção de novas ontologias. Para tal, recomenda-se o uso do método de Engenharia de Ontologias proposto na seção 3.4.2.
3. *Seleção/Construção de Modelos de Tarefa*: De forma análoga à perspectiva de domínio, uma análise da perspectiva de tarefa deve ser realizada. O propósito da análise de tarefa é decompor a tarefa real da aplicação em um número de tarefas genéricas. Idealmente, dever-se-ia empregar uma abordagem única, uniforme, para ambas as perspectivas de domínio e tarefa, ou seja, tarefas deveriam ser estudadas utilizando-se ontologias. Contudo, ontologias de tarefa ainda não têm sido devidamente exploradas, estando em um estágio bastante prematuro. Assim, uma abordagem pragmaticamente mais eficiente consiste em utilizar modelos de tarefa, como proposto pelos vários métodos baseados em tarefa, alguns deles discutidos na seção 3.2. Em especial, os modelos de tarefa de CommonKADS têm sido utilizados com bastante sucesso pela comunidade de IA e, portanto, podem ser adotados nesta etapa.
4. *Compatibilização da Ontologia com os Modelos de Tarefa*: Nesta etapa, especifica-se a interação entre as ontologias e os modelos de tarefa. Ao utilizarmos duas estratégias separadas para desenvolver modelos de resolução de problema, assumimos que é desejável que as modelagens de tarefa e de

domínio tenham um acoplamento suave. O ponto de ligação entre esses dois modelos é exatamente o papel que o conhecimento descrito no modelo de domínio deverá desempenhar no processo de resolução de problema, descrito nos modelos de tarefa. Assim, os *papéis de conhecimento*[2] podem ser considerados o ponto-chave no controle da interação, e como tal, precisam ser descritos claramente. Ao se mapear conceitos da ontologia em papéis dos modelos de tarefa, estabelece-se claramente a relação entre os papéis de conhecimento e as categorias ontológicas que podem preenchê-los. Com este mapeamento, tem-se um modelo completo do esqueleto da aplicação.

5. *Aquisição e Modelagem do Conhecimento Específico da Aplicação*: Uma vez delineado o esqueleto da aplicação, é preciso dar-lhe corpo. Assim, é necessário instanciar a ontologia com o conhecimento necessário para a aplicação e associar métodos de resolução de problemas às tarefas do modelo de tarefas. Enquanto a ontologia define os conceitos usados no universo de discurso, a instanciação de uma ontologia compreende um conjunto de declarações sobre elementos do universo de discurso, usando os conceitos e relações definidos na ontologia. Além das instanciações das ontologias, alguns tipos de conhecimento, específicos da aplicação, tais como heurísticas e conhecimento compilado, devem ser elicitados e instanciados. É importante frisar, contudo, que as ontologias guiam fortemente a realização desta tarefa.
6. *Projeto*: Assim como em qualquer desenvolvimento de software, a fase de Projeto diz respeito a considerações de aspectos tecnológicos impostos pela plataforma de implementação.
7. *Implementação e Testes*: Na implementação, as especificações de projeto são transformadas no sistema final, levando-se em consideração restrições não funcionais, de caráter simbólico e tecnológico. Finalmente, o sistema deve ser testado para verificar se atende aos propósitos para os quais foi projetado.

[2] Um *papel de conhecimento* em um modelo de tarefas descreve o papel que uma porção de conhecimento desempenha na resolução do problema descrito por este modelo.

Comparações com outras Abordagens

A abordagem descrita apresenta muitos pontos de confluência com o método para desenvolvimento de SBCs no domínio médico, proposto no âmbito do Projeto GAMES (HEIJST, 1995). Em GAMES, dois modelos básicos são utilizados: o modelo epistemológico, que especifica o conhecimento requerido pelo sistema, e o computacional, que considera a especificação de representações e algoritmos no nível simbólico.

As atividades ligadas ao modelo computacional estão diretamente relacionadas às atividades de projeto, implementação e testes da abordagem anteriormente proposta, sendo que, uma vez que a arquitetura dos modelos computacionais de GAMES é baseada em uma arquitetura de quadro-negro (*blackboard*), atividades mais específicas concernentes a esta arquitetura são descritas.

A construção do modelo epistemológico de GAMES envolve quatro atividades: (i) construção de um modelo de tarefa para a aplicação, (ii) seleção e configuração das ontologias apropriadas e, se necessário, refinamento destas, (iii) mapeamento da ontologia nos modelos de tarefa e (iv) instanciação da ontologia de aplicação.

Poder-se-ia argumentar que estas atividades correspondem exatamente às atividades iniciais da proposta apresentada, a fora uma mudança na ordem das atividades de construção das ontologias e dos modelos de tarefa. Entretanto, esta não é a diferença fundamental, mas apenas uma conseqüência dela. De fato, a diferença básica está na filosofia adotada. Em primeiro lugar, ontologias na proposta anteriormente definida são ontologias de domínio e não de aplicação. Assim, uma ontologia em nossa abordagem não é construída para uma aplicação específica e está bastante em linha com a Análise de Domínio na Engenharia de Software. Em GAMES, uma ontologia é construída para uma aplicação específica, razão pela qual modelos de tarefa são construídos antes das ontologias.

Em segundo lugar, uma vez que em GAMES ontologias são de aplicação, as instanciações das ontologias produzem todo o conhecimento necessário para o sistema em desenvolvimento. Em nossa proposta, instanciações de ontologias, assim como as ontologias, representam conhecimento reutilizável. Na etapa de aquisição e

modelagem do conhecimento específico da aplicação, algumas instanciações existentes podem ser reutilizadas enquanto outras precisam ser capturadas. Contudo, além das instanciações da ontologia, esta etapa deve elicitar e modelar outros tipos de conhecimento, estes específicos da aplicação em questão.

Finalmente, GAMES pressupõe a existência de uma biblioteca de ontologias e oferece apenas diretrizes gerais para apoiar a construção de ontologias, diretrizes estas fortemente orientadas à tarefa. A proposta aqui definida, por outro lado, está apoiada no método para a construção de ontologias, proposto na seção 3.4.2.

NECHES et al. (1991) idealizaram um processo de desenvolvimento de SBCs bastante afinado com a abordagem de Engenharia de Conhecimento anteriormente proposta. Este processo começaria pela montagem de componentes reutilizáveis, incluindo porções de bases de conhecimento e resolvedores de problemas de propósito especial. Algum esforço seria necessário para conectar essas partes, contudo, a maior parte deste esforço seria despendido na modelagem do conhecimento especializado e na construção de resolvedores de problema para as tarefas específicas do sistema em questão. Entretanto, ainda que em (NECHES et al., 1991) vários aspectos enfocando a reutilização de conhecimento sejam discutidos, os autores não propõem um método de Engenharia de Conhecimento baseado nestas idéias.

3.5 - Conclusões do Capítulo

Desde a introdução da tecnologia de sistemas baseados em conhecimento nos anos 60, tem se verificado uma demanda cada vez maior por aplicações construídas usando esta tecnologia, sobretudo a partir da metade da década passada. Inicialmente, muita ênfase foi dada à questão da tecnologia de representação e à aquisição de conhecimento como um processo de transferência do conhecimento de um especialista para um sistema. Como resultado, a produtividade no desenvolvimento foi muito baixa e os sistemas desenvolvidos estavam muito aquém das expectativas de seus usuários.

Para mudar este quadro, passou-se a considerar a modelagem como foco da Engenharia de Conhecimento. Este capítulo explorou algumas facetas da modelagem de conhecimento, procurando identificar elementos relacionados com a produtividade

no desenvolvimento de SBCs. Em primeiro lugar, discutiu-se aspectos relacionados à representação de conhecimento e seus níveis, procurando mostrar que a modelagem de conhecimento deve se dar no nível de conhecimento, mais especificamente no nível ontológico.

Na seqüência, foram brevemente apresentadas e comentadas algumas das principais abordagens para a modelagem de conhecimento encontradas na literatura. Pode-se perceber que as abordagens discutidas estão fortemente centradas no conceito de tarefa e defendem que o conhecimento sobre o universo da aplicação deve ser capturado com base na tarefa que o sistema se propõe a tratar. Entretanto, acreditamos que uma abordagem mais produtiva deve considerar tanto a perspectiva de tarefa quanto a perspectiva de domínio, com ênfase na última. Neste contexto, ontologias assumem um importante papel.

Foram discutidos diversos aspectos de ontologias, tais como critérios de qualidade, linguagens para expressar ontologias, impactos na aquisição de conhecimento, métodos para construção e vantagens e desvantagens do uso de ontologias. A modelagem de conhecimento baseada em ontologias é bastante recente e há, ainda, muitas questões em aberto. Para avançar em direção a uma abordagem mais sistemática para a construção de ontologias, foi proposta uma abordagem sistemática para a Engenharia de Ontologias, procurando agregar características das principais idéias sobre desenvolvimento de ontologias. Além disso, propôs-se também uma linguagem gráfica para expressar ontologias e mecanismos de decomposição, para facilitar o processo de construção de ontologias.

Finalmente, uma proposta de Engenharia de Conhecimento foi elaborada, visando conciliar as abordagens baseadas em tarefa e em ontologias. Observando o uso de ontologias nesta abordagem, é possível notar uma forte analogia entre a Engenharia de Ontologias e a Engenharia de Domínio. Esta constatação é bastante útil, já que nos permite olhar de perto trabalhos feitos na área da Engenharia de Domínio, buscando transpor muitas de suas discussões para a Engenharia de Ontologias. De fato, a relação existente entre a Engenharia de Domínio e a Engenharia de Software é a mesma existente entre a Engenharia de Ontologias e a Engenharia de Conhecimento, como mostram as tabelas 3.1 e 3.2.

Tabela 3.1 - Paralelo entre Engenharia de Software e Engenharia de Domínio (PRIETO-DÍAZ, 1991).

Engenharia de Software	**Engenharia de Domínio**
Análise de Requisitos	Análise de Domínio
Especificação do Sistema	Especificação da Infra-estrutura
Projeto e Implementação	Projeto da Infra-estrutura

Tabela 3.2 - Paralelo entre Engenharia de Conhecimento e Engenharia de Ontologias.

Engenharia de Conhecimento	**Engenharia de Ontologias**
Análise de Requisitos do Sistema	Análise de Requisitos da Ontologia
Modelagem de Conhecimento	Captura da Ontologia
Projeto e Implementação	Formalização e Operacionalização

Nos capítulos que se seguem, são usadas muitas das idéias discutidas neste capítulo. No capítulo 4, é proposta uma infra-estrutura para apoiar a reutilização de conhecimento, explorando ontologias e modelos de tarefa. No capítulo 5, a abordagem de Engenharia de Ontologias é utilizada para construir uma ontologia de processo de desenvolvimento de software e, no capítulo 6, a abordagem de Engenharia de Conhecimento baseada em Ontologias é usada no desenvolvimento de uma ferramenta baseada em conhecimento.

Capítulo 4

Servidores de Conhecimento

Conforme discutido no capítulo 2, a integração em Ambientes de Desenvolvimento de Software deve ser vista como uma questão de quatro dimensões: dados, controle, apresentação e conhecimento (TRAVASSOS, 1994). Entretanto, a maioria dos ambientes atuais não considera a última dimensão.

No âmbito do Projeto TABA (ROCHA et al., 1990), a questão da integração de conhecimento tem sido objeto de estudo. Neste capítulo, discutimos como a Estação TABA, um meta-ADS, tem procurado promover a integração de conhecimento, oferecendo um conjunto de funcionalidades para armazenar e para tratar conhecimento.

A seção 4.1 procura dar uma visão geral da Estação TABA e de sua estrutura. A seção 4.2 explora, especificamente, a questão do conhecimento neste ambiente. Na seção 4.3, introduzimos a noção de Servidor de Conhecimento como uma infra-estrutura para promover a integração de conhecimento e discutimos sua arquitetura. A seção 4.4 enfoca as idéias apresentadas no contexto de um Servidor de Conhecimento de Processo de Software. Finalmente, na seção 4.5, são apresentadas as conclusões deste capítulo.

4.1 - A Estação TABA

O projeto TABA (ROCHA et al., 1990) tem como objetivo a construção de uma Estação de Trabalho configurável para o desenvolvimento de software, para

atender às particularidades de domínios de aplicação e projetos específicos. A motivação para a realização deste projeto está na constatação de que domínios de aplicação e projetos diferentes têm características diferentes e que estas devem incidir nos ambientes de desenvolvimento através dos quais os engenheiros de software desenvolvem as aplicações (TRAVASSOS, 1994). Para atender a este objetivo, a Estação TABA possui quatro funções:

I. auxiliar o engenheiro de software no planejamento e instanciação do ambiente mais adequado ao desenvolvimento de um produto específico;
II. auxiliar o engenheiro de software a implementar as ferramentas necessárias ao ambiente definido em (I);
III. permitir aos desenvolvedores do produto (software) o uso da Estação através do ambiente definido e instanciado em (I);
IV. permitir a execução do software na própria Estação para ele configurada.

Estas funções determinam quatro ambientes na Estação TABA (TRAVASSOS, 1994):

- *Ambiente Especificador e Instanciador de ADSs*: é o meta-ambiente TABA. Sua função é especificar o ADS mais adequado para o desenvolvimento de um produto de software, em um determinado contexto, e instanciar este ADS;
- *Ambiente para Construção de Ferramentas*: este ambiente auxilia o engenheiro de software na construção de novas ferramentas para a Estação TABA e sua incorporação ao meta-ambiente;
- *Ambiente de Desenvolvimento*: é o ADS que foi especificado e instanciado através do meta-ambiente, e,
- *Ambiente de Execução*: é o local onde o produto de software poderá ser executado.

A Estrutura da Estação TABA

Um ADS é um produto de software e, como tal, responde a estímulos dos usuários de acordo com as funcionalidades que foram previstas em sua construção e que estão disponíveis para uso. Estas funcionalidades têm que trabalhar integradas, de forma harmônica e, o que é fundamental, independentes mas não livres, no sentido de

guardarem uma estreita relação entre si, de modo que o ADS possa estabelecer uma seqüência de ativação (TRAVASSOS, 1994).

Esta forma de estruturação é possível quando se consegue tratar as funcionalidades como componentes separados, contendo controle sobre a sua existência, suas informações e estados associados, e possuindo um conjunto de funcionalidades básicas associadas, que são estimuladas através de uma comunicação organizada entre o usuário e o sistema. Levando em consideração estes aspectos, a estrutura da Estação TABA busca distribuir as funcionalidades necessárias em *componentes*, capazes de interagir diretamente entre si, permitindo, assim, a obtenção de toda a funcionalidade do ambiente (TRAVASSOS, 1994).

Tendo em vista a necessidade de se modelar a estrutura de um ADS, realçando sua constituição e mostrando que, na realidade, os serviços existentes não interagem aos pares, mas sim como um todo, adotou-se, para a Estação TABA, a estrutura mostrada na figura 4.1. Nesta estrutura, busca-se mostrar que um ADS está inserido no contexto de produtos de software e é o responsável, com seu conjunto de funcionalidades e informações associadas, por traduzir uma representação do mundo real para o mundo computacional (TRAVASSOS, 1994).

Os componentes presentes na Estação TABA são (TRAVASSOS, 1994):

- *Componente Repositório Comum*: responsável pelo controle, gerenciamento e armazenamento dos objetos manuseados pelo ambiente e meta-ambiente. É de sua responsabilidade manter a consistência e a integridade das informações;
- *Componente Controle de Versões*: controla e gerencia versões de documentos e itens de software produzidos. A partir dele, o usuário tem condição de obter informações sobre um determinado momento no desenvolvimento.
- *Componente de Interface com o Usuário*: responsável pela gerência e controle da interação do usuário com o ADS. É de sua responsabilidade garantir a consistência da apresentação das ferramentas e permitir, na medida do possível, que o usuário ajuste a aparência do ADS às suas preferências pessoais;
- *Componente Cooperação*: trata da comunicação entre os usuários e suas necessidades de interação com outros membros da equipe. Possui definições

de protocolos de comunicação que devem ser utilizados no convívio no ADS. Este componente precisa auxiliar o componente de Interface com o Usuário no sentido de prover funcionalidades e recursos que o estendam de forma a suportar a interface de grupo;

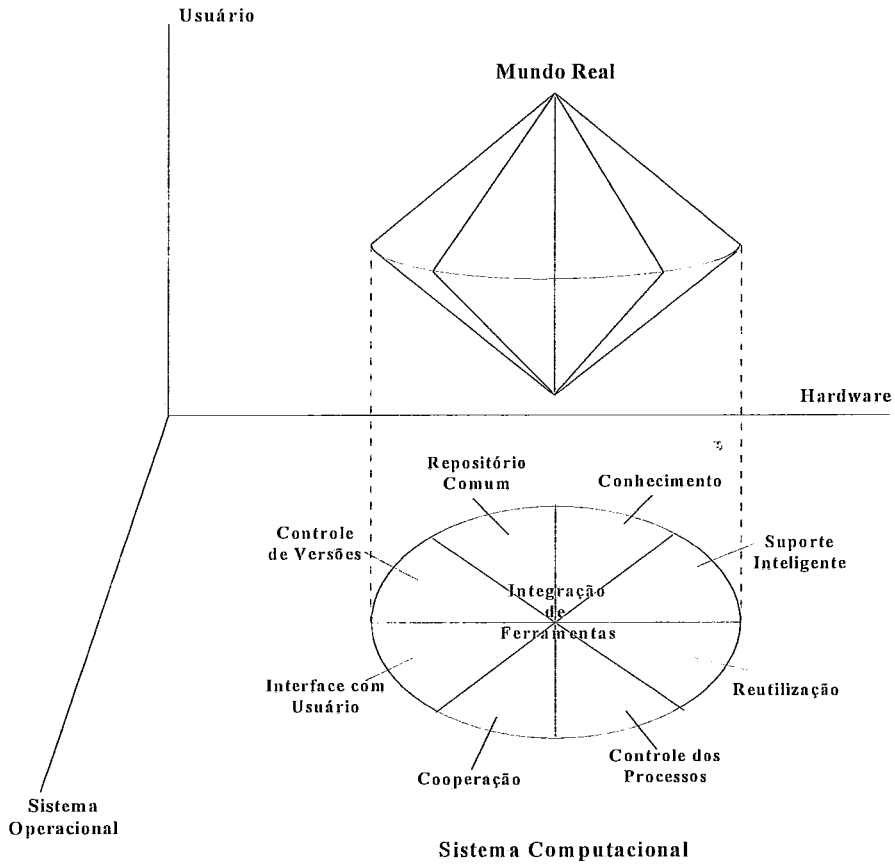


Figura 4.1 - Estrutura de um ADS na Estação TABA (TRAVASSOS, 1994).

- *Componente Controle dos Processos*: responsável pelo controle e gerenciamento do processo de desenvolvimento. Identifica atividades, gerencia a execução das ferramentas e controla os papéis e atividades dos usuários, levantando dados sobre a realização do processo para o meta-ambiente;
- *Componente Reutilização*: provê o ADS de mecanismos que possibilitam a reutilização de trabalhos anteriores. Esta reutilização pode se dar a nível da especificação do ADS, de componentes para a construção de ferramentas, bem como de componentes de todo tipo para a construção da aplicação. Assim sendo, este é um componente muito importante no meta-ambiente, pois

possibilita ao instanciador e ao construtor de ferramentas o acesso a itens de software e ferramentas previamente desenvolvidos;

- *Componente Suporte Inteligente*: fornece inteligência global ao ambiente, assistindo o usuário na utilização do meta-ambiente e dos ADSs, através de assistentes inteligentes;
- *Componente Conhecimento*: incorpora mecanismos para o armazenamento e a utilização de conhecimento, que podem ser utilizados tanto pelo meta-ambiente, quanto pelos ADSs instanciados.

A utilização, em conjunto, destes componentes permite a integração de ferramentas ao ambiente. Estes componentes definem políticas de armazenamento e a forma de interface com o usuário, provêem recursos para a incorporação de ferramentas externas ao ambiente, e, de forma mais específica, definem a filosofia de integração da Estação TABA e de seus ambientes instanciados (TRAVASSOS, 1994).

4.2 - O Conhecimento na Estação TABA

Dentre os componentes que estruturam a arquitetura da Estação TABA, dois merecem destaque no que se refere à questão do conhecimento: o componente *Suporte Inteligente* e o componente *Conhecimento*. O primeiro fornece inteligência global ao ambiente, apoiando o engenheiro de software na utilização do meta-ambiente e dos ADSs instanciados, através de assistentes inteligentes. O último incorpora funcionalidades para o armazenamento e a utilização de conhecimento.

Embora possa parecer, à primeira vista, que estes dois componentes possuem as mesmas funcionalidades, o que ocorre, de fato, é um forte relacionamento entre eles: o componente *Suporte Inteligente* utiliza os mecanismos de armazenamento e utilização de conhecimento, providos pelo componente *Conhecimento*, para apoiar o desenvolvedor na utilização da Estação e de seus ADSs instanciados. O componente *Conhecimento*, por sua vez, funciona para o ambiente, auxiliando outros componentes em seu funcionamento.

No tocante à infra-estrutura, o componente *Conhecimento* é o elemento principal. Tendo em vista a necessidade de se integrar conhecimento, a Estação TABA deve suportar bases de conhecimento modulares, uniformes e passíveis de

acesso às ferramentas. Cabe ao componente *Conhecimento* prover ao ADS a funcionalidade básica para a manipulação de bases de conhecimento.

Uma base de conhecimento é um conjunto de representações de fatos sobre o mundo, sendo que cada representação individual é dita uma sentença (RUSSELL et al., 1995). Para tratar conhecimento, basicamente, um ADS deve permitir adicionar e remover sentenças a uma base de conhecimento e consultar o que é conhecido. Diferentes funções de consulta devem ser providas, incluindo desde uma versão que apenas confirme se uma consulta feita é passível de derivação a partir das sentenças de uma base de conhecimento, até uma versão que retorne um conjunto de objetos que tornem a consulta verdadeira. Sentenças são expressas em uma linguagem de representação de conhecimento (RUSSELL et al., 1995), que oferece os mecanismos de inferência. Assim, o modelo básico do componente de *Conhecimento* da Estação TABA apresenta as seguintes classes, como mostra o diagrama de classes da figura 4.2, que usa a convenção da UML[1] (FOWLER et al., 1997):

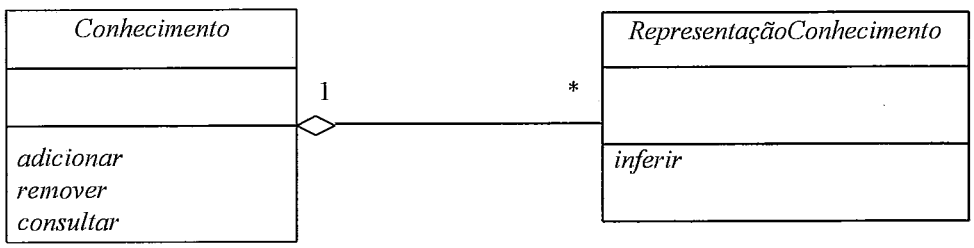


Figura 4.2 - Modelo Básico do Componente de Conhecimento.

- *Classe Conhecimento*: é responsável pelo armazenamento e acesso ao conhecimento na Estação TABA. É uma classe abstrata que fornece funcionalidades básicas para suas especializações, que descrevem vários tipos de conhecimento, tais como conhecimento sobre processos de desenvolvimento, atividades, procedimentos e domínios de aplicação, entre outros, como mostra a figura 4.3. O conhecimento sobre domínios de aplicação deve ser especializado para cada domínio particular e, portanto,

[1] UML - Unified Modeling Language. Na convenção da UML, um item abstrato (classe ou método) tem seu nome escrito em itálico. Assim, na figura 4.2, tanto as classes *Conhecimento* e *RepresentaçãoConhecimento* quanto seus métodos são abstratos.

esta classe aparece no modelo como uma classe abstrata. Outros tipos de conhecimento, além dos mostrados na figura 4.3, podem ser descritos, bastando para tal, que novas especializações da classe *Conhecimento* sejam criadas. Desta forma, a estrutura proposta toma um formato bastante flexível, permitindo uma fácil evolução.

- *Classe Representação de Conhecimento*: é responsável por prover o ambiente com a capacidade de representar conhecimento utilizando diferentes tecnologias de representação, dadas pelas suas especializações, como mostra a figura 4.3. Não existe a priori uma forma melhor, mais eficiente e mais fácil de representar o conhecimento, mas sim uma maior adequação ao problema que se quer resolver. Deste modo, idealmente, um ADS deve tornar disponíveis várias formas de representação.

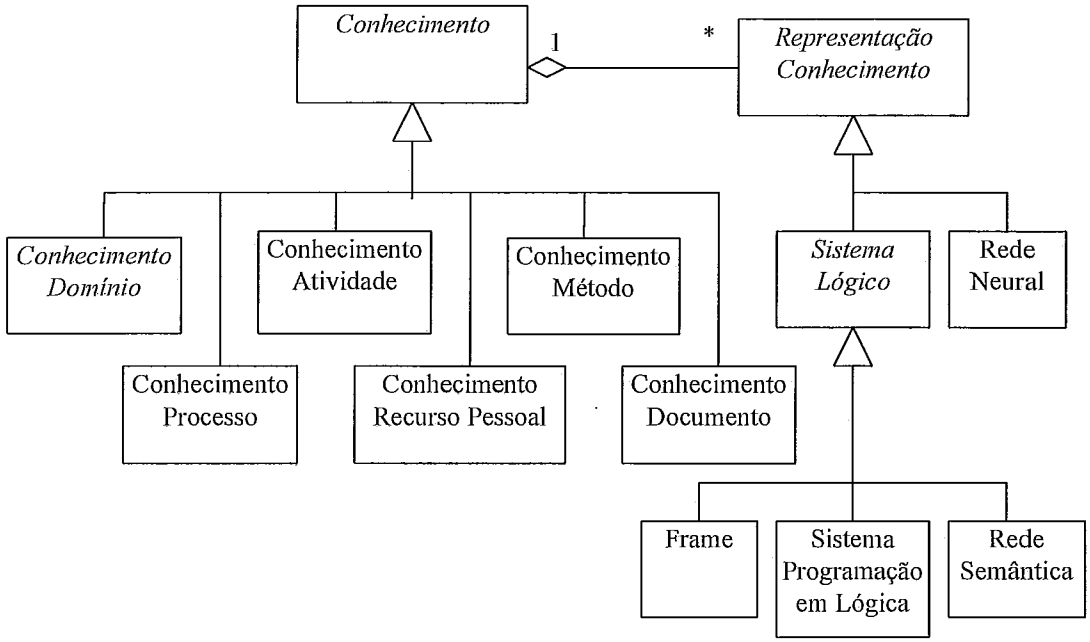


Figura 4.3 - O Componente *Conhecimento*.

O componente *Suporte Inteligente* consiste de uma hierarquia de classes, tendo como classe raiz a classe abstrata *Assistente Inteligente*. As subclasses desta hierarquia descrevem as várias ferramentas com suporte baseado em conhecimento desenvolvidas utilizando a infra-estrutura de conhecimento da Estação TABA. A figura 4.4 mostra os componentes *Suporte Inteligente* e Conhecimento e suas relações com outros componentes da Estação TABA.

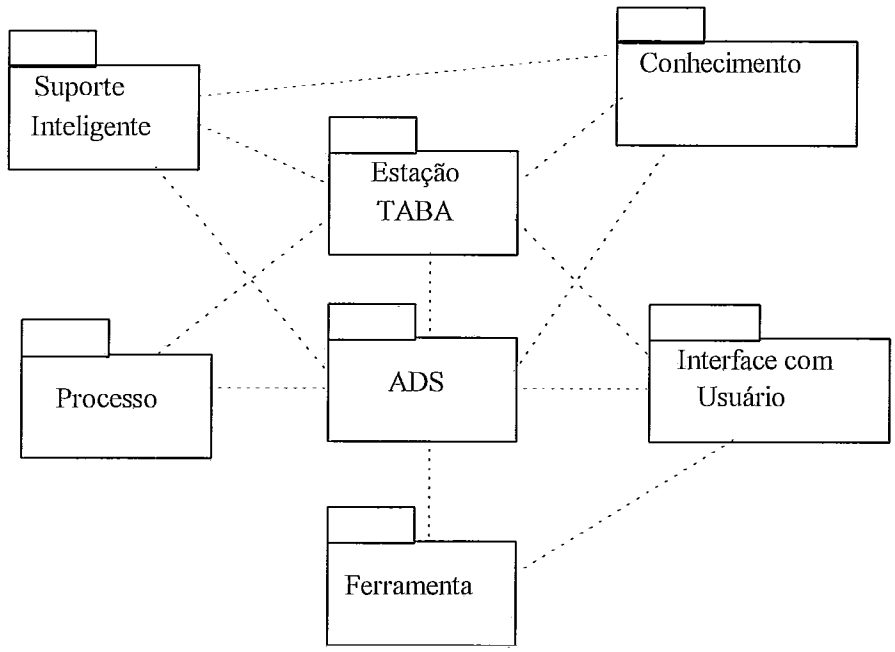


Figura 4.4 - Os Componentes *Suporte Inteligente* e *Conhecimento* na Estação TABA.

O Conhecimento no Meta-Ambiente TABA

O objetivo do meta-ambiente da Estação TABA é permitir a especificação de ADSs adequados a diferentes contextos e sua instanciação como um ambiente integrado. Esta instanciação se faz a partir de um conjunto de facilidades e funcionalidades que suportam o engenheiro de software na definição e construção de novos ambientes.

Basicamente, o conhecimento pode ser utilizado no meta-ambiente para apoiar a definição de processos de software e instanciação do ADS correspondente, através de um assistente inteligente desenvolvido com esta finalidade.

O Conhecimento em um Ambiente Instanciado TABA

Em função do domínio de aplicação, ambientes são especificados e instanciados pelo meta-ambiente, de modo a prover o ADS mais adequado para o desenvolvimento de um produto de software. Estes ambientes, por sua vez, são compostos de ferramentas, internas ou externas, que necessitam trabalhar de forma integrada.

No que se refere ao conhecimento, o meta-ambiente deve tornar disponível para um ADS instanciado, todo o conhecimento relevante para o processo em curso,

isto é, o processo que o ADS automatiza. Este conhecimento pode ser utilizado, então, por ferramentas para acompanhamento, execução e adaptação do processo de desenvolvimento (ARAÚJO, 1998). Além disso, em função do domínio da aplicação, outros tipos de conhecimento poderão ser tratados, assim como novos assistentes inteligentes poderão ser incorporados ao ADS.

A Infra-estrutura de Conhecimento da Estação TABA

No que tange à representação de conhecimento, a implementação da infra-estrutura de conhecimento da Estação TABA, mostrada na figura 4.3, foi iniciada através da classe *Sistema de Programação em Lógica*, oferecendo funcionalidades para a descrição de conhecimento usando Prolog (FALBO et al., 1996) (FALBO et al., 1997). Com o intuito de reduzir os esforços necessários para realização, optou-se por utilizar uma máquina Prolog externa, o SWI-Prolog (WIELEMAKER, 1998).

A Estação TABA está implementada em Eiffel (MEYER, 1992), uma linguagem orientada a objetos, e, assim, a máquina de inferência do SWI-Prolog foi encapsulada na classe Eiffel *Sistema de Programação em Lógica.* Para tal, foi necessário estabelecer um mecanismo de comunicação entre as duas linguagens. Uma vez que as linguagens SWI-Prolog e Eiffel não oferecem mecanismos que permitam uma comunicação direta entre elas, mas, por outro lado, fornecem mecanismos para comunicação com código externo escrito em linguagem C, foi adotada uma abordagem de interfaceamento SWI-Prolog/C/Eiffel, como mostra a figura 4.5.

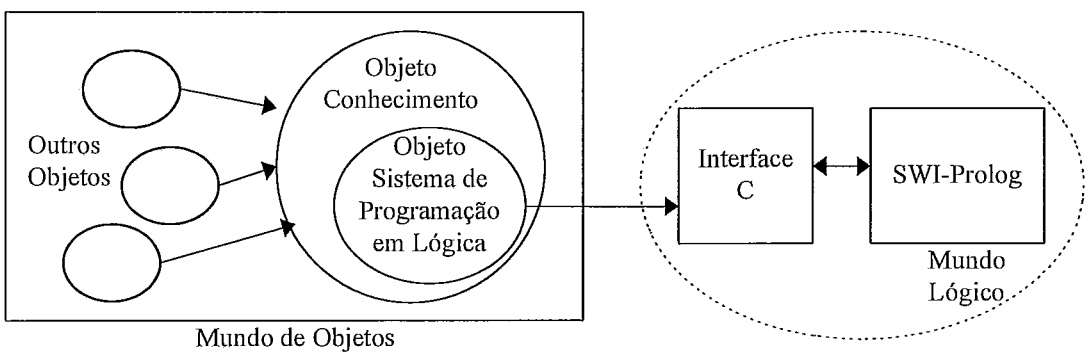


Figura 4.5 - A Interface entre os Mundos Lógico e de Objetos.

Serviços definidos em Eiffel encapsulam a máquina Prolog. As operações necessárias à comunicação entre as linguagens C e SWI-Prolog foram encapsuladas em métodos privados da classe *Sistema de Programação em Lógica*, não sendo, portanto, passíveis de acesso aos demais objetos. Essas operações foram definidas como funções externas, utilizando recursos para interfaceamento com a linguagem C, fornecidos pelo SWI-Prolog. Assim, garantiu-se a integridade do sistema, já que os serviços de inferência são oferecidos apenas aos usuários da classe *Sistema de Programação em Lógica.*

Os objetos da classe *Sistema de Programação em Lógica* enviam mensagens para a interface C que, juntamente com a máquina Prolog, funcionam como um objeto externo (representado pela linha pontilhada na figura 4.5). Este executa uma operação e retorna um resultado. Assim, o impacto da interface no mundo de objetos é minimizado.

A Infra-estrutura de Conhecimento da Estação TABA e a Integração de Conhecimento

Analisando a infra-estrutura de conhecimento proposta para a Estação TABA (figura 4.3) é possível notar que sua preocupação principal está centrada na questão da representação de conhecimento e, portanto, no nível simbólico (NEWELL, 1982). Entretanto, conforme discutido no capítulo 3, para se obter integração de conhecimento, é necessário considerar, principalmente, aspectos do nível de conhecimento (NEWELL, 1982). Tornar disponíveis vários formalismos de representação não garante a integração de conhecimento. Assim, esta infra-estrutura é, na realidade, apenas um ferramental para facilitar a integração.

O ponto-chave para a integração de conhecimento está na modelagem de conhecimento. O conhecimento tem que ser modelado *para* o reuso. Componentes de conhecimento devem estar disponíveis no ambiente para serem usados pelas várias ferramentas.

Esta constatação levou a uma mudança no enfoque do estudo sobre integração de conhecimento: ao invés de buscarmos a integração de conhecimento através da integração de tecnologias de representação, passamos a explorar a construção de componentes de conhecimento, fundamentados em ontologias (FALBO et al., 1998a)

e modelos de tarefa. Componentes construídos a partir de ontologias descrevem o conhecimento sobre o domínio de interesse. Componentes construídos a partir dos modelos de tarefa descrevem conhecimento genérico de tarefa, aplicável a vários domínios. À infra-estrutura suportando componentes de conhecimento construídos sob este enfoque, demos o nome de Servidores de Conhecimento.

Cabe, ainda, comentar a razão pela qual utilizamos enfoques distintos para a construção dos componentes de conhecimento. A priori, ontologias podem ser utilizadas para modelar tanto o conhecimento de domínio quanto o de tarefa. Contudo, o conhecimento sobre os tipos principais de tarefas já foram bastante estudados ((CHANDRASEKARAN et al., 1993), (SCHREIBER et al., 1993), (BREUKER et al., 1994), entre outros) e, portanto, optamos por utilizar diretamente os resultados destes trabalhos, mais especificamente os modelos de tarefa de CommonKADS (BREUKER et al., 1994), para a construção dos componentes de conhecimento de tarefa.

4.3 - Servidores de Conhecimento e a Integração

Poder-se-ia pensar que a integração de conhecimento seria mais facilmente atingida se construíssemos um SBC capaz de concentrar todo o conhecimento relevante para um ADS. Entretanto, é impossível antever e modelar todo este conhecimento e, mais importante, esta abordagem é indesejável. O conhecimento deve ser capturado e modelado à medida que novas necessidades são detectadas. Assim, uma potencial abordagem consiste em desenvolver uma estrutura capaz de capturar um modelo de conhecimento, que seja modular e passível de extensão. Esta estrutura é um Servidor de Conhecimento.

Servidores de Conhecimento podem ser vistos como uma nova classe de aplicações. Enquanto um SBC tem a finalidade de atuar como um agente resolvedor de problemas, um Servidor de Conhecimento tem por objetivo prover uma infra-estrutura comum para o desenvolvimento de um conjunto de agentes dessa natureza *em um universo de discurso*, i.e., um Servidor de Conhecimento é uma infra-estrutura de conhecimento sobre um universo de discurso. Basicamente, um Servidor de Conhecimento tem de prover:

- um vocabulário comum com interpretação definida dos termos no universo de discurso, e,
- um conjunto de máquinas de inferência customizadas para os tipos de problema mais comumente encontrados no universo de discurso em questão.

Para prover um vocabulário comum, de acordo com o capítulo 3, um Servidor de Conhecimento deve adotar uma representação de conhecimento baseada em ontologia, isto é, deve ser construído com base em um conjunto de ontologias definindo formalmente os termos usados no universo de discurso apoiado pelo ADS, de modo que todas as aplicações e assistentes baseados em conhecimento desenvolvidos para este ADS compartilhem o mesmo vocabulário. Para permitir a customização de máquinas de inferência, o Servidor de Conhecimento deve prover resolvedores de problemas para os tipos de problema que aparecem com maior freqüência neste mesmo universo de discurso.

Outras características desejáveis de um Servidor de Conhecimento incluem: facilidades para tratar algumas tecnologias de representação de conhecimento e facilidades para prover interoperabilidade entre elas. Deve-se notar que estas facilidades dizem respeito a aspectos do nível simbólico de NEWELL (1982), em contraste com as anteriores que referem-se a aspectos do nível de conhecimento.

Uma vez que, neste trabalho, enfocamos a integração de conhecimento no nível de conhecimento e não do nível simbólico, utilizamos uma única tecnologia de representação, e os mecanismos de inferência por ela oferecidos, e enfatizamos a organização do conhecimento, como o fator fundamental para a integração de conhecimento.

A meta de um Servidor de Conhecimento no contexto de ADSs é servir de base para a construção dos vários assistentes baseados em conhecimento de um ADS, garantindo uma semântica comum e permitindo o reuso e o compartilhamento de conhecimento. Por conseguinte, sua arquitetura é fortemente influenciada pela arquitetura dos Sistemas Baseados em Conhecimento (SBCs). Assim, antes de propormos uma arquitetura para Servidores de Conhecimento, é necessário olhar atentamente para a arquitetura dos SBCs.

Arquitetura dos Sistemas Baseados em Conhecimento

Sistemas Baseados em Conhecimento têm como ponto principal da sua arquitetura a separação entre o conhecimento do domínio do problema e o conhecimento geral da solução do problema (WERNECK, 1995). Assim sendo, têm como componentes básicos de sua arquitetura a *base de conhecimento*, contendo os conceitos, relações, fatos, regras e outras representações do conhecimento de domínio, e a *máquina de inferência*, contendo as estratégias de raciocínio e inferência, independentes do domínio. A figura 4.6 apresenta uma arquitetura básica, geral para os SBCs.

A base de conhecimento é dividida em duas partes. A parte genérica contém o conhecimento geral sobre o domínio, enquanto a memória de trabalho contém os dados específicos de um caso. A máquina de inferência é o mecanismo de controle do sistema que avalia e aplica o conhecimento da parte genérica de acordo com as informações da memória de trabalho.

Outros componentes desejáveis nessa arquitetura incluem: *interface com o usuário*, que pode variar de simples diálogos até um processador de linguagem natural, *mecanismo de explicação*, para elucidar a linha de raciocínio adotada em um caso, e um *editor de base de conhecimento*, para aquisição e codificação na base genérica de novos conhecimentos (WERNECK, 1995).

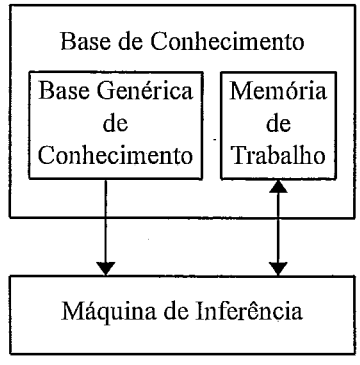


Figura 4.6 - Arquitetura Básica Geral de SBCs (WERNECK, 1995).

A Arquitetura de Servidores de Conhecimento

Do ponto de vista de Servidores de Conhecimento, a arquitetura de SBCs apresentada anteriormente mostra que dois componentes merecem atenção especial: a

base genérica de conhecimento e a máquina de inferência. Uma vez que um Servidor de Conhecimento visa apoiar a construção de SBCs, ele deve prover os engenheiros de conhecimento com módulos de conhecimento, passíveis de serem combinados para atender às especificidades de uma aplicação particular, e máquinas de inferência customizadas para tipos de problema variados.

Para permitir o reuso de conhecimento de tarefa é necessário prover *templates* de resolvedores genéricos de problema. Ao invés de prover apenas sistemas de representação e suas máquinas genéricas de inferência, um Servidor de Conhecimento deve oferecer uma biblioteca de resolvedores de problemas, passíveis de serem instanciados e adaptados para aplicações particulares.

No que tange ao reuso de conhecimento de domínio, é desejável que a base de conhecimento do Servidor seja modular e baseada em ontologias. Ontologias e suas instanciações podem ser implementadas em módulos de conhecimento, de modo que a base de conhecimento de uma aplicação que se comprometa com uma ou várias ontologias venha a ser a conjunção dos módulos de conhecimento correspondentes, mais o conhecimento específico da aplicação particular.

Desta forma, como mostra a figura 4.7, a arquitetura de um Servidor de Conhecimento compreende essencialmente:

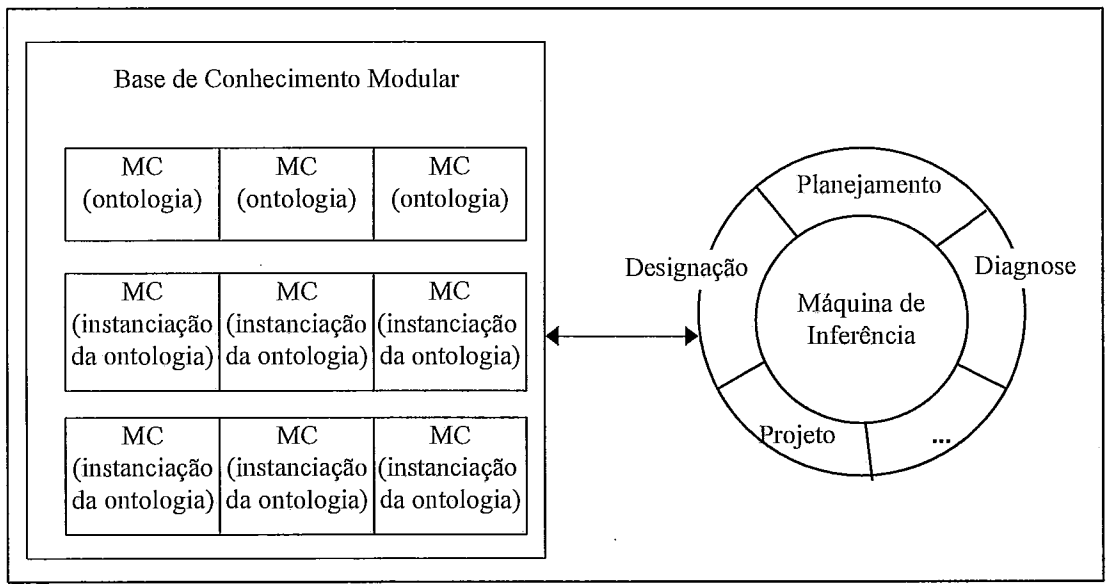


Figura 4.7 - Arquitetura Geral de Servidores de Conhecimento

- uma base de conhecimento modular, onde cada módulo de conhecimento (MC) contém um corpo de conhecimento reutilizável, construído com base em uma ontologia, e,
- a máquina de inferência do sistema de representação adotado, associada a um conjunto de *templates* de resolvedores de problemas, para especializá-la para os tipos de problema mais freqüentemente encontrados no universo de discurso.

A base de conhecimento modular é essencialmente uma biblioteca de ontologias, já que seu critério de modularização é dado pelo uso de ontologias. Cada ontologia é implementada em um módulo de conhecimento na linguagem do sistema de representação, assim como cada uma de suas instanciações.

A máquina de inferência é aquela provida pelo sistema de representação de conhecimento adotado. Os *templates* de resolvedores de problemas, por sua vez, implementam modelos de tarefa genéricos para os tipos de problemas que ocorrem com freqüência no universo de discurso do Servidor de Conhecimento.

Os módulos de conhecimento e os *templates* de resolvedores de tipos de problema são os componentes reutilizáveis que o Servidor de Conhecimento oferece para auxiliar o processo de construção de SBCs no universo de discurso suportado por ele. Além disso, guardam estreita relação entre si: os papéis de conhecimento considerados em um *template* de resolvedor de problema devem ser preenchidos com o conhecimento de domínio descrito nos módulos de conhecimento. Assim, com estes componentes disponíveis, é possível uma mudança de enfoque na construção dos assistentes inteligentes.

4.4 - Um Servidor de Conhecimento de Processo de Software

Para materializar a idéia de Servidores de Conhecimento, foi desenvolvido, para a Estação TABA, um protótipo de um Servidor de Conhecimento para o universo de discurso de processos de desenvolvimento de software, o Servidor de Conhecimento de Processo (SCP). A motivação para a construção deste servidor advém do reconhecimento da importância de se ter conhecimento sobre processos de

software explicitamente representado e passível de compartilhamento por várias ferramentas em um ADS.

A Automatização do Processo de Software na Estação TABA

Inicialmente, a Estação TABA utilizava a definição de um processo de software apenas para nortear a utilização de ferramentas ao longo do desenvolvimento, sem garantir, com isto, que o processo pudesse ser realmente acompanhado ou modificado, quando necessário (ARAÚJO, 1998). A instanciação de um ambiente era apoiada por XAMÃ (AGUIAR, 1992), um assistente inteligente, que considerava apenas alguns ingredientes essenciais para a definição de um processo, entre eles a escolha de um modelo de ciclo de vida e a escolha de métodos e ferramentas para apoiar a realização de suas atividades. Desta forma, o ambiente trabalhava em função de um processo pré-estabelecido no momento de sua instanciação.

Entretanto, um processo de software deve suportar adaptações, pois, na prática, todo processo de desenvolvimento sofre alterações durante a sua execução. Assim, ao invés de se instanciar todas as atividades e seus relacionamentos no momento da instanciação do ambiente, deve-se gerar, através do meta-ambiente, uma descrição do processo, contendo as decisões estabelecidas no modelo inicial do processo definido (ARAÚJO, 1998).

Esta característica tornou inadequada a abordagem inicialmente adotada, levando ao desenvolvimento de novas funcionalidades para apoiar a descrição, o acompanhamento e a execução de processos de software, aproximando a Estação TABA de um ambiente centrado em processo (ARAÚJO, 1998).

De maneira geral, a estrutura atual para tratamento do processo de desenvolvimento de software na Estação TABA leva em consideração as seguintes características (ARAÚJO, 1998):

- permite a descrição de processos de software, expressando atividades, suas características e seus relacionamentos;
- implementa um mecanismo de tratamento para a descrição do processo que permite a instanciação e o controle das atividades, verificando a disponibilidade de recursos e artefatos necessários à sua execução e,

observando pré-atividades e pós-atividades, de forma a garantir a execução do processo de desenvolvimento;

- permite que sejam feitas alterações durante a execução do processo, sem que estas afetem as atividades já concluídas, ou seja, as mudanças são válidas a partir do ponto atual de execução;
- oferece alternativas para alteração do processo de desenvolvimento, que não causam inconsistências no mesmo. Outras alternativas podem ser construídas a qualquer momento pelo Engenheiro de Software responsável pelo ambiente.

Esta estrutura torna muito mais flexíveis as alterações no processo, uma vez que estas são feitas em uma descrição de processo, ao invés de impactar diretamente no processo em questão (ARAÚJO, 1998).

Para apoiar a atual estrutura de tratamento de processos de software, a Estação TABA oferece algumas ferramentas. O meta-ambiente permite a descrição de processos de desenvolvimento. A ferramenta de acompanhamento de processo (ARAÚJO, 1998) possibilita um efetivo acompanhamento do processo, através de uma interface gráfica que permite a navegação pelas atividades do processo, observando várias informações importantes, tais como o estado da atividade, artefatos consumidos e produzidos, recursos utilizados, métricas estabelecidas, etc. A ferramenta de execução de processo (ARAÚJO, 1998) é responsável por interagir com as atividades, proporcionando a execução do processo definido. Esta interação está associada à instanciação de atividades, recebimento de estímulos destas, controle de seu estado corrente, verificação da disponibilidade de recursos e artefatos necessários, recebimento de aviso de término de atividades, verificação de conclusão de pré-atividades e disponibilização de pós-atividades, se possível. A ferramenta de adaptação do processo permite que as alterações necessárias no processo de desenvolvimento sejam feitas de forma a não causar inconsistências no modelo existente.

A ferramenta de acompanhamento do processo é particularmente importante para o mecanismo de automatização do processo da Estação TABA, pois esta ferramenta é o ponto de partida para a execução e adaptação do processo, ou seja, as

ferramentas trabalham de forma conjunta e integrada, sendo que a de acompanhamento do processo é a interface com o engenheiro de software para a execução e adaptação do processo de desenvolvimento instanciado.

A Competência do Servidor de Conhecimento de Processo

A definição e a gerência do processo de desenvolvimento são tarefas complexas, mesmo dispondo de um repositório de componentes para se reutilizar. Para a definição, existem várias possibilidades de combinação dos componentes, além de ser necessário conhecimento sobre cada componente e sobre como os componentes se relacionam entre si. Para a gerência é necessário determinar quais dos componentes apresentados são adequados ao contexto desejado, além de analisar se a solução determinada pela aplicação do componente no novo contexto acarreta a solução esperada (ARAÚJO, 1998).

De fato, estas tarefas requerem conhecimento e experiência e, portanto, são fortes candidatas a um apoio baseado em conhecimento. O meta-ambiente e as ferramentas de acompanhamento, execução e adaptação do processo podem facilitar a realização destas atividades, principalmente se oferecerem suporte baseado em conhecimento para gerentes e engenheiros de software. Por esta razão, escolhemos o universo de discurso de processos de software para a construção de um protótipo de Servidor de Conhecimento.

Entretanto, este universo de discurso é bastante amplo, envolvendo um grande volume de conhecimento. Neste trabalho, optamos por considerar apenas o conhecimento envolvido na definição de um processo de software – o primeiro passo no tratamento de processos de software – para efeito da construção de um protótipo do Servidor de Conhecimento de Processo. Este conhecimento é a base para o apoio à descrição de processos de software no meta-ambiente, mas é, também, extremamente útil para outras ferramentas, como as ferramentas de acompanhamento, execução e adaptação de processo. A partir deste núcleo, o Servidor de Conhecimento poderá ser estendido para contemplar uma competência mais abrangente.

A Arquitetura do Servidor de Conhecimento de Processo

Levando em consideração a competência do Servidor de Conhecimento de Processo, a arquitetura geral dos Servidores de Conhecimento, mostrada na figura 4.7, foi instanciada na arquitetura mostrada na figura 4.8.

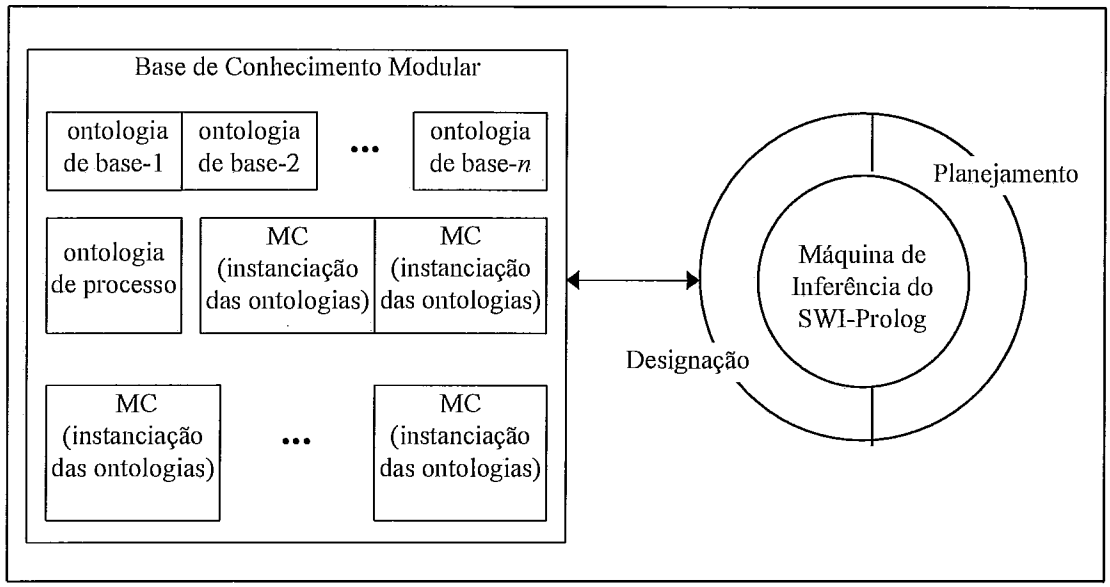


Figura 4.8 - O Servidor de Conhecimento de Processo na Estação TABA.

A base de conhecimento modular deve ser baseada em uma ontologia de processo de software e em outras ontologias, desenvolvidas como base para a ontologia de processo de software. As instanciações das ontologias, também, devem derivar módulos de conhecimento.

Os *templates* de resolvedores de problemas devem customizar a máquina de inferência para os tipos de problemas inerentes à competência do Servidor de Conhecimento, ou seja, a definição de processos de software. A definição de um processo envolve, tipicamente, dois tipos de problema genéricos: o planejamento e a designação. Tarefas de planejamento têm por objetivo construir um plano com base no estado inicial do mundo e no estado que se deseja atingir. O plano resultante é representado, de modo geral, como uma seqüência de ações que, quando executada partindo do estado inicial, deve conduzir ao estado meta (VALENTE, 1994). Tarefas de designação visam preencher uma estrutura com elementos, de modo que determinados requisitos e restrições sejam satisfeitos. Quando a estrutura a ser

preenchida é um plano, os elementos a serem designados às atividades do plano são geralmente recursos e o problema é então chamado de *programação* ou *escalonamento* (SUNDIN, 1994). Modelos de tarefa para estes dois tipos de problemas devem ser adaptados para o contexto da definição de processos de software, derivando os *templates* de resolvedores de problemas correspondentes.

Deve-se realçar que a infra-estrutura de conhecimento existente na Estação TABA comporta a arquitetura de Servidores de Conhecimento proposta. A máquina de inferência do SWI-Prolog está integrada ao ambiente e, portanto, as ontologias e suas instanciações poderão ser implementadas nesta linguagem, como objetos da classe Conhecimento. Os resolvedores genéricos de tipos de problemas, por sua vez, podem ser implementados em Eiffel como sub-classes da classe Assistente Inteligente.

4.5 - Conclusões do Capítulo

Neste capítulo, apresentamos a Estação TABA, um meta-ADS, cuja filosofia de integração considera a integração de conhecimento. A estrutura da Estação TABA busca distribuir as funcionalidades necessárias em componentes, capazes de interagir diretamente entre si de modo a obter a funcionalidade completa do ambiente. Dentre estes componentes, dois estão relacionados à questão do conhecimento: o componente *Conhecimento*, que provê a infra-estrutura para gerência e armazenamento de conhecimento, e o componente *Suporte Inteligente*, que reúne ferramentas baseadas em conhecimento para assistir os usuários do ambiente na realização de suas tarefas.

Entretanto, a infra-estrutura de conhecimento da Estação TABA não é suficiente para promover a integração de conhecimento. De fato, esta infra-estrutura é apenas um facilitador para a integração. Neste contexto, introduzimos a noção de Servidor de Conhecimento como uma estrutura capaz de promover a integração de conhecimento, servindo de apoio para a construção de sistemas baseados em conhecimento.

Basicamente, um Servidor de Conhecimento provê resolvedores genéricos de problemas e bases modulares de conhecimento baseadas em ontologias, permitindo, assim, o reuso de conhecimento de tarefa e de domínio, respectivamente. Uma vez

que o Servidor torna disponível conhecimento sobre um universo de discurso, o engenheiro de software pode se utilizar deste conhecimento para compreender este universo e para propor uma solução inicial para o problema. Desta forma, uma aquisição de conhecimento preliminar pode ser feita, sem consumir muito tempo dos especialistas. Navegando as ontologias do domínio, o engenheiro de software ganha um entendimento do problema e pode montar uma base de conhecimento inicial. Os modelos de tarefa, por sua vez, podem ser usados para estabelecer uma estratégia inicial para a resolução do problema. Adaptando um modelo de tarefa para problema em mãos e fazendo um mapeamento de seus papéis de conhecimento com o conhecimento de domínio correspondente, tem-se um protótipo inicial do sistema. Além disso, ao propor esta solução preliminar, o engenheiro de software pode identificar que tipos de conhecimento estão faltando e, assim, tem um mecanismo eficiente para guiar a segunda etapa da aquisição de conhecimento, agora junto aos especialistas.

Para materializar a idéia de Servidores de Conhecimento, optou-se por desenvolver um Servidor de Conhecimento de Processo de Software (SCP). No próximo capítulo, apresentamos uma ontologia de processo de software, utilizada como base para a derivação dos módulos de conhecimento para este Servidor. As ontologias são o elemento-chave para a construção de Servidores de Conhecimento, já que seus principais componentes, os módulos de conhecimento, são derivados delas e de suas instanciações.

No capítulo 6, outros aspectos do desenvolvimento do Servidor de Conhecimento de Processo são discutidos, tais como o desenvolvimento dos *templates* de resolvedores de problemas, a implementação do SCP, e o uso deste Servidor para apoiar a construção de um assistente inteligente para a definição de processos de software.

Capítulo 5

Uma Ontologia de Processo de Software

Neste capítulo, apresentamos a ontologia de processo de desenvolvimento de software desenvolvida para o Servidor de Conhecimento de Processo, utilizando o método de Engenharia de Ontologias, definido na seção 3.4.2.

Esta ontologia foi construída com o apoio de diversos especialistas em processo de software da COPPE/UFRJ, buscando-se um consenso sobre o vocabulário e as restrições a serem consideradas para processos de software no âmbito do Projeto TABA. Além destes especialistas do domínio, os seguintes trabalhos foram utilizados como fonte para a construção desta ontologia: (ISO 9000-3, 1991) (PAULK et al., 1993), (TRAVASSOS, 1994), (ROCHA et al., 1994), (TRAVASSOS et al., 1995), (WERNECK, 1995), (ROCHA et al., 1996), (WERNER et al., 1996) e (PRESSMAN, 1997).

A primeira atividade realizada foi a identificação do propósito e a especificação inicial dos requisitos da ontologia. Dada a complexidade do domínio, a abordagem de ontologias em níveis foi adotada. Assim, a ontologia de processo foi construída tendo por base ontologias centrais do domínio, a saber ontologias de atividade, procedimento e recurso. Para cada uma dessas ontologias, o método de Engenharia de Ontologias foi recursivamente aplicado.

Na especificação de requisitos, foram enumeradas questões de competência que serviram de base para a fase seguinte, a captura da ontologia. Nesta fase, foram elaborados modelos usando LINGO e conceitos e axiomas foram descritos utilizando

linguagem natural e exemplos. As questões de competência relacionadas foram utilizadas para avaliar as ontologias em desenvolvimento. Uma vez que há uma forte interação entre as ontologias centrais, estas foram sendo desenvolvidas de forma integrada, i.e., apoiadas nos conceitos e relações das demais ontologias já construídas.

Na formalização, optou-se pela lógica de primeira ordem, estabelecendo-se uma linguagem formal de primeira ordem sobre processo de software. Foram identificados as constantes e os predicados[1] compondo o alfabeto da linguagem e, a partir dele, os axiomas da ontologia foram formalizados. O processo de avaliação das ontologias estendeu-se, também, durante a formalização. Além disso, esta etapa provocou a revisão de alguns conceitos e relações, mostrando que há uma estreita interdependência entre as fases de captura e formalização de uma ontologia.

Este capítulo está estruturado da seguinte forma: na seção 5.1 é apresentada a etapa de identificação de propósito e especificação de requisitos da ontologia de processo de software. A competência da ontologia é definida e os cenários de utilização são descritos. Uma breve descrição textual do universo de discurso é apresentada e utilizada para decompor o problema. As seções 5.2, 5.3 e 5.4 apresentam, respectivamente, as ontologias de atividade, procedimento e recurso, incluindo suas formalizações. Na seção 5.5, a ontologia de processo volta à cena, em uma abordagem de ontologias em níveis. A seção 5.6 descreve, resumidamente, o processo de instanciação das ontologias. Finalmente, na seção 5.7, são apresentadas as conclusões do capítulo.

5.1 - Ontologia de Processo de Software: Especificação de Requisitos

A definição de uma ontologia de processo de software tem por objetivo apoiar a aquisição, organização, reuso e compartilhamento de conhecimento sobre processos de desenvolvimento de software. Para atingir este objetivo, a ontologia deve prover um vocabulário e um conjunto de axiomas fixando a semântica dos termos neste domínio de discurso.

[1] Como convenção, constantes e predicados aparecem neste texto em *itálico*, sendo que constantes são iniciadas por letras maiúsculas e predicados por letras minúsculas.

A ontologia foi projetada para dar suporte ao desenvolvimento de ferramentas baseadas no conhecimento sobre processos de software em um meta-ADS: o meta-ambiente TABA (ROCHA et al., 1990). Toda ferramenta baseada em conhecimento de processo que venha a se comprometer com esta ontologia, estará compartilhando um vocabulário comum, facilitando a comunicação entre os desenvolvedores e, o que é mais importante, abrindo caminho para permitir o compartilhamento e reuso de bases de conhecimento, seja no meta-ambiente, seja nos ambientes instanciados.

Ao construir uma ontologia de processo de software, estamos fixando interpretações para os elementos de um corpo de conhecimento sobre processos de software. Este conhecimento poderá ser usado pelo meta-ambiente para instanciar ambientes específicos e por ferramentas dos ambientes instanciados para prover suporte a um processo de desenvolvimento específico.

Descrição do Universo de Discurso

Um processo de desenvolvimento de software consiste de um conjunto de atividades relacionadas que têm lugar no desenvolvimento de um produto de software. Estas atividades são arranjadas segundo um ciclo de vida, e são realizadas através de procedimentos específicos para cada atividade.

A definição do ciclo de vida de um processo e dos procedimentos a serem utilizados na realização das atividades é fortemente dependente das características do produto que se pretende construir. O domínio de aplicação e o propósito do sistema guiam a escolha da tecnologia de desenvolvimento a ser aplicada e do paradigma a ser adotado em um projeto. Esses dois fatores, por sua vez, têm grande impacto na escolha de um modelo de ciclo de vida e dos procedimentos a serem adotados no processo. Assim, não existe um processo de desenvolvimento que seja adequado a qualquer projeto; ao contrário, para cada projeto, deve-se definir um processo apropriado.

As atividades de um processo são realizadas por pessoas, que podem desempenhar diferentes papéis nas diversas atividades do processo de desenvolvimento. Cada atividade gera um conjunto de artefatos e utiliza recursos. Estes artefatos podem ser documentos ou produtos de software (WERNER et al., 1996) (WERNECK, 1995).

Os modelos de ciclo de vida de sistemas descrevem as etapas do processo e as atividades a serem realizadas em cada etapa. A definição dessas etapas e atividades possibilita prover pontos de avaliação (*checkpoints*) para o controle gerencial das decisões tomadas durante o desenvolvimento e a vida do sistema (WERNECK, 1995).

Métodos são prescrições explícitas para a realização de uma ou mais atividades do ciclo de vida. Eles devem refletir um conjunto de diretivas para a aplicação sistemática de técnicas e instrumentos. As técnicas podem ser definidas como um conjunto de princípios para a execução de uma tarefa específica do processo. Os instrumentos tornam possível o uso de métodos e técnicas. Os métodos estabelecem uma seqüência de passos para o desenvolvimento e uma forma de tomar decisões ao longo das atividades do processo. Um método deve ser utilizado para apoiar as atividades fundamentais a serem realizadas no desenvolvimento de sistemas (WERNECK, 1995).

As atividades fundamentais do processo de desenvolvimento de um software estão relacionadas às atividades de gerência do processo de desenvolvimento, construção e avaliação da qualidade. Por isso, os métodos e as ferramentas que compõem um ADS são distribuídos considerando esses três grupos, em função da natureza da atividade para a qual dão suporte. Métodos e ferramentas de gerência apoiam o engenheiro de software e/ou o gerente de projeto no planejamento e acompanhamento gerencial do projeto. Métodos e ferramentas de construção auxiliam no processo de produção de um software, ao longo do seu ciclo de vida. Métodos e ferramentas de avaliação da qualidade apoiam desde o planejamento da qualidade até a verificação e validação do sistema (WERNER et al., 1996) (WERNECK, 1995).

Na gerência do processo são desempenhadas atividades muito importantes para o sucesso do desenvolvimento de um software, pois elas controlam os recursos e as atividades técnicas do processo. O objetivo principal da gerência é garantir que o software será entregue dentro do prazo estimado, de acordo com os custos estabelecidos, e com os atributos funcionais e de qualidade requisitados pelo usuário. A gerência controla as atividades de construção, visando desenvolver um plano de desenvolvimento que possa ser executado. Suas atividades compreendem o planejamento do projeto, com estimativas de custos, cronograma e identificação dos

recursos necessários e acompanhamento dos recursos, qualidade e produtividade (WERNECK, 1995).

A definição de um processo de software envolve informações como: as atividades a serem realizadas, recursos humanos envolvidos e suas responsabilidades dentro do processo, além de produtos consumidos e gerados, dentre outras (WERNER et al., 1996). A seguir são apresentados alguns dos principais conceitos relacionados à modelagem de processos de desenvolvimento de software (TRAVASSOS et al., 1995):

- **Atividades**: são as tarefas ou trabalhos a serem realizados. Uma atividade requer recursos e pode consumir ou produzir artefatos. Para sua realização, uma atividade pode adotar um procedimento. Uma atividade pode ser decomposta em outras atividades. Além disso, atividades, em qualquer nível, podem depender da finalização de outras atividades, denominadas pré-atividades.
- **Artefatos**: são produtos de software produzidos ou consumidos por atividades durante a sua realização. São exemplos de artefatos: manuais de qualidade, manuais de revisão, diagramas de fluxos de dados, diagramas de objetos, código fonte, etc. Um artefato pode ser decomposto em outros artefatos (composição de artefatos).
- **Procedimentos**: são condutas bem estabelecidas e ordenadas para a realização de atividades. Alguns procedimentos podem ser parcialmente automatizados por ferramentas de software. São exemplos de procedimentos: métodos de construção de sistemas, tal como o método de Booch (BOOCH, 1994) para o desenvolvimento orientado a objetos, técnicas de avaliação da qualidade, tais como inspeções e walkthroughs, roteiros diversos para a produção de documentos, normas de programação, etc.
- **Recursos**: são as pessoas, as ferramentas de software, os equipamentos, ou quaisquer outros recursos necessários à execução de uma atividade. Um recurso humano, especificamente, desempenha um papel na execução das atividades do processo.

- **Processos**: são coleções de atividades relacionadas que têm lugar durante o desenvolvimento de um produto.

Cenários de Utilização

Sabemos que não existe um processo único que seja adequado às particularidades de todo e qualquer desenvolvimento de software. Em função das características específicas de um domínio de aplicação, ou de um projeto, um processo de desenvolvimento deve ser definido. O apoio à definição de processos de software é a competência da ontologia definida neste capítulo. Esta ontologia é parte essencial do Servidor de Conhecimento de Processos (SCP), descrito na seção 4.5.

Além disso, estamos interessados em processos automatizados ou, pelo menos, semi-automatizados. Para cada procedimento dentro do processo que for passível de, pelo menos, suporte parcialmente automatizado, devem ser designadas ferramentas. Essas ferramentas não devem ser vistas como elementos isolados, mas sim integradas, formando um todo coerente, isto é, um ADS.

Neste contexto, meta-ambientes assumem um importante papel: dadas as características de um projeto, ou de um conjunto de projetos, o meta-ambiente instancia um ADS adequado às suas demandas. Para tal, é fundamental que o meta-ambiente tenha conhecimento sobre processos de desenvolvimento de software, de modo a prover suporte às tarefas de definição de um processo e instanciação de um ambiente específico. Esta é a principal motivação para a construção da ontologia sobre processos definida neste capítulo: considerar o SCP dentro do contexto de um meta-ADS.

Uma vez definido o processo, um ADS pode ser instanciado e o conhecimento sobre o processo do ADS recém-criado pode ser para ele disponibilizado. Assim, este corpo de conhecimento pode ser compartilhado pelas ferramentas do ADS que, por ventura, venham precisar manipular alguma parte do mesmo. De fato, o conhecimento disponibilizado para o ambiente instanciado é um extrato do conhecimento disponível para o meta-ambiente. A figura 5.1 ilustra os cenários de usos da ontologia de processo de software.

É importante lembrar que o controle e o acompanhamento de projetos de software não são objetivos do SCP e, portanto, não são requisitos de nossa ontologia.

Para que o controle e o acompanhamento de projetos possam ser adequadamente considerados, uma outra ontologia, construída em conformidade com e estendendo a ontologia aqui proposta, será necessária. Este contexto impõe muitos outros requisitos e revelam vários outros conceitos, que deverão ser tratados em um trabalho futuro.

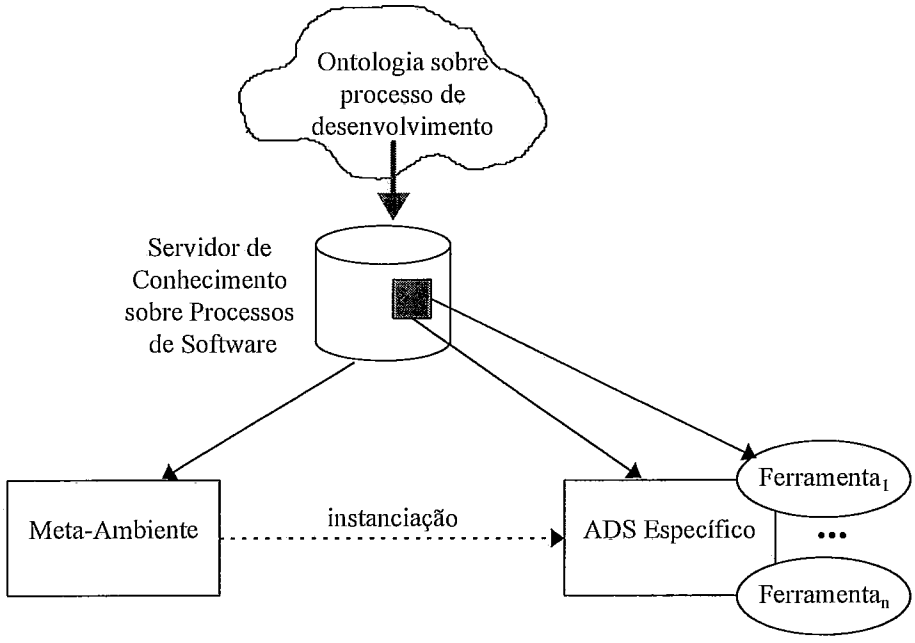


Figura 5.1 - Usos Projetados para a Ontologia no Contexto de um Meta-ADS.

Decomposição do Domínio de Interesse

Analisando a descrição do universo de discurso anteriormente apresentada, é possível notar que o contexto da definição de um processo de software é bastante complexo e, portanto, construir uma ontologia sobre processos de desenvolvimento de software não é uma tarefa fácil. Como premissa básica, é fundamental termos em mente o critério de comprometimento ontológico mínimo. Com base nesse critério, a ontologia deve descrever aspectos gerais, válidos para quaisquer processos, contendo apenas seus elementos essenciais. A adição de detalhes em uma ontologia a torna mais específica e, portanto, menos reutilizável. Idealmente, a ontologia deve ser mantida tão simples e ampla quanto possível.

No entanto, mesmo considerando apenas os aspectos gerais de processos, este domínio ainda é extremamente complexo. Necessitamos, pois, de um mecanismo de decomposição, que nos permita construir a ontologia por partes. Assim, a estratégia

adotada consistiu em se definir sub-domínios do domínio de processos de software, e construir ontologias básicas sobre esses sub-domínios. Uma vez definidas as ontologias básicas, estas foram utilizadas de forma integrada para construir a ontologia de processo de software, em uma abordagem de ontologias em níveis.

Analisando a descrição do universo de discurso, é possível perceber três conceitos que são fundamentais no contexto de processos de software: atividade, procedimento e recurso. Estes foram escolhidos como base para a construção das ontologias centrais, apresentadas na seqüência. Na seção 5.6, retornarmos à ontologia de processo de software, em uma abordagem de ontologias em níveis.

5.2 - Ontologia de Atividade

Identificação de Propósito e Especificação de Requisitos

O conceito de atividade está no cerne de qualquer modelo de processo de software. De fato, um processo de desenvolvimento é sempre orientado a atividade e, portanto, a habilidade de representar adequadamente atividades é um aspecto fundamental para uma ontologia de processo de software.

Atividades podem ocorrer em vários níveis, desde uma tarefa elementar até uma etapa do processo de desenvolvimento. Utilizaremos o conceito de atividade com o intuito de representar todo esse espectro. Assim, uma atividade é uma primitiva de transformação, que utiliza artefatos de entrada para gerar artefatos de saída, apoiada por recursos.

Uma vez que a ontologia de atividade deve servir de base para a ontologia de processo, seu uso planejado é o mesmo, ou seja, a definição de processos de desenvolvimento de software. Assim, tal ontologia deve ser capaz de responder a questões como:

1. Qual a natureza de uma atividade?
2. Como uma atividade pode ser decomposta?
3. Que atividades devem anteceder uma dada atividade?
4. Que artefatos são consumidos por uma determinada atividade?
5. Que artefatos são produzidos por uma determinada atividade?
6. Qual a natureza de um artefato?

7. Que recursos são necessários para que uma atividade possa ser realizada?
8. Que procedimentos podem ser adotados na realização de uma atividade?

Captura e Formalização da Ontologia

Analisando as questões de competência anteriormente relacionadas, identificamos alguns aspectos bastante relevantes no domínio de atividades, que foram alvo desta ontologia:

- taxonomia de atividades (questão 1)
- decomposição de atividades (questão 2);
- encadeamento de atividades (questão 3);
- atividades como primitivas de transformação (questões 4, 5 e 6);
- recursos requeridos por atividades (questão 7); e
- adoção de procedimentos na realização de atividades (questão 8).

Os dois últimos aspectos mostram interações entre a ontologia de atividade e as ontologias de recurso e procedimento, respectivamente, e foram tratados por ocasião do estudo destes sub-domínios.

Taxonomia de Atividades

No contexto de processos de desenvolvimento de software, atividades, quanto a sua natureza, podem ser classificadas em: atividades de construção, atividades de gerência e atividades de avaliação da qualidade. As atividades de avaliação da qualidade podem, ainda, ser divididas em atividades de certificação e atividades de teste. A figura 5.2 mostra a taxonomia de atividades.

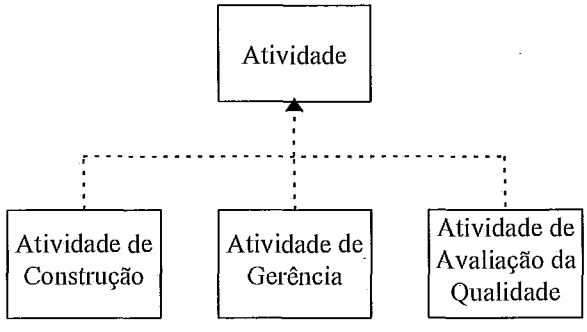


Figura 5.2 - Taxonomia de Atividades.

As *atividades de construção* são aquelas atividades diretamente relacionadas ao processo de construção do software, ao longo do seu ciclo de vida. Exemplos de atividades de construção, independentemente do ciclo de vida adotado, incluem: levantamento de requisitos, modelagem, projeto e codificação, entre outras.

As *atividades de gerência* são aquelas relacionadas ao planejamento e acompanhamento gerencial do projeto, tais como elaboração, execução, monitoramento, controle e revisão do plano de projeto.

As *atividades de avaliação da qualidade* são aquelas relacionadas com a garantia da qualidade do produto em desenvolvimento e do processo de software utilizado, tais como revisões e inspeções de produtos (intermediários ou finais) do desenvolvimento e testes.

Para formalizar a existência de diferentes tipos de atividades, definimos os seguintes predicados, representando cada um dos tipos identificados na taxonomia: *atividade(a)*, indicando que *a* é uma atividade; *atconstrução(a)*, indicando que *a* é uma atividade de construção; *atgerência(a)*, indicando que *a* é uma atividade de gerência; e *atavqualidade(a)*, indicando que *a* é uma atividade de avaliação da qualidade.

É bom lembrar que, conforme discutido na seção 3.3.4, a notação de sub-tipo empregada na figura 5.2, reflete os seguintes axiomas epistemológicos[2]:

$(\forall\ a)\ (atconstrução(a) \rightarrow atividade(a))$	(AE1)
$(\forall\ a)\ (atgerência(a) \rightarrow atividade(a))$	(AE2)
$(\forall\ a)\ (atavqualidade(a) \rightarrow atividade(a))$	(AE3)

Uma vez que axiomas desta natureza são implicitamente descritos pela notação de LINGO, eles não serão mais apresentados neste trabalho. Sempre que a notação para sub-tipos for empregada, assumimos que existem axiomas deste tipo.

[2] Axiomas epistemológicos, em contraste com os axiomas ontológicos, são derivados simplesmente da estrutura dos conceitos e não de seus significados particulares. Portanto, podem ser derivados automaticamente por uma ferramenta para edição de ontologias, quando tais estruturas forem utilizadas. A numeração desses axiomas será precedida das letras AE para diferenciá-los dos axiomas ontológicos, cuja numeração será precedida apenas pela letra A.

Decomposição de Atividades

O conceito de atividade está sendo utilizado para cobrir um amplo espectro de atividades, incluindo atividades macroscópicas e atividades elementares. Contudo, é importante identificar que pequenas atividades compõem uma atividade maior. Para tal, introduzimos os conceitos de super-atividade e sub-atividade, como mostram os papéis na figura 5.3.

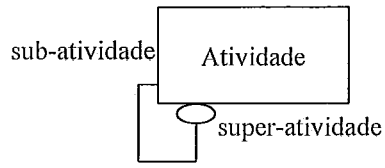


Figura 5.3 - Decomposição de Atividades.

Uma *super-atividade* é uma atividade que não pode ser realizada diretamente, isto é, ela é composta de outras atividades menores e sua realização se dá, na realidade, através da realização dessas sub-atividades. Uma *sub-atividade*, por sua vez, é uma atividade que compõe uma atividade maior, a sua super-atividade.

O predicado *subatividade*(a_1, a_2) indica que a atividade a_1 é uma sub-atividade (ou é parte) da atividade a_2, enquanto o predicado *superatividade*(a_2,a_1) indica que a_2 é uma super-atividade de (ou é um todo, cuja uma das partes é) a_1. Estes predicados estão relacionados segundo a seguinte sentença:

$$(\forall\, a_1, a_2)\ (subatividade(a_1, a_2) \leftrightarrow superatividade(a_2, a_1)) \qquad \text{(A1)}$$

Uma vez que a lógica de primeira ordem não é uma lógica poli-sortida, é necessário definir axiomas de consolidação[3] que estabeleçam que tipos de objetos podem ser utilizados como argumentos em um predicado. Assim, o seguinte axioma de consolidação deve ser observado:

$$(\forall\, a_1, a_2)\ (subatividade(a_1, a_2) \rightarrow atividade(a_1) \wedge atividade(a_2)) \quad \text{(AC1)}$$

[3] Axiomas de consolidação têm por objetivo verificar a coerência das informações existentes e não representam conseqüências lógicas, isto é, não derivam nova informação. Para diferenciá-los dos outros tipos de axiomas, a numeração dos axiomas de consolidação será precedida pelas letras AC.

Este axioma estabelece que os argumentos a_1 e a_2 do predicado *subatividade* têm de ser atividades.

Deve-se realçar que os conceitos de super e sub-atividade são relativos, ou seja, uma atividade a_i pode ser uma sub-atividade de uma atividade a e uma super-atividade para um conjunto de atividades $a_{i,1}$, $a_{i,2}$, ..., $a_{i,n}$. A decomposição de atividades, assim como qualquer composição, é transitiva, isto é, se uma atividade a_1 é uma sub-atividade da atividade a_2 e a_2 é uma sub-atividade da atividade a_3, então a_1 é também uma sub-atividade de a_3. O axioma epistemológico abaixo formaliza esta propriedade da relação de composição.

$$(\forall\, a_1, a_2, a_3)\ (subatividade(a_1, a_2) \wedge subatividade(a_2, a_3) \rightarrow subatividade(a_1, a_3)) \qquad \text{(AE4)}$$

A decomposição de atividades é também assimétrica, isto é, se uma atividade a_1 é uma sub-atividade da atividade a_2, então a_2 não pode ser uma sub-atividade de a_1

.

$$(\forall\, a_1, a_2)\ (subatividade(a_1, a_2) \rightarrow \neg\, subatividade(a_2, a_1)) \qquad \text{(AE5)}$$

Para registrarmos o fato de uma atividade não ser mais decomponível e, portanto, ser passível de realização direta, introduzimos o conceito de *atividade elementar*: uma atividade elementar é aquela que não pode ser decomposta e, portanto, é passível de realização direta. Seu equivalente inverso é o conceito de *macro-atividade*: uma macro-atividade é aquela atividade que não é parte de nenhuma outra.

Os predicados *atividadeelementar(a)* e *macroatividade(a)* indicam, respectivamente, que uma atividade *a* é uma atividade elementar ou uma macro-atividade. Estes predicados são definidos em termos dos conceitos de sub-atividade e super-atividade, como mostram as sentenças abaixo:

$$(\forall\, a)\ (atividadeelementar(a) \leftrightarrow \neg\, (\exists\, a_1)\ (subatividade(a_1, a))) \qquad \text{(A2)}$$

$$(\forall\, a)\ (macroatividade(a) \leftrightarrow \neg\, (\exists\, a_1)\ (superatividade(a_1, a))) \qquad \text{(A3)}$$

Encadeamento de Atividades

Atividades são realizadas seguindo uma ordem específica. Um aspecto primordial no contexto de um processo de desenvolvimento é capturar o encadeamento de atividades. Para tal, introduzimos os conceitos de pré-atividade e pós-atividade, mostrados na figura 5.4.

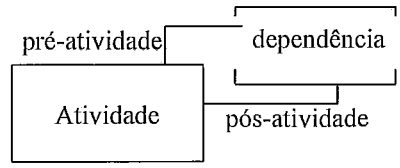


Figura 5.4 - Encadeamento de Atividades.

Uma atividade a_1 é dita uma *pré-atividade* de uma atividade a_2, se a_1 precisa ser realizada para que a_2 também o seja. A atividade a_2, por sua vez, é dita uma *pós-atividade* de a_1, já que ela só pode ser realizada após a realização de a_1. Assim, se a_1 é uma pré-atividade de a_2, então a_2 é uma pós-atividade de a_1.

Estes conceitos foram formalizados através dos predicados *preatividade*(a_1 ,a_2) e *posatividade*(a_2 ,a_1), indicando que a_1 é uma pré-atividade de (antecede) a_2 ou, de forma inversa, que a_2 é uma pós-atividade de (sucede) a_1.

$$(\forall\, a_1,\, a_2)\,(preatividade(a_1,\, a_2) \leftrightarrow posatividade(a_2,\, a_1)) \qquad \text{(A4)}$$

Assim como a relação de decomposição, a relação de precedência entre atividades é transitiva e assimétrica, como mostram as sentenças abaixo.

$$(\forall\, a_1,\, a_2,\, a_3)(preatividade(a_1,\, a_2) \wedge preatividade(a_2,\, a_3) \rightarrow preatividade(a_1,\, a_3)) \qquad \text{(A5)}$$

$$(\forall\, a_1,\, a_2)\,(preatividade(a_1,\, a_2) \rightarrow \neg\, preatividade(a_2,\, a_1)) \qquad \text{(A6)}$$

Além disso, o seguinte axioma de consolidação deve ser observado:

$$(\forall\, a_1,\, a_2)\,(preatividade(a_1,\, a_2) \rightarrow atividade(a_1) \wedge atividade(a_2)) \qquad \text{(AC2)}$$

Atividades como Primitivas de Transformação

Durante a realização de atividades, artefatos de entrada - os insumos para a atividade - são transformados em artefatos de saída - os produtos da atividade. Neste sentido, os artefatos de entrada são "incorporados" aos artefatos de saída. Dizemos "incorporados" para denotar que o conteúdo (informações) de um artefato de entrada é, de alguma forma, incorporado ao artefato de saída. A figura 5.5 mostra estas relações.

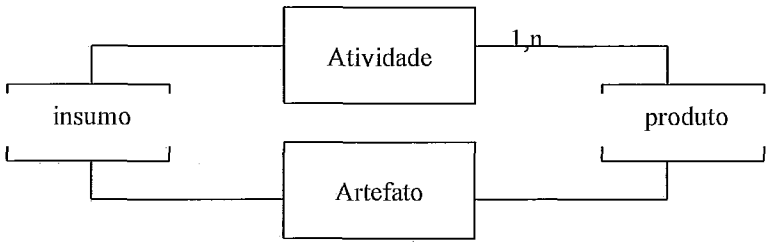


Figura 5.5 - Atividades como Primitivas de Transformação.

Artefatos possuem uma importante propriedade que define seu tipo. Uma vez que a reutilização de software tem sido utilizada cada vez mais no desenvolvimento de aplicações complexas, é importante distinguir *componentes de software* como um tipo especial de artefato. Componentes de software são artefatos oriundos de uma biblioteca de componentes, tais como classes de uma biblioteca de classes, um framework de reuso ou um padrão gerativo. *Artefatos de código*, por sua vez, são artefatos que consistem de porções de código-fonte, passíveis de execução, tais como programas, sub-programas, rotinas, classes, procedimentos ou funções, que foram gerados no próprio desenvolvimento. Finalmente, os demais artefatos são ditos *documentos*. Utilizamos o predicado *artefato(s,t)* para denotar que *s* é um artefato do tipo *t*, onde *t* assume um dos seguintes valores: *{Componente, Documento, Código}*.

As relações *insumo* e *produto*, mostradas na figura 5.5, foram formalizadas pelos predicados *insumo(s,a)*, denotando que o artefato *s* é um insumo para a atividade *a*, e *produto(s,a)*, denotando que o artefato *s* é um produto da atividade *a*. Para garantir sua integridade, os seguintes axiomas de consolidação devem ser observados:

$$(\forall\ a,\ s)\ (\ insumo(s,\ a) \rightarrow artefato(s,*) \wedge atividade(a)) \qquad \text{(AC3)}$$

$$(\forall\ a,\ s)\ (produto(s,\ a) \rightarrow artefato(s,*) \wedge atividade(a)) \qquad \text{(AC4)}$$

onde o asterisco (*) indica que não importa o valor atribuído para este argumento[4].

A partir dos predicados *insumo* e *produto*, podemos elaborar uma outra definição para o conceito de *pré-atividade*:

Uma atividade a_1 é uma pré-atividade de uma atividade a_2, se e somente se, existe pelo menos um artefato s que é um produto de a_1 e um insumo para a_2.

$$(\forall\ a_1, a_2)\ (preatividade(a_1, a_2) \leftrightarrow (\exists\ s)\ (insumo(s, a_2) \wedge produto(s, a_1)) \qquad \text{(A7)}$$

Podemos, ainda, estabelecer relações entre insumos e produtos de uma atividade composta, dadas pelos seguintes axiomas:

Um artefato s é um insumo para uma atividade composta a, onde a é decomposta nas sub-atividades $\{a_1, a_2, ..., a_n\}$, se s é um insumo para alguma atividade a_i, mas não é um produto de nenhuma outra atividade a_k, onde a_i e $a_k \in \{a_1, a_2, ..., a_n\}$.

$$(\forall\ a, a_1, ..., a_n, s)\ (subatividade(a_1, a) \wedge ... \wedge subatividade(a_n, a) \wedge insumo(s, a_i) \wedge ((\neg\exists\ a_k)\ produto(s, a_k)) \rightarrow insumo(s, a)) \qquad \text{(A8)}$$

Um artefato s é um produto de uma atividade composta a, onde a é decomposta nas sub-atividades $\{a_1, a_2, ..., a_n\}$, se s é um produto de alguma atividade $a_i \in \{a_1, ..., a_n\}$, e é um insumo para alguma atividade b, sendo que $b \notin \{a_1, a_2, ..., a_n\}$.

$$(\forall\ a, a_1, ..., a_n, s)\ (subatividade(a_1, a) \wedge ... \wedge subatividade(a_n, a) \wedge produto(s, a_i) \wedge ((\exists\ b)\ (insumo(s, b) \wedge b \notin \{a_1, a_2, ..., a_n\})) \rightarrow produto(s, a)) \qquad \text{(A9)}$$

[4] Sempre que, em um axioma, um dos argumentos de um predicado puder assumir um valor arbitrário, representamos este argumento com um asterisco (*).

A cardinalidade (1,n) do conceito *atividade* para com a relação *produto*, mostrada na figura 5.5, reflete o seguinte axioma: toda atividade tem de produzir pelo menos um artefato.

$$(\forall\ a)\ (atividade\ (a) \rightarrow (\exists\ s)\ (produto(s,a)) \qquad (A10)$$

Entretanto, assim como no caso das relações "sub-tipo-de", os axiomas desta natureza não serão mais apresentados, já que são implicitamente descritos pela notação de LINGO[5].

Artefatos, assim como atividades, são elementos compostos. Um artefato pode ser composto de outros sub-artefatos, como mostra a figura 5.6. Axiomas inerentes à relação de composição, similares aos descritos para atividades, são igualmente válidos, mas não são mostrados, por estarem implícitos pela notação de LINGO.

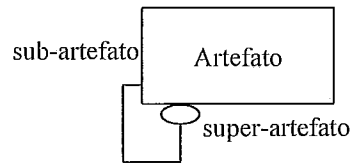


Figura 5.6 - Decomposição de Artefato.

A figura 5.7 mostra o modelo completo da ontologia de atividade e a tabela 5.1 apresenta o dicionário dos termos básicos correspondentes.

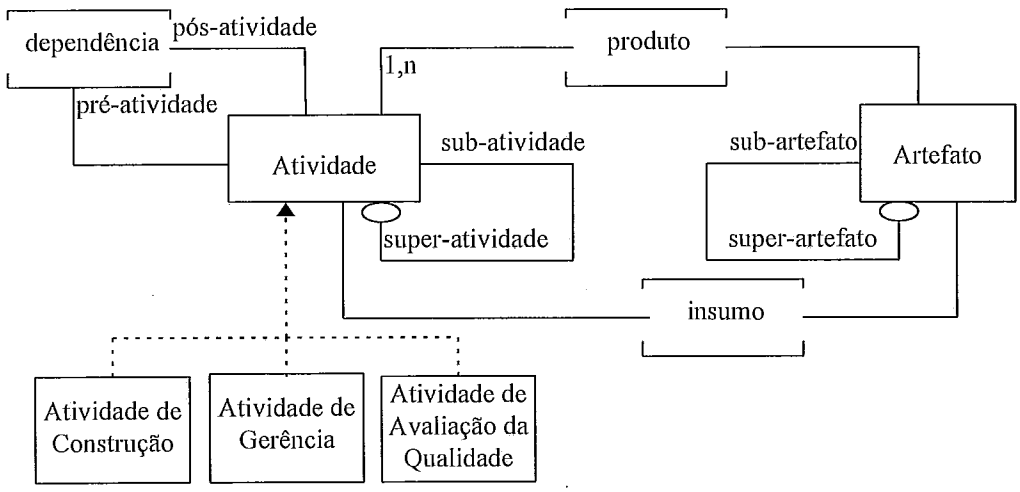


Figura 5.7 - Ontologia de Atividade.

[5] Uma ferramenta para edição de ontologias empregando LINGO poderia gerar tais axiomas automaticamente e, portanto, não haveria necessidade de escrevê-los.

Tabela 5.1 - Dicionário de Termos da Ontologia de Atividade.

Artefato	um **insumo** para, ou um **produto** de, uma **atividade**, no sentido de ser um objeto de transformação da atividade. Uma importante propriedade de um artefato é o seu tipo. Os tipos de **artefatos** incluem: **artefatos de código**, **componentes de software** e **documentos**.
Artefato de Código	porção de código-fonte, passível de execução, gerada no próprio desenvolvimento. Ex: programas, sub-programas, classes, frameworks, rotinas, funções, etc.
Atividade	ação que transforma **artefatos** de entrada (**insumos**) em **artefatos** de saída (**produtos**). Em função de sua natureza, atividades podem ser classificadas em **atividades de gerência**, **atividades de construção** e **atividades de avaliação da qualidade**.
Atividade de Avaliação da Qualidade	**atividade** relacionada com a garantia da qualidade do produto em desenvolvimento ou do processo utilizado neste desenvolvimento. Ex: revisões e inspeções de produtos (intermediários ou finais) do desenvolvimento e testes.
Atividade de Construção	**atividade** diretamente relacionada ao processo de construção do software. Ex: especificação de requisitos, modelagem, projeto, codificação, etc.
Atividade de Gerência	**atividade** relacionada ao planejamento e acompanhamento gerencial do projeto. Ex: elaboração, execução, monitoramento, controle e revisão do plano de projeto.
Atividade Elementar	**atividade** que não pode mais ser decomposta, isto é, não é **super-atividade** de nenhuma outra e, portanto, é passível de realização direta.
Componente de Software	**artefato** oriundo de uma biblioteca de componentes. Ex: uma classe de uma biblioteca de classes, um framework de reuso, um padrão gerativo, etc.
Documento	**artefato** de software não passível de execução. Ex: documento de especificação de requisitos, plano de projeto, plano de qualidade, relatório de avaliação da qualidade, etc.
Insumo	relação entre um **artefato** e uma **atividade**, indicando que o **artefato** é utilizado como "matéria-prima" pela **atividade**, sendo de alguma forma incorporado a outro **artefato**, o **produto** da **atividade**.
Macro-atividade	**atividade** que não é parte de outra, isto é, não é **sub-atividade** de nenhuma outra **atividade**.
Pós-atividade	faceta da relação de dependência entre duas **atividades** a_1 e a_2. Se a realização de a_2 depende da realização de a_1 então a_2 é dita uma pós-atividade de a_1.
Pré-atividade	faceta da relação de dependência entre duas **atividades** a_1 e a_2. Se a_1 precisa ser realizada para que a_2 o seja então, a_1 é dita uma pré-atividade para a_2.
Produto	relação entre um **artefato** e uma **atividade**, indicando que o **artefato** é produzido pela **atividade**.
Sub-artefato	faceta da relação de composição entre dois **artefatos** s_1 e s_2. Se s_2 é parte de s_1 então s_2 é dito um sub-artefato de s_1.
Super-artefato	faceta da relação de composição entre dois **artefatos** s_1 e s_2. Se s_1 é decomposto em outros artefatos, dentre eles s_2, então s_1 é dito um super-artefato de s_2.
Sub-atividade	faceta da relação de composição entre duas **atividades** a_1 e a_2. Se a_2 é parte de a_1 então a_2 é dita uma sub-atividade de a_1.
Super-atividade	faceta da relação de composição entre duas **atividades** a_1 e a_2. Se a_1 é decomposta em outras atividades, dentre elas a_2, então a_1 é dita uma super-atividade de a_2.

5.3 - Ontologia de Procedimento

Propósito e Especificação de Requisitos

Na definição de um processo, é importante determinar como as atividades serão realizadas. Para tal, procedimentos devem ser adotados na realização de atividades. Assim é preciso saber se um determinado procedimento *pode ser* adotado na realização de uma atividade, já que certos procedimentos são definidos para apoiar tipos específicos de atividades. Um procedimento para avaliação da qualidade, por exemplo, só pode apoiar atividades de avaliação da qualidade.

Outro aspecto a ser contemplado na ontologia de procedimento diz respeito à sua adequação em relação a uma tecnologia de desenvolvimento e a um paradigma. Além disso, o contexto da ontologia inclui também a (semi-)automatização de procedimentos e, portanto, é necessário considerar as ferramentas a serem usadas.

Podemos listar as seguintes questões de competência para a ontologia de procedimentos apresentada nesta seção:

1. Que procedimentos podem ser adotados na realização de uma atividade?
2. Qual a natureza de um determinado procedimento?
3. A que tecnologia de desenvolvimento e/ou paradigma um procedimento é adequado?
4. Que ferramentas podem ser utilizadas para (semi-)automatizar um procedimento?

Captura e Formalização da Ontologia

As questões de competência anteriormente relacionadas nos remetem aos seguintes aspectos a serem tratados pela ontologia de procedimento:

- taxonomia de procedimentos (questão 2);
- adequação de procedimentos (questões 1 e 3); e
- (semi-)automatização de procedimentos (questão 4).

Dentre estes aspectos, retardamos a discussão do último item, uma vez que ele trata da interação com a ontologia de recurso, trabalhada na próxima seção.

Taxonomia de Procedimentos

Procedimentos, quanto à sua natureza, podem ser classificados em métodos, técnicas e diretrizes. Métodos e técnicas podem ser diferenciados, ainda, em relação

ao tipo de atividade que podem apoiar. Diretrizes, por sua vez, podem ser roteiros ou normas. A figura 5.8 mostra a taxonomia de procedimentos.

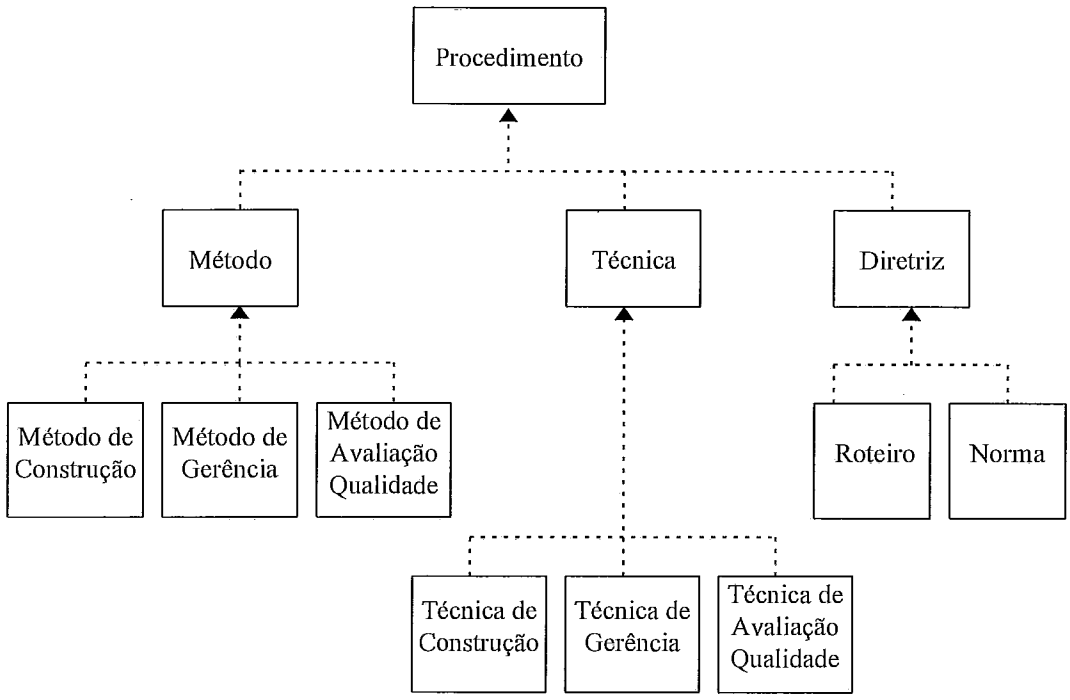


Figura 5.8 - Taxonomia de Procedimentos.

Um *método* é um procedimento sistemático para a realização de uma ou mais atividades, definindo passos e heurísticas. Uma *técnica* é também um procedimento para a realização de uma atividade, contudo, menos rígido e detalhado. Técnicas descrevem apenas aspectos gerais para a realização da atividade, sem impor um conjunto de sub-atividades para esta realização. Ou seja, a diferença básica entre um método e uma técnica é exatamente o caráter sistemático dos métodos, que definem padrões em termos de sub-atividades a serem executadas para a realização de uma super-atividade.

Com base no tipo de atividade que podem apoiar, métodos são classificados em *métodos de construção*, *métodos de gerência* e *métodos de avaliação da qualidade*. Métodos de construção, incluem, entre outros, métodos de análise e projeto, tal como o método de Booch (BOOCH, 1994) para o desenvolvimento orientado a objetos. Métodos de gerência incluem, por exemplo, os métodos de

estimativa de custos, tal como o COCOMO (BOEHM, 1981). O método Rocha (ROCHA, 1987) é um exemplo de um método de avaliação da qualidade.

Técnicas, assim como os métodos, podem ser classificadas quanto ao tipo de atividade que apoiam: *técnicas de construção* (p.ex.: técnicas top-down e bottom-up), *técnicas de gerência* (p.ex.: técnica para estimativa de tamanho por linhas de código (PRESSMAN, 1997)) e *técnicas de avaliação da qualidade* (p.ex.: inspeções, walkthroughs, testes de caixa branca e testes de caixa preta).

Finalmente, *diretrizes* são classificadas em *roteiros* e *normas*. Roteiros representam procedimentos para a elaboração de documentos e, portanto, só podem ser utilizados em atividades que produzam documentos. Normas representam procedimentos que visam estabelecer padrões para a realização de atividades, excluindo a elaboração de documentos, como por exemplo, normas de programação.

Para formalizar o conceito de procedimento e seus vários tipos, definimos os seguintes predicados: *procedimento(p)*, denotando que *p* é um procedimento; *método(m)*, denotando que *m* é um método; *técnica(t)*, denotando que *t* é uma técnica; *diretriz(d)*, denotando que *d* é uma diretriz; *metconstrução(m)*, denotando que *m* é um método de construção; *metgerência(m)*, denotando que *m* é um método de gerência; *metavqualidade(m)*, denotando que *m* é um método de avaliação da qualidade; *tecconstrução(t)*, denotando que *t* é uma técnica de construção; *tecgerência(t)*, denotando que *t* é uma técnica de gerência; *tecavqualidade(t)*, denotando que *t* é uma técnica de avaliação da qualidade; *roteiro(r)*, denotando que *r* é um roteiro; e finalmente, *norma(n)*, denotando que *n* é uma norma.

Adequação de Procedimentos

Durante a definição de um processo de desenvolvimento de software, certamente a seguinte pergunta precisa ser respondida: que procedimentos podem ser adotados na realização das diversas atividades do processo? Uma vez que a competência desta ontologia é, exatamente, a definição de processos, é necessário descrever quando um procedimento pode ser adotado por uma atividade. A figura 5.9 mostra o modelo correspondente.

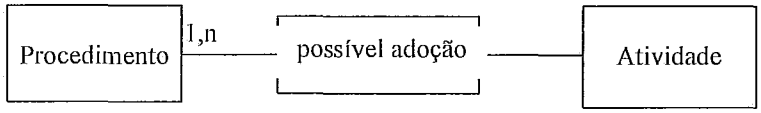


Figura 5.9 - Relação entre Procedimento e Atividade.

Definimos o predicado *possíveladoção(p,a)* para denotar que o procedimento *p* pode ser adotado na realização da atividade *a*, sendo que o seguinte axioma de consolidação deve ser observado:

$$(\forall\, p,a)\ (possíveladoção(p,a) \rightarrow procedimento(p) \wedge atividade(a)) \quad \text{(AC5)}$$

Basicamente, há uma forte relação entre tipos de procedimentos e tipos de atividades. Roteiros só podem ser adotados por atividades que produzam documentos; métodos e técnicas de construção por atividades de construção; métodos e técnicas de gerência por atividades de gerência; e métodos e técnicas de avaliação da qualidade por atividades de avaliação da qualidade. Estas dependências estão expressas nos seguintes axiomas de consolidação:

Se uma atividade *a* pode adotar um roteiro *r* como procedimento, então *a* gera pelo menos um artefato *s*, tal que *s* é um documento.

$$(\forall\, a, r)\ (possíveladoção(r,a) \wedge roteiro(r) \rightarrow (\exists\ s)\ (produto(s,a) \wedge artefato(s,Documento)) \quad \text{(AC6)}$$

Se *p* é um método ou técnica de construção e *p* pode ser adotado por *a*, então *a* é uma atividade de construção.

$$(\forall\, a, p)\ ((metconstrução(p) \vee tecconstrução(p)) \wedge possíveladoção(p,a) \rightarrow atconstrução(a)) \quad \text{(AC7)}$$

Se *p* é um método ou técnica de gerência e *p* pode ser adotado por *a*, então *a* é uma atividade de gerência.

$$(\forall\, a, p)\ ((metgerência(p) \vee tecgerência(p)) \wedge possíveladoção(p,a) \rightarrow atgerência(a)) \quad \text{(AC8)}$$

Se *m* é um método ou técnica de avaliação da qualidade e *p* pode ser adotado por *a*, então *a* é uma atividade de avaliação da qualidade.

$$(\forall\ a, m)\ (\ metavqualidade\ (p) \vee\ tecavqualidade(p)) \wedge possíveladoção\ (p,a) \rightarrow atavqualidade(a)) \qquad \text{(AC9)}$$

Além disso, há restrições no que se refere à granularidade das atividades. Um método impõe uma certa decomposição para uma atividade e, portanto, não pode ser adotado na realização de atividades elementares, já que estas não são passíveis de decomposição. Técnicas, por outro lado, só podem ser adotadas na realização de atividades elementares, já que não descrevem como a atividade deve ser decomposta. É exatamente o caráter sistemático dos métodos que os diferencia de técnicas. Ou seja, ao se utilizar um método na realização de uma atividade, este método impõe uma particular decomposição da atividade em termos de um padrão de atividades. Este aspecto é capturado pelo modelo da figura 5.10 e pelos seguintes axiomas de consolidação:

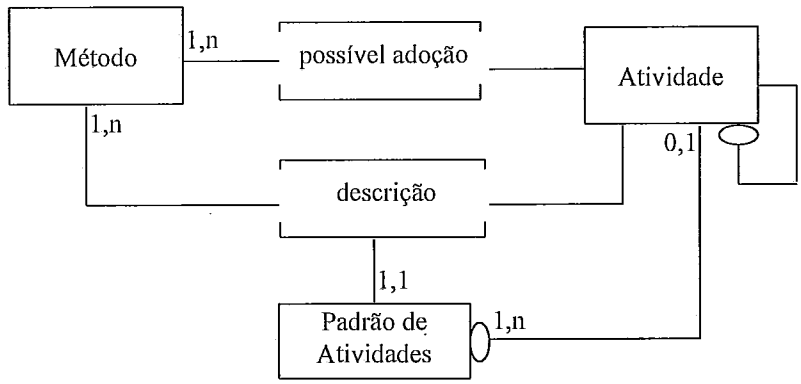


Figura 5.10 - Descrição de Padrões de Atividades por Métodos.

Se *m* é um método e *m* pode ser adotado por *a*, então *a* tem de ser uma macro-atividade.

$$(\forall\ a, m)\ (metódo(m) \wedge possíveladoção(m,a) \rightarrow macroatividade(a)) \qquad \text{(AC10)}$$

Se t é uma técnica e t pode ser adotada por a, então a tem de ser uma atividade elementar

$$(\forall\ a,\ t)\ (técnica(t) \wedge possíveladoção(t,a) \rightarrow atividadeelementar(a)) \qquad \text{(AC11)}$$

Para formalizar o modelo da figura 5.10, foram definidos os seguintes predicados: *padrãoat(pa)*, denotando que *pa* é um padrão de atividades; *padrão-composição(a,pa)*, denotando que a atividade *a* compõe o padrão de atividades *pa*; e *descrição(m,a,pa)*, indicando que o método *m* descreve a atividade *a* através do padrão de atividades *pa*. Ainda, os seguintes axiomas de consolidação devem ser observados:

$$(\forall\ a,m,pa)\ (descrição(m,a,pa) \rightarrow possíveladoção(m,a) \wedge padrãoat(pa)) \qquad \text{(AC12)}$$

$$(\forall\ a,a_1,pa)\ (padrão\text{-}composição(a_1,pa) \wedge descrição(m,a,pa) \rightarrow (a_1 \neq a \wedge \neg\ superatividade(a_1,a)) \qquad \text{(AC13)}$$

Outro aspecto a ser considerado é a adequação de procedimentos à tecnologia e ao paradigma a serem empregados no desenvolvimento. O método CommonKADS (BREUKER et al., 1994), por exemplo, é adequado à construção de sistemas baseados em conhecimento. O método de Booch (BOOCH, 1994), por sua vez, é apropriado ao desenvolvimento com o paradigma de objetos. A figura 5.11 mostra estas dependências.

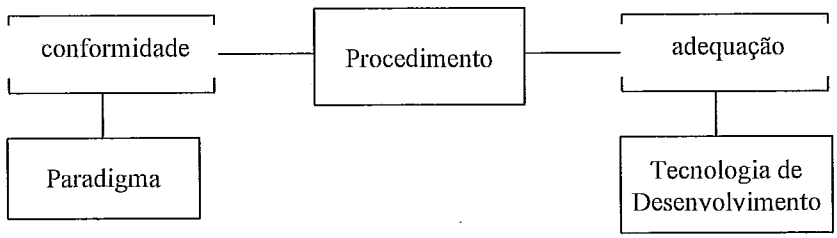


Figura 5.11 - Adequação de Procedimentos a Tecnologias de Desenvolvimento e Paradigmas.

Para representar estas relações na linguagem formal, utilizamos os predicados *proced-conformidade(p,pd)*, denotando que o procedimento *p* está em conformidade com o paradigma *pd*, e *proced-adequação(p,td)*, denotando que o procedimento *p* é adequado ao desenvolvimento utilizando a tecnologia de desenvolvimento *td*. Os seguintes axiomas de consolidação devem ser observados:

$$(\forall\, p,pd)\ (proced\text{-}conformidade(p,pd) \rightarrow procedimento(p) \wedge paradigma(pd)) \qquad (AC14)$$

$$(\forall\, p,td)\ (proced\text{-}adequação(p,td) \rightarrow procedimento(p) \wedge tecnologia(td)) \qquad (AC15)$$

onde os predicados *paradigma(pd)* e *tecnologia(td)* denotam, respectivamente, que *pd* é um paradigma e *td* é uma tecnologia de desenvolvimento.

A figura 5.12 mostra o modelo completo da ontologia de procedimento e a tabela 5.2 o Dicionário de Termos correspondente.

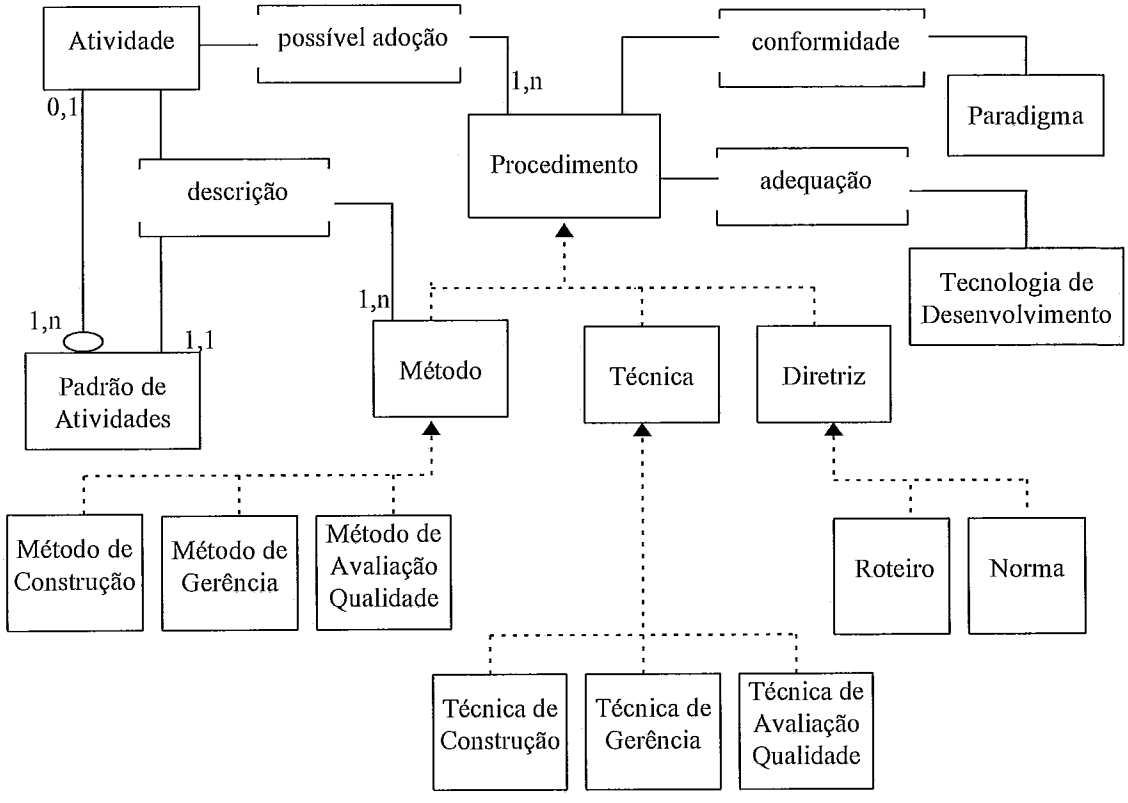


Figura 5.12 - Ontologia de Procedimento.

Tabela 5.2 - Dicionário de Termos da Ontologia de Procedimento.

Adequação	relação entre um **procedimento** e uma **tecnologia de desenvolvimento**, indicando que o **procedimento** é adequado ao desenvolvimento usando a **tecnologia de desenvolvimento**.
Conformidade	relação entre um **procedimento** e um **paradigma**, indicando que o **procedimento** segue os princípios do **paradigma**.
Diretriz	**procedimento** que visa estabelecer um padrão para a realização de **atividades**. Diretrizes podem ser divididas em **roteiros** e **normas**.
Método	**procedimento** *sistemático*, definindo passos e heurísticas para a realização de uma ou mais **atividades**. Quanto ao tipo de atividade que podem apoiar, os métodos podem ser classificados em **métodos de construção, métodos de gerência** e **métodos de avaliação da qualidade**.
Método de Avaliação da Qualidade	**método** para apoiar **atividades de avaliação da qualidade**. Ex: O método Rocha para avaliação da qualidade de produtos de software.
Método de Construção	**método** para apoiar **atividades de construção**. Ex: O método de Booch para desenvolvimento orientado a objetos.
Método de Gerência	**método** para apoiar **atividades de gerência**. Ex: Métodos de estimativa de custo.
Norma	**diretriz** que visa estabelecer padrões para a realização de **atividades** que não sejam de elaboração de documentos. Ex: normas de programação.
Paradigma	filosofia adotada na construção do software, abrangendo um conjunto de princípios e conceitos que norteiam o desenvolvimento. Ex.: paradigma estrutural e orientado a objetos.
Possível Adoção	uma relação entre um **procedimento** e uma **atividade**, indicando que o **procedimento** pode ser adotado na realização da **atividade**.
Procedimento	conduta bem estabelecida e ordenada para a realização de uma **atividade**. Quanto à sua natureza, procedimentos podem ser classificados em **métodos, técnicas** e **diretrizes**.
Roteiro	**diretriz** para a elaboração de documentos. Ex: roteiro de plano de projeto.
Técnica	**procedimento** para a realização de uma **atividade**, que não descreve como realizá-la em termos de **sub-atividades**. Técnicas, em função da natureza das **atividades** que podem apoiar, são classificadas em: **técnicas de construção, técnicas de avaliação da qualidade** e **técnicas de gerência**.
Técnica de Avaliação da Qualidade	**técnica** para avaliar a qualidade do **processo de desenvolvimento** ou dos **artefatos** nele gerados. Ex: inspeções, walkthroughs e testes.
Técnica de Construção	**técnica** para apoiar **atividades de construção**. Ex: técnicas top-down e bottom-up.
Técnica de Gerência	**técnica** para apoiar **atividades de gerência**. Ex: técnicas para estimar tempo e esforço.
Tecnologia de Desenvolvimento	tecnologia a ser empregada no desenvolvimento do software. Ex.: tecnologia convencional de processamento de dados, tecnologia de sistemas baseados em conhecimento, etc.

5.4 - Ontologia de Recurso

Propósito e Especificação de Requisitos

Ser um recurso é uma característica derivada do papel que um elemento do universo de discurso desempenha em uma atividade. Assim, as propriedades dos recursos são determinadas pelas atividades e, conseqüentemente, há uma forte interação entre as ontologias de recurso e atividade.

Antes de discutirmos a ontologia de recursos, é importante destacar a concepção de recurso adotada neste trabalho. Uma vez que uma atividade é uma primitiva de transformação, ela transforma insumos em produtos. Em se tratando de desenvolvimento de software, insumos e produtos são artefatos, já contemplados na ontologia de atividade. Entretanto, outros elementos são necessários para a realização de uma atividade, tais como agentes humanos, equipamentos de hardware e ferramentas de software. Tais elementos são agentes que atuam ou apoiam a realização da atividade, mas não podem ser considerados "matérias-primas" para a atividade. De fato, eles apenas auxiliam o processo, mas não são incorporados ao produto de software. Tais elementos são considerados recursos para a atividade.

Mais uma vez, os cenários de utilização descritos na seção 5.2 têm uma forte influência no propósito e nos requisitos da ontologia, agora de recurso, e podemos enumerar as seguintes questões de competência:

1. Que recursos são requeridos por uma atividade?
2. Qual a natureza de um determinado recurso?
3. Que ferramentas de software podem ser utilizadas para (semi-)automatizar um procedimento?

Captura e Formalização da Ontologia

Analisando as questões de competência anteriormente relacionadas, identificamos os seguintes aspectos relevantes a serem abordados na ontologia de recurso:

- taxonomia de recursos (questão 2);
- recursos requeridos por atividades (questão 1);
- (semi-)automatização de procedimentos (questão 3).

Taxonomia de Recursos

No contexto de processos de desenvolvimento de software, recursos, quanto à sua natureza, podem ser classificados em: recursos de hardware, recursos humanos e recursos de software. Recursos de software podem ser desmembrados em duas categorias: ferramentas de software e sistemas de apoio. Além disso, a taxonomia de procedimentos tem um impacto direto sobre a taxonomia de recursos, no tocante às ferramentas de software. Uma vez que ferramentas de software são recursos de software utilizados para (semi-)automatizar procedimentos, em função do tipo de procedimento que podem apoiar, as ferramentas de software classificam-se em: ferramentas de construção, ferramentas de gerência, ferramentas de avaliação da qualidade e ferramentas de propósito geral. A figura 5.13 mostra a taxonomia de recursos.

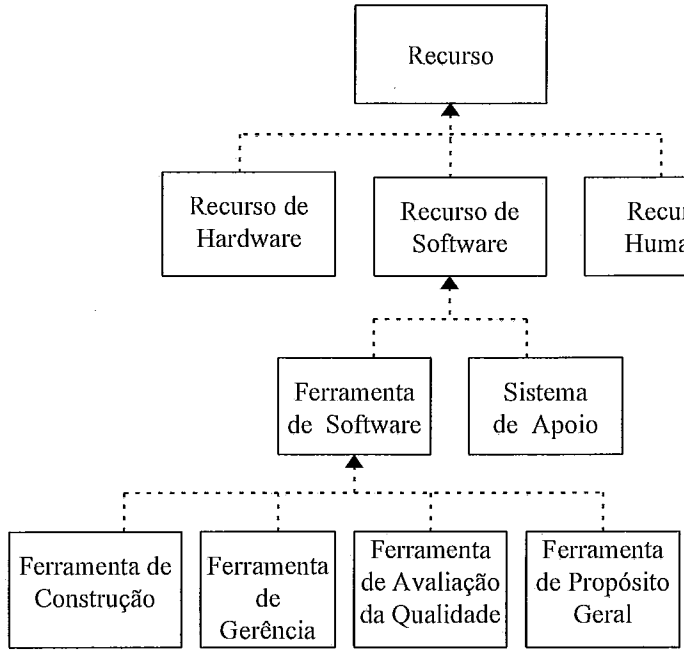


Figura 5.13 - Taxonomia de Recursos.

Recursos de hardware são quaisquer máquinas ou equipamentos necessários para a realização de uma atividade, tais como computadores, impressoras, kits multimídia, máquinas fotográficas, ou qualquer outro tipo de equipamento a ser utilizado em uma atividade.

Recursos humanos são quaisquer tipos humanos necessários para a realização de uma atividade, tais como engenheiros de software, engenheiros de conhecimento, projetistas, gerentes de projeto, programadores e usuários, entre outros.

Recursos de software são quaisquer produtos de software utilizados para apoiar a realização de uma atividade. Basicamente, dividem-se em *ferramentas de software* e *sistemas de apoio*. Ferramentas de software são recursos de software utilizados para (semi-)automatizar um procedimento. Quanto ao tipo de procedimento que (semi-) automatizam, ferramentas de software são classificadas em: ferramentas de construção, de gerência, de avaliação da qualidade e de propósito geral. Sistemas de apoio, por sua vez, são recursos de software que não automatizam um procedimento, mas ainda assim são necessários para a realização da atividade, tal como um sistema de gerenciamento de redes.

Como o próprio nome induz, *ferramentas de construção* são ferramentas que apoiam procedimentos de construção, tais como ferramentas CASE, linguagens de programação, ferramentas de depuração e sistemas gerenciadores de bancos de dados, entre outros.

Ferramentas de gerência são aquelas utilizadas para (semi-)automatizar procedimentos de gerência, tais como assistentes para elaboração de planos de projeto, sistemas de controle de versões, etc.

Ferramentas de avaliação da qualidade buscam apoiar procedimentos de avaliação da qualidade, tais como ferramentas para coleta automática de métricas, geradores de massas de teste, geradores de casos de teste, entre outras.

Outras ferramentas que não são construídas com o intuito de apoiar um tipo específico de procedimento, mas, ao contrário, podem ser utilizadas em diversas situações, são ditas *ferramentas de propósito geral.* Nesta categoria enquadram-se editores de texto, editores de formulário, editores de figuras, etc.

Para formalizar o conceito de recurso e seus diferentes tipos, os seguintes predicados foram definidos: *recurso(r)*, denotando que *r* é um recurso; *rechumano(r)*, denotando que *r* é um recurso humano; *rechardware(r)*, denotando que *r* é um recurso de hardware; *recsoftware(r)*, denotando que *r* é um recurso de software; *ferramenta(f)*, denotando que *f* é uma ferramenta de software; *sistemaapoio(s)*, denotando que *s* é um

sistema de apoio; *ferconstrução(f)*, denotando que *f* é uma ferramenta de construção; *fergerência(f)*, denotando que *f* é uma ferramenta de gerência; *feravqualidade(f)*, denotando que *f* é uma ferramenta de avaliação da qualidade; e *ferpropgeral(f)*, denotando que *f* é uma ferramenta de propósito geral.

Recursos Requeridos por Atividades

Uma atividade do desenvolvimento de software é uma primitiva de transformação que transforma artefatos de entrada em artefatos de saída. Entretanto, para que uma atividade possa ser realizada, outros elementos, além dos artefatos, são necessários. Estes elementos, ditos recursos, não são matérias-primas para a atividade, mas são igualmente importantes para a sua realização. A figura 5.14 modela esta situação.

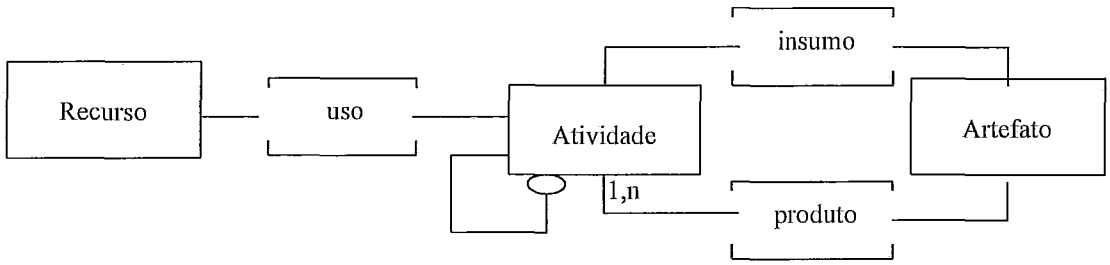


Figura 5.14 - Recursos Requeridos por Atividades.

Definimos o predicado *uso(r,a)*, denotando que o recurso *r* é necessário para a realização da atividade *a*. Para fixar os tipos dos argumentos deste predicado, o seguinte axioma de consolidação foi definido:

$$(\forall\, a,r)\ (uso(r,a) \rightarrow recurso(r) \wedge atividade(a)) \qquad \text{(AC16)}$$

O seguinte axioma estabelece a dependência entre os recursos de uma sub-atividade e de sua super-atividade:

Se um recurso *r* é requerido por uma atividade a_1 e a_1 é uma sub-atividade de *a*, então *r* é requerido por *a*.

$$(\forall\, a_1,a,r)\ ((uso(r,a_1) \wedge subatividade(a_1,a)) \rightarrow uso(r,a)) \qquad \text{(A12)}$$

(Semi-)Automatização de Procedimentos

Tendo em vista que desejamos não apenas definir um processo, mas também as ferramentas capazes de (semi-)automatizá-lo, é necessário considerar que ferramentas podem ser utilizadas para (semi-)automatizar um procedimento, como mostra a figura 5.15.

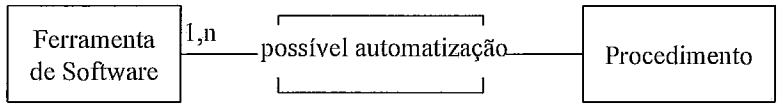


Figura 5.15 - Ferramentas Adequadas para a (Semi-)Automatização de Procedimentos.

O predicado *possívelautomatização(f,p)* foi definido para denotar que a ferramenta *f* pode ser usada para (semi-)automatizar o procedimento *p*, sendo que o seguinte axioma de consolidação tem de ser satisfeito:

$$(\forall f,p)\ (possívelautomatização(f,p) \rightarrow ferramenta(f) \wedge procedimento(p)) \qquad \text{(AC17)}$$

Além disso, as seguintes restrições devem ser observadas:

Se *p* é um método ou técnica de construção e *f* pode ser usada para (semi-) automatizar *p*, então *f* tem de ser uma ferramenta de construção ou de propósito geral.

$$(\forall f, p)\ ((metconstrução(p) \vee tecconstrução(p)) \wedge possívelautomatização(f,p) \rightarrow (ferconstrução(f) \vee ferpropgeral(f))) \qquad \text{(AC18)}$$

Se *p* é um método ou técnica de gerência e *f* pode ser usada para (semi-) automatizar *p*, então *f* tem de ser uma ferramenta de gerência ou de propósito geral.

$$(\forall f, p)\ ((metgerência\ (p) \vee tecgerência(p)) \wedge possívelautomatização(f,p) \rightarrow (fergerência(f) \vee ferpropgeral(f))) \quad \text{(AC19)}$$

Se p é um método ou técnica de avaliação da qualidade e f pode ser usada para (semi-) automatizar p, então f tem de ser uma ferramenta de avaliação da qualidade ou de propósito geral.

$$(\forall f, p)\ ((metavqualidade(p) \vee tecavqualidade(p)) \wedge possívelautomatização(f,p) \rightarrow (feravqualidade(f) \vee ferpropgeral(f))) \quad \text{(AC20)}$$

Uma vez tratada a (semi-)automatização de procedimentos, é possível tratar de forma mais adequada a questão de uma ferramenta utilizada como recurso por uma atividade. De fato, esta relação fica melhor caracterizada pela definição que se segue:

Uma ferramenta de software f só pode ser requerida por uma atividade a, se for capaz de (semi-)automatizar um procedimento p passível de adoção por a.

$$(\forall a, f)\ (uso(f,a) \rightarrow (\exists p)\ (possíveladoção(p,a) \wedge possívelautomatização(f,p))) \quad \text{(AC21)}$$

A figura 5.16 mostra o modelo completo da ontologia de recurso, sendo seus termos descritos no Dicionário de Termos da tabela 5.3.

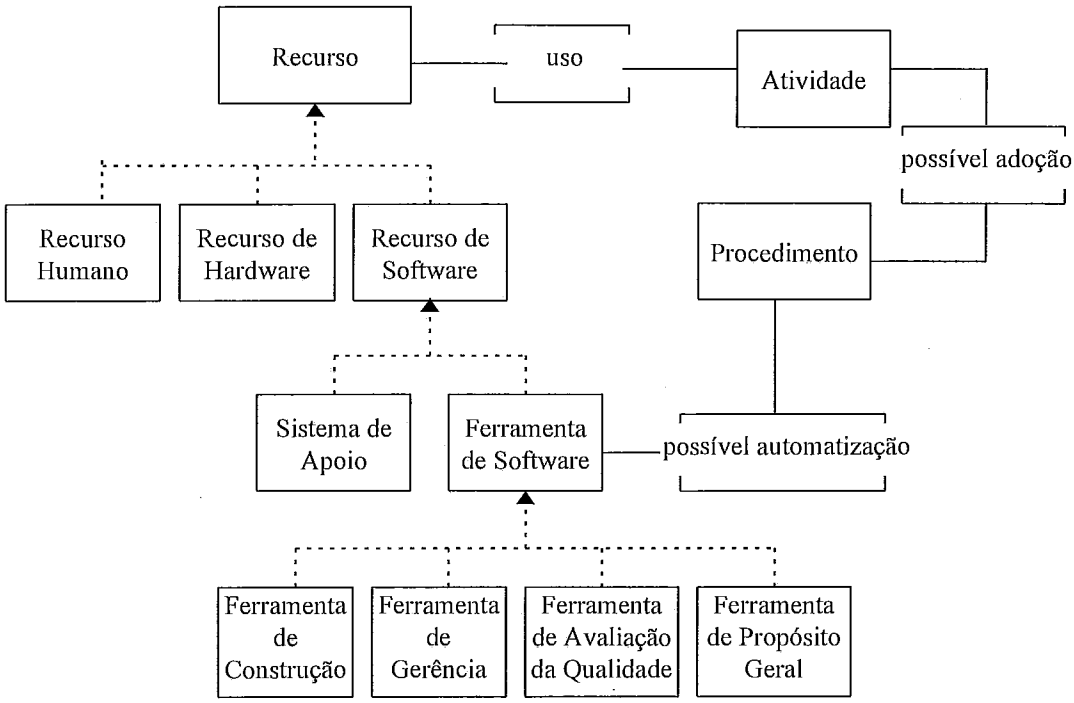


Figura 5.16 - Ontologia de Recursos.

Tabela 5.3 - Dicionário de Termos da Ontologia de Recursos.

Ferramenta de Software	**recurso de software** utilizado para (semi-)automatizar um **procedimento** adotado na realização de uma **atividade**. Quanto ao tipo de **procedimento** que podem (semi-)automatizar, **ferramentas de software** podem ser classificadas em **ferramentas de construção, ferramentas de gerência, ferramentas de avaliação da qualidade** e **ferramentas de propósito geral**.
Ferramenta de Avaliação da Qualidade	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de avaliação da qualidade**. Ex: ferramentas para coleta automática de dados de métricas, geradores de massas de teste, geradores de casos de teste, etc.
Ferramenta de Construção	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de construção**. Ex: ferramentas CASE, linguagens de programação, SGBDs,...
Ferramenta de Gerência	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de gerência**. Ex: sistema de controle de versões, etc.
Ferramenta de Propósito Geral	**ferramenta de software** que não apoia um tipo específico de **procedimento**, podendo ser utilizada para (semi-)automatizar diversos tipos de procedimentos. Ex: editor de texto, editor de figura, editor de formulário, etc.
Possível Automatização	uma relação entre uma **ferramenta de software** e um **procedimento**, indicando que a **ferramenta de software** pode ser usada para (semi-) automatizar o **procedimento**.
Recurso	qualquer coisa que seja necessária para a realização de uma **atividade**, mas que não seja um **insumo** para a **atividade**, no sentido de não ser objeto de transformação por parte da **atividade**. Recursos podem ser classificados em **recursos de hardware, recursos de software** e **recursos humanos**.
Recurso de Hardware	equipamento de hardware necessário para a realização de uma **atividade**. Ex: computadores, kit multimídia, ...
Recurso de Software	software necessário para a realização de uma **atividade**, mas que não é incorporado ao produto desta.
Recurso Humano	agente humano necessário para a realização de uma **atividade**. Ex: engenheiro de software, programador, especialista de domínio, etc.
Sistema de Apoio	recurso de software requerido por uma **atividade**, mas que não (semi-) automatiza um **procedimento**. Ex: sistema de gerenciamento de redes.
Uso	uma relação entre um **recurso** e uma **atividade**, indicando que o **recurso** é necessário para a realização da **atividade**.

5.5 - Ontologia de Processo de Software

Especificação de Requisitos Adicionais

Uma vez desenvolvidas as ontologias centrais capazes de suportar a construção da ontologia de processo, podemos nos concentrar nos aspectos relevantes dos processos de software que ainda não foram contemplados pelas demais ontologias. Assim, as seguintes questões de competência precisam ser (re)consideradas:

1. Que atividades devem ser consideradas em um processo?
2. Que procedimentos podem ser adotados na realização de atividades no contexto de um processo?

Captura da Ontologia

As questões de competência anteriormente enumeradas apontam os seguintes aspectos, relevantes para a ontologia de processo de software:

- Modelos de ciclo de vida (questão 1);
- Definição de um processo de software (questão 1);
- Adequação à tecnologia de desenvolvimento e ao paradigma adotados no desenvolvimento (questão 2).

Modelos de Ciclo de Vida

O ciclo de vida de um software inicia quando um produto de software é solicitado e termina quando este não está mais disponível. Desta forma, o ciclo de vida contém todo o conjunto das atividades de desenvolvimento, operação e manutenção (ESA, 1991). Um *modelo de ciclo de vida* estrutura atividades de um projeto em fases e define uma abordagem para organizar estas fases.

Avaliando as principais abordagens para estruturação de modelos de ciclo de vida encontradas na literatura ((DAVIS et al., 1988) (BERSOFF et al., 1991) (ESA, 1991) (PRESSMAN, 1997)), entre elas as abordagens seqüencial linear (ou em cascata) e evolutiva, podemos observar duas formas básicas de se estruturar conjuntos de fases: seqüência e iteração. Na estruturação seqüencial, as fases são realizadas apenas uma vez, sendo permitido um retorno apenas à fase anterior para correção de possíveis falhas detectadas. Na estruturação iterativa, um conjunto de fases é realizado várias vezes, segundo algum critério estabelecido.

A abordagem em cascata, por exemplo, pode ser descrita como uma única combinação seqüencial de todas as fases. A abordagem evolutiva, por sua vez, pode ser vista como uma única combinação iterativa de todas as fases do ciclo de vida. Estes dois modelos retratam os dois extremos do conjunto de possíveis abordagens para modelos de ciclo de vida: o primeiro é uma combinação puramente seqüencial, enquanto o último é uma combinação puramente iterativa. Tratando os modelos de

ciclo de vida como combinações híbridas de fases, tendo partes seqüenciais e partes iterativas, é possível mapear outros modelos de ciclo de vida.

Desta forma, um modelo de ciclo de vida define um conjunto de macro-atividades (ou fases) que um processo de desenvolvimento deve apresentar e a ordem em que elas devem ser realizadas, na forma de combinações. Combinações, assim como modelos de ciclo de vida, possuem uma importante propriedade que define a natureza da ordenação de suas fases: a estrutura, que pode ser seqüencial ou iterativa. O diagrama da figura 5.17 mostra esta concepção para a estrutura dos modelos de ciclo de vida.

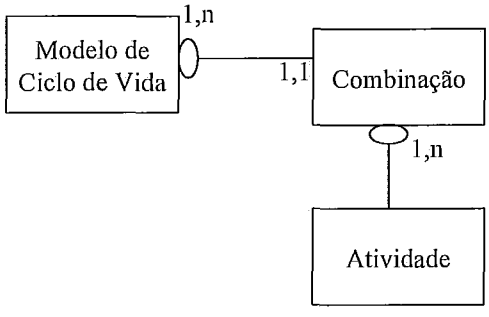


Figura 5.17 - Estrutura dos Modelos de Ciclo de Vida.

Para formalizar esta concepção de estruturação de modelos de ciclo de vida, introduzimos os seguintes predicados: *mciclovida(mcv,e)*, denotando que *mcv* é um modelo de ciclo de vida, cuja forma de estruturação básica é *e*; *combinação(c,e)*, denotando que *c* é uma combinação, cuja forma de estruturação é *e*; *mcv-composição(c,mcv,n)*, indicando que *c* é a *n*-ésima combinação do modelo de ciclo de vida *mcv*; e *comb-composição(a,c,n)*, denotando que *a* é a *n*-ésima fase da combinação *c*. O terceiro argumento nos dois últimos predicados (*n*) é necessário para capturar a ordem das combinações de um modelo de ciclo de vida e a ordem das fases dentro de uma combinação, respectivamente. O segundo argumento dos dois primeiros predicados representa a estrutura e só pode assumir um dos seguintes valores: *{Seq, Iter}*, denotando as organizações seqüencial e iterativa, respectivamente. Além disso, os seguintes axiomas de consolidação tem de ser observados:

$$(\forall\ c,mcv)\ (mcv\text{-}composição(c,mcv,n) \rightarrow mciclovida(mcv,*) \wedge combinação(c,*) \wedge n \in N^{+}) . \quad \text{(AC22)}$$

$$(\forall\ a,c\ n)\ (comb\text{-}composição(a,c,n) \rightarrow macroatividade(a) \wedge combinação(c,*) \wedge n \in N^{+}) . \quad \text{(AC23)}$$

onde N^{+} representa o conjunto dos números naturais, maiores que zero ({1,2,...}).

Definição de um Processo de Software

Basicamente, um processo consiste de um conjunto estruturado de atividades e, por conseguinte, toda a infra-estrutura envolvida na realização destas (artefatos, procedimentos e recursos) como mostra a figura 5.18.

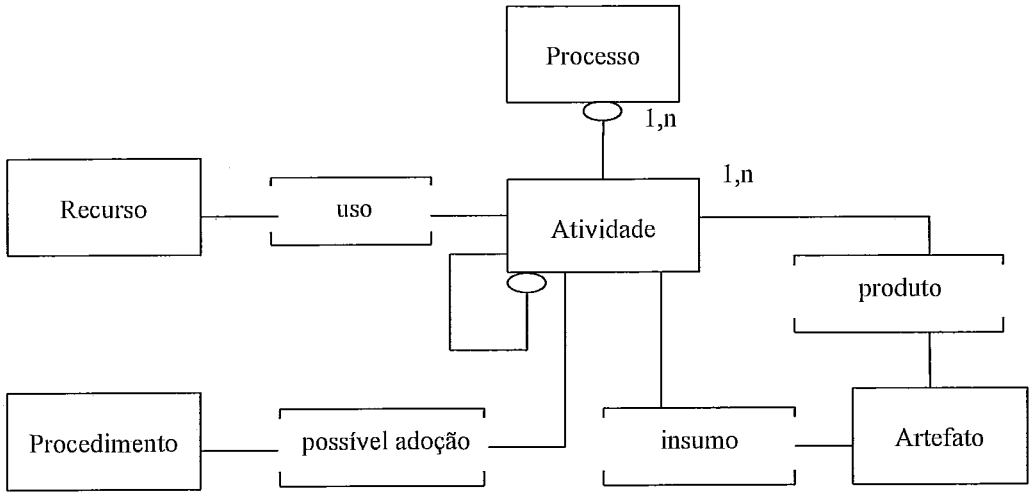


Figura 5.18 - Composição Básica de um Processo.

O predicado *proc-composição(a,pr)* indica que a atividade *a* é parte do processo *pr*, sendo que o seguinte axioma de consolidação tem de ser válido:

$$(\forall\ a,pr)\ (proc\text{-}composição(a,pr) \rightarrow processo(pr) \wedge atividade(a)) \quad \text{(AC24)}$$

onde o predicado *processo(pr)* denota que *pr* é um processo.

Uma importante recomendação da maioria dos modelos de qualidade de processo, tais como a ISO 9000-3 (1991) e o CMM (PAULK et al., 1993), é que a definição de um processo de software se dê a partir da escolha de um modelo de ciclo

de vida, isto é, as atividades de um processo devem ser definidas e ordenadas segundo a estrutura do modelo de ciclo de vida utilizado como referência. Assim, uma vez escolhido um modelo de ciclo de vida como referência para um processo, a estrutura inicial do processo corresponde ao conjunto de macro-atividades que compõem o modelo de ciclo de vida, como mostra a figura 5.19. Note que o engenheiro de software é livre para adaptar esta estrutura para o desenvolvimento em questão, podendo descartar algumas fases propostas ou adicionar outras não descritas no modelo de ciclo de vida.

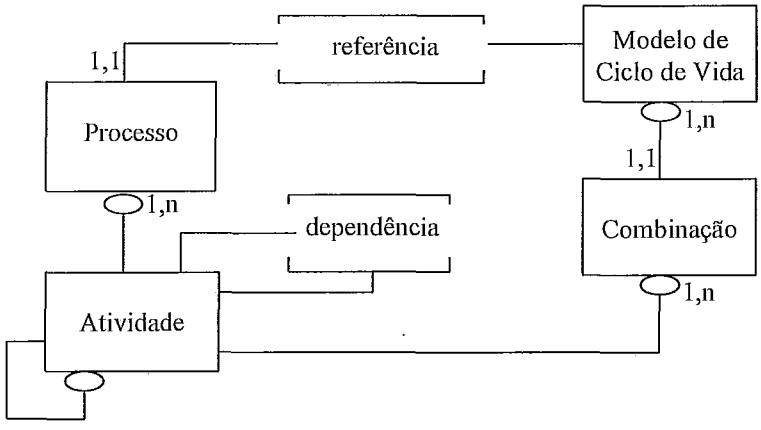


Figura 5.19 - Definição de um Processo com Base em um Modelo de Ciclo de Vida.

Adequação à Tecnologia de Desenvolvimento e ao Paradigma

Processos de software são definidos com base na tecnologia de desenvolvimento e no paradigma a serem adotados no desenvolvimento. Paralelamente, procedimentos são também fortemente dependentes destes dois fatores e, portanto, é necessário estabelecer sob que condições procedimentos podem ser utilizados dentro do contexto de um processo. A figura 5.20 mostra estas relações.

Os predicados *processo-adequação(pr,td)* e *processo-conformidade(pr,pd)* foram definidos para denotar, respectivamente, que o processo *pr* é adequado à tecnologia de desenvolvimento *td* e está em conformidade com o paradigma *pd*, sendo que os seguintes axiomas de consolidação devem ser observados:

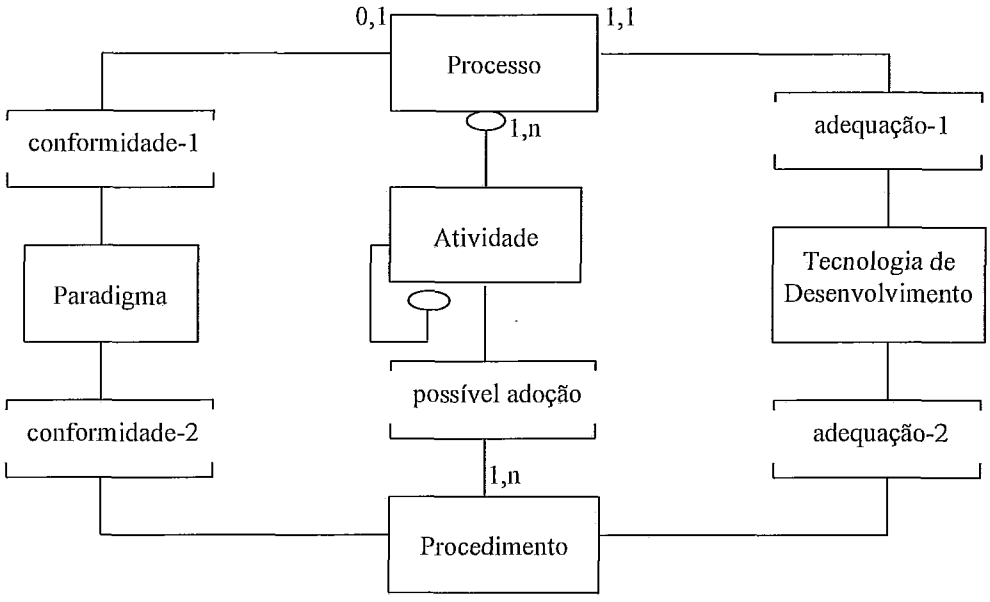


Figura 5.20 - Adequação a Tecnologias de Desenvolvimento e Paradigmas.

$$(\forall\, pr,td)\ (\ processo\text{-}adequação(pr,td) \rightarrow processo(pr) \wedge tecnologia(td)) \qquad \text{(AC25)}$$

$$(\forall\, pr,pd)\ (processo\text{-}conformidade(pr,pd) \rightarrow processo(pr) \wedge paradigma(pd)) \qquad \text{(AC26)}$$

Uma vez estabelecidas as relações entre processo, tecnologia de desenvolvimento e paradigma, devemos restringir a adoção de procedimentos por atividades do processo. Um procedimento só pode ser adotado por uma atividade do processo, se ele for adequado à tecnologia de desenvolvimento e ao paradigma que definem o processo. O método KADS (SCHREIBER et al., 1993), por exemplo, só pode ser adotado por uma atividade, se esta for parte de um processo definido para a tecnologia de sistemas baseados em conhecimento. De maneira análoga, o método de Booch (BOOCH, 1994) só pode ser adotado dentro do contexto de um processo definido para o paradigma de objetos. Assim, os seguintes axiomas de consolidação devem ser considerados:

Se *a* é uma atividade do processo *pr*, definido com base em uma tecnologia de desenvolvimento *td*, e *p* pode ser adotado por *a*, então *p* tem de ser um procedimento adequado à tecnologia de desenvolvimento *td*.

(∀ a, p, pr,td) (possíveladoção(p,a) ∧ proc-composição(a,pr) ∧ processo-adequação(pr,td) → proced-adequação(p,td) (AC27)

Se *a* é uma atividade do processo *pr*, definido em conformidade com o paradigma *pd*, e *p* pode ser adotado por *a*, então *p* tem de estar em conformidade com *pd*.

(∀ a, p, pr,pd) (possíveladoção(p,a) ∧ proc-composição(a,pr) ∧ processo-conformidade(pr,pd) → proced-conformidade(p,pd) (AC28)

O modelo completo da ontologia de processo de software e o Dicionário de Termos correspondente são mostrados no Anexo A.

5.6 - A Instanciação da Ontologia de Processo

Uma instanciação de uma ontologia consiste de um número de declarações sobre objetos do universo de discurso, usando os conceitos e as relações definidos na ontologia (VALENTE, 1995). Estas declarações, juntamente com os axiomas da ontologia, são a base para a construção de módulos de conhecimento reutilizáveis.

Uma vez que a instanciação da ontologia de processo de software tem por objetivo gerar módulos de conhecimento para a construção de um protótipo do Servidor de Conhecimento de Processo, apenas alguns indivíduos deste amplo universo de discurso foram considerados. A tabela 5.4 mostra parte da instanciação realizada.

Tabela 5.4 - Parte da Instanciação da Ontologia de Processo de Software.

Conceito	Indivíduo do Universo de Discurso	Formalização
Tecnologia de Desenvolvimento	Sistemas Convencionais de Processamento de Dados	*tecnologia(ConvencionalPD)*
	Sistemas Baseados em Conhecimento	*tecnologia(SBC)*
Paradigma	Estrutural	*paradigma(Estrutural)*
	Orientado a Objetos	*paradigma(OO)*
Modelos de Ciclo de Vida	Seqüencial Linear / Cascata	*mciclovida(Cascata,Seq)*
	Incremental	*mciclovida(Incremental,Seq)*
	RAD	*mciclovida(RAD,Seq)*
	Evolutivo Básico	*mciclovida(EvolutivoB,Iter)*
	Prototipagem Operacional	*mciclovida(PrototipOp,Iter)*
	Paralelo/Recursivo	*mciclovida(ParRec,Iter)*
	para SBCs	*mciclovida(MSBC,Iter)*
Atividades de Construção	Análise e Especificação de Requisitos	*atconstrução(Análise)*
	Projeto Arquitetural	*atconstrução(ProjArquitetural)*
	Projeto Detalhado	*atconstrução(ProjDetalhado)*
	Implementação	*atconstrução(Implementação)*
Atividades de Gerência	Determinação de Escopo	*atgerência(DeterminaEscopo)*
	Planejamento do Projeto	*atgerência(Planejamento)*
	Realização de Estimativas	*atgerência(Estimativas)*
	Análise de Riscos	*atgerência(AnáliseRiscos)*
	Determinação de Cronograma	*atgerência(DeterminaCronog)*
	Elaboração de Plano de Projeto	*atgerência(ElaboraPlanoProj)*
	Acompanhamento e Controle de Projeto	*atgerência(ControleProjeto)*
Atividades de Avaliação da Qualidade	Elaboração do Plano de Qualidade	*atavqualidade(ElaboraPlQual)*
	Auditoria	*atavqualidade(Auditoria)*
	Coleta de Dados de Métricas	*atavqualidade(ColetaMétrica)*
	Análise de Dados de Métricas	*atavqualidade(AnáliseMétrica)*
	Gerência de Configuração	*atavqualidade(GerConfigura)*
	Revisão Técnica	*atavqualidade(RevisãoTécnica)*
	Teste de Unidade	*atteste(TesteUnidade)*
	Teste de Integração	*atteste(TesteIntegração)*
	Teste de Validação	*atteste(TesteValidação)*
	Teste de Sistema	*atteste(TesteSistema)*

Tabela 5.4 - Parte da Instanciação da Ontologia de Processo de Software (continuação).

Artefatos	Especificação de Requisitos	*artefato(EspReq,Documento)*
	Especificação de Projeto	*artefato(EspProj,Documento)*
	Relatório de Revisão Técnica	*artefato(RelRT,Documento)*
	Plano de Gerência de Riscos	*artefato(PlGerRisc,Documento)*
	Plano de Projeto	*artefato(PlanProj,Documento)*
	Sub-programa	*artefato(SubPrograma,Código)*
	Classe	*artefato(Classe, Código)*
Métodos de Construção	Análise Estruturada	*metconstrução(AnEstr)*
	Projeto Estruturado	*metconstrução(ProjEstr)*
	Método de Coad-Yourdon	*metconstrução(CoadYourdon)*
	Método de Booch	*metconstrução(Booch)*
	OMT	*metconstrução(OMT)*
	CommonKADS	*metconstrução(CommonKADS)*
Técnicas de Construção	Reunião com questões livres de contexto	*tecconstrução(ReuniãoQLC)*
	Entrevista	*tecconstrução(Entrevista)*
	FAST	*tecconstrução(FAST)*
	JAD	*tecconstrução(JAD)*
	QFD	*tecconstrução(QFD)*
	Protocolo por Telefone	*tecconstrução(ProtocoloFone)*
	Prototipagem Descartável	*tecconstrução(ProtDescartável)*
Métodos de Gerência	COCOMO	*metgerência(COCOMO)*
	Estimativa para OO	*metgerência(EstimaOO)*
Técnicas de Gerência	Estimativas de LOCs	*tecgerência(EstimaLOC)*
	Estimativas de FPs	*tecgerência(EstimaFP)*
	Modelos Empíricos de Estimativa	*tecgerência(ModEmpíricoEst)*
	Equação de Software	*tecgerência(EqSoft)*
	Checklist de Itens de Risco	*tecgerência(ChecklistRisco)*
	Tabela de Riscos	*tecgerência(TabelaRiscos)*
	Rede de Tarefas	*tecgerência(RedeTarefas)*
	PERT/CPM	*tecgerência(PERTCPM)*
	Gráfico de Gantt	*tecgerência(GráficoGantt)*
Métodos de Avaliação da Qualidade	Método Rocha	*metavqualidade(Rocha)*

Tabela 5.4 - Parte da Instanciação da Ontologia de Processo de Software (continuação).

Técnicas de Avaliação da Qualidade	Inspeção	*teccertificação(Inspeção)*
	Walkthrough	*teccertificação(Walkthrough)*
	Teste de Caixa Branca	*tecteste(TesteCaixaBranca)*
	Teste de Caixa Preta	*tecteste(TesteCaixaPreta)*
	Teste de Caminho Básico	*tecteste(TesteCaminhoB)*
	Teste de Condição	*tecteste(TesteCondição)*
	Teste de Fluxo de Dados	*tecteste(TesteFluxoDados)*
	Teste Baseado em Grafo	*tecteste(TesteBaseGrafo)*
	Análise de Valor Limite	*tecteste(AnValorLimite)*
	Teste de GUI	*tecteste(TesteGUI)*
	Teste de Facilidades de Ajuda	*tecteste(TesteHelp)*
	Teste de Particionamento	*tecteste(TesteParticiona)*
	Teste Baseado em Cenário	*tecteste(TesteBaseCenário)*
	Teste Derivado de Modelo Comportamental	*tecteste(TesteModComp)*
Roteiros	Plano de Gerência de Riscos	*roteiro(RotPlanoGerRiscos)*
	Plano de Projeto	*roteiro(RotPlanoProjeto)*
	Especificação de Requisitos	*roteiro(RotEspecRequisitos)*
	Especificação de Projeto	*roteiro(RotEspecProjeto)*
	Plano de Qualidade	*roteiro(RotPlanoQualidade)*
	Relatório de RTF	*roteiro(RotRelRTF)*
Recursos Humanos	Gerente de Projeto	*rechumano(GerenteProjeto)*
	Engenheiro de Software	*rechumano(EngSoftware)*
	Projetista de Software	*rechumano(ProjSoftware)*
	Programador	*rechumano(Programador)*
	Cliente	*rechumano(Cliente)*
	Usuário Final	*rechumano(UsuárioFinal)*
Ferramentas de Propósito Geral	Editor de Texto	*ferpropgeral(EditorTexto)*
	Editor de Formulário	*ferpropgeral(EditorFormulário)*
	Editor de Figura	*ferpropgeral(EditorFigura)*
Ferramentas de Construção	CASE para Análise Estruturada	*ferconstrução(CASEAnEst)*
	CASE para Método de Booch	*ferconstrução(CASEBooch)*
	CASE para OMT	*ferconstrução(CASEOMT)*
	CASE para Coad-Yourdon	*ferconstrução(CASECYourdon)*

Além da descrição de instâncias de conceitos, a instanciação da ontologia de processo envolveu também a instanciação das relações. Para ilustrar este processo, apresentamos a seguir algumas destas instanciações.

Definição de Modelos de Ciclo de Vida

Para ilustrar a instanciação de modelos de ciclo de vida usando a ontologia proposta, tomemos dois exemplos.

- *Modelo Seqüencial Linear / Cascata*: Este modelo sugere uma abordagem puramente seqüencial para o desenvolvimento de software, como mostra a figura 5.21, e portanto, pode ser definido como uma única combinação seqüencial de todas as fases, como mostra a figura 5.22.

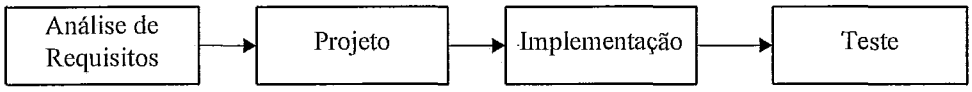


Figura 5.21 - Modelo Seqüencial Linear / Cascata.

mciclovida(Cascata,Seq)
combinação(CCascata, Seq)
mcv-composição(CCascata, Cascata, 1)
comb-composição(Análise, CCascata, 1)
comb-composição(Projeto, CCascata, 2)
comb-composição(Implementação, CCascata, 3)
comb-composição(Teste, CCascata, 4)

Figura 5.22 - Formalização da Instanciação do Modelo em Cascata.

- *Modelo Incremental*: é uma variação do modelo seqüencial linear na qual as fases de análise de requisitos e projeto da arquitetura são realizadas para o software como um todo. Uma vez definida a arquitetura do software, as demais fases (projeto detalhado, implementação e teste) são divididas em unidades mais gerenciáveis e, assim sendo, o software é distribuído em várias versões, cada uma delas com funcionalidade e capacidade aumentadas, como mostra a figura 5.23. Assim, é definido por duas combinações, uma seqüencial e outra iterativa, como mostra a figura 5.24.

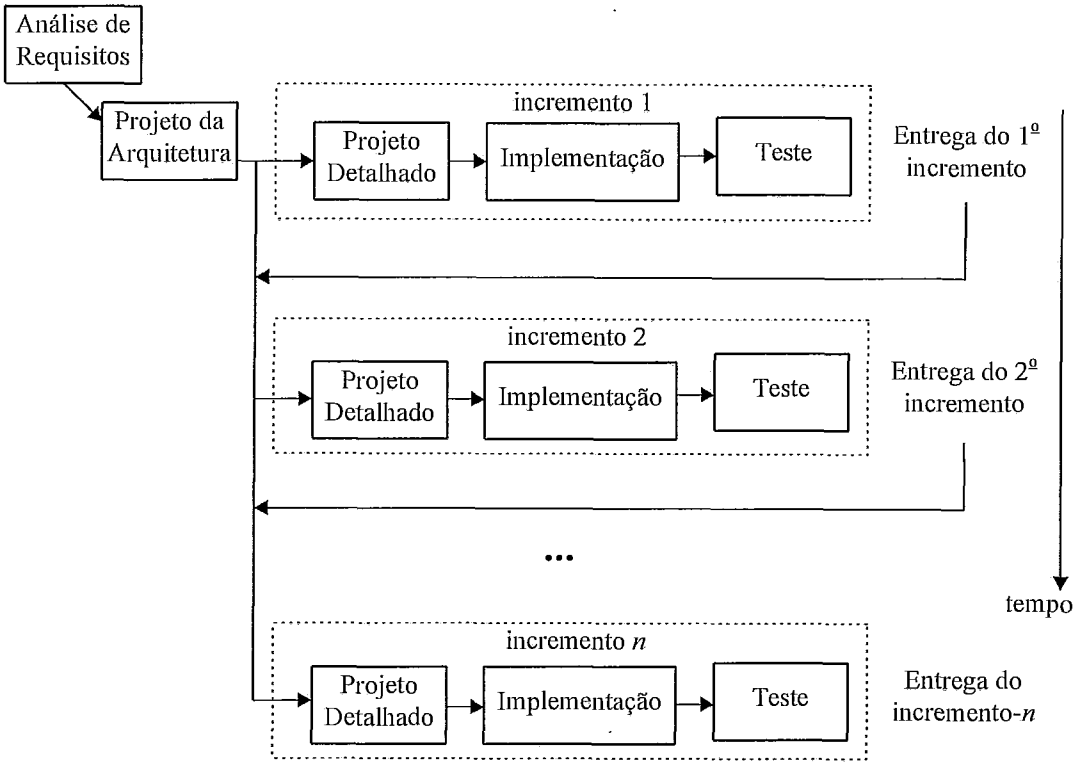


Figura 5.23 - Modelo Incremental.

mciclovida(Incremental,Seq)
combinação(CInc1, Seq)
combinação(CInc2, Iter)
mcv-composição(CInc1, Incremental, 1)
mcv-composição(CInc2, Incremental, 2)
comb-composição(Análise, CInc1, 1)
comb-composição(ProjetoArquitetural, CInc1, 2)
comb-composição(ProjetoDetalhado, CInc2, 1)
comb-composição(Implementação, CInc2, 2)
comb-composição(Teste, CInc2, 3)

Figura 5.24 - Formalização da Instanciação do Modelo Incremental.

Métodos e Padrões de Atividade

Conforme mencionado na seção 5.4, métodos podem ser adotados na realização de macro-atividades, impondo-lhes uma particular decomposição em termos de um padrão de atividades (figura 5.10). Como um exemplo de instanciação deste modelo, tomemos o método de Coad-Yourdon (COAD et al., 1992) (COAD et al., 1993).

- *Método de Coad-Yourdon*: é um método adequado à tecnologia de Sistemas Convencionais de Processamento de Dados e foi definido em conformidade com o paradigma Orientado a Objetos. Este método pode ser adotado por atividades de Análise de Requisitos e Projeto, sendo que a primeira é decomposta nas seguintes sub-atividades, como mostra a formalização na figura 5.25:

  Análise: (i) Localização de Classes e Objetos,
  (ii) Identificação de Estruturas,
  (iii) Identificação de Assuntos,
  (iv) Definição de Atributos e
  (v) Definição de Serviços

proced-adequação(CoadYourdon,ConvencionalPD)
proced-conformidade(CoadYourdon,OO)
possíveladoção(CoadYourdon, Análise)
descrição(CoadYourdon,Análise,POOACY)
padrãoat(POOACY)
padrão-composição(LocalizaçãoClassesObjetos, POOACY, 1)
padrão-composição(IdEstruturas, POOACY, 2)
padrão-composição(IdAssuntos, POOACY, 3)
padrão-composição(DefAtributos, POOACY, 4)
padrão-composição(DefServiços, POOACY, 5)

Figura 5.25 - Instanciação do Método de Coad-Yourdon e um de seus Padrões de Atividade.

Atividades, Procedimentos e Recursos

Para ilustrar a instanciação das relações envolvendo atividades, artefatos, procedimentos e recursos, tomemos o seguinte exemplo, formalizado na figura 5.26:

- *Análise de Riscos*: A atividade de análise de riscos pode ser decomposta nas seguintes sub-atividades:

  (i) *identificar riscos*: esta atividade pode ser apoiada pela técnica de Checklist de Itens de Risco e produz uma Lista de Itens de Risco. Requer a presença do gerente de projeto, engenheiro de software e cliente;

  (ii) *projetar riscos*: esta atividade utiliza a Lista de Itens de Risco para gerar uma Tabela Risco-Probabilidade-Impacto, podendo ser apoiada pela técnica de Tabela de Riscos. Requer a presença do gerente de projeto, engenheiro de software e cliente;

  (iii) *documentar riscos*: a partir da Lista de Itens de Risco e da Tabela Risco-Probabilidade-Impacto, deve-se gerar o Plano de Gerência, Monitoração e Abrandamento de Riscos. Pode adotar o roteiro Plano de Gerência de Riscos, sendo realizada pelo engenheiro de software;

  (iv) *abrandar, monitorar e gerenciar riscos*: com base no Plano de Gerência, Monitoração e Abrandamento de Riscos, o gerente de projeto deve monitorar e gerenciar os riscos identificados, atualizando sempre a documentação.

subatividade(IdentificarRiscos, AnáliseRiscos)
subatividade(ProjetarRiscos, AnáliseRiscos)
subatividade(DocumentarRiscos, AnáliseRiscos)
subatividade(MonitorarGerenciarRiscos, AnáliseRiscos)
possíveladoção(ChecklistRisco, IdentificarRiscos)
uso(GerenteProjeto,IdentificarRiscos)
uso(EngSoftware,IdentificarRiscos)
uso(Cliente,IdentificarRiscos)
produto(ListaItensRisco, IdentificarRiscos)
possíveladoção(TabelaRiscos, ProjetarRiscos)
uso(GerenteProjeto,ProjetarRiscos)
uso(EngSoftware,ProjetarRiscos)
uso(Cliente,ProjetarRiscos)
insumo(ListaItensRisco, ProjetarRiscos)
produto(TabRiscos, ProjetarRiscos)
possíveladoção(RotPlanoGerRiscos, DocumentarRiscos)
uso(EngSoftware,DocumentarRiscos)
insumo(ListaItensRisco, DocumentarRiscos)
insumo(TabRiscos, DocumentarRiscos)
produto(PlanoGerMonAbranRiscos, DocumentarRiscos)
uso(GerenteProjeto,MonitorarGerenciarRiscos)
insumo(PlanoGerMonAbranRiscos, MonitorarGerenciarRiscos)
produto(PlanoGerMonAbranRiscosAtualizado,MonitorarGerenciarRiscos)

Figura 5.26 - Instanciação da Atividade de Análise de Riscos.

5.7 - Conclusões do Capítulo

Neste capítulo, apresentamos uma ontologia de processo de software construída utilizando o método para Engenharia de Ontologias proposto na seção 3.4.2. A figura 3.12 mostra o processo de desenvolvimento de uma ontologia e destaca a forte interação entre as fases de captura e formalização. Na prática, esta relação foi nitidamente percebida. De fato, quando alguma parte do modelo elaborado na fase de captura não é facilmente formalizada, isto constitui uma forte indicação de que o domínio não foi adequadamente modelado e deve ser revisto. Assim, quando o processo de formalização não transcorrer de forma natural, é necessário um retorno à fase de captura.

A definição dos axiomas é uma das atividades que apresenta maior complexidade no desenvolvimento de uma ontologia. Conforme discutido na fase de avaliação da ontologia, não é desejável escrever mais axiomas do que o necessário para caracterizar as soluções das questões de competência. Assim, deve-se avaliar uma ontologia para evitar redundâncias no conjunto de axiomas e, portanto, a etapa de avaliação deve transcorrer em paralelo com as fases de captura e formalização.

As ontologias desenvolvidas têm por objetivo guiar a aquisição de conhecimento no desenvolvimento de sistemas baseados no conhecimento sobre processos de software e, portanto, foram incorporadas à base de conhecimento do Servidor de Conhecimento de Processo, descrito no capítulo 6. Desta forma, este conhecimento pode ser compartilhado por vários sistemas. Ainda no capítulo 6, a abordagem de Engenharia de Conhecimento baseada em Reutilização, apresentada na seção 3.4.3, foi utilizada no desenvolvimento de uma ferramenta baseada em conhecimento para auxiliar engenheiros de software na definição de processos e instanciação de ADSs. Esta abordagem pressupõe o uso de ontologias e, assim, mostramos como utilizar estes componentes na construção de uma aplicação específica.

Capítulo 6

O Servidor de Conhecimento de Processo

No capítulo 5, apresentamos uma ontologia de processo de software que guiou a construção dos componentes de conhecimento do universo de discurso - processos de desenvolvimento de software. Estes componentes são a base para a construção de um Servidor de Conhecimento de Processo, conforme discutido no capítulo 4.

Neste capítulo, descrevemos a construção de um protótipo do Servidor de Conhecimento de Processo. Uma vez que um Servidor de Conhecimento tem de prover, também, componentes de conhecimento de tarefa, na seção 6.1 discutimos o desenvolvimento de modelos de tarefa para o desenvolvimento de software, tomando por base a biblioteca de CommonKADS (VALENTE et al., 1994). A seção 6.2 explora a implementação de um protótipo do Servidor de Conhecimento de Processo na Estação TABA. Finalmente, para ilustrar como este Servidor de Conhecimento pode apoiar a construção de ferramentas baseadas em conhecimento de processo de software, mostramos na seção 6.3, o desenvolvimento de *Assist-Pro*, um assistente inteligente para apoiar a definição de processos de desenvolvimento de software no meta-ambiente TABA. A seção 6.4 apresenta as conclusões do capítulo.

6.1 - Modelos de Tarefa

A despeito da grande variedade de problemas em um certo universo de discurso, é geralmente possível identificar alguns problemas que são típicos ou recorrentes. Por exemplo, diagnose é um problema característico na prática médica;

em eletrônica, não apenas diagnose mas também problemas de projeto são típicos. Estes tipos de problemas são freqüentemente usados para guiar a pesquisa sobre a resolução de problemas, como é possível notar pelo grande número de abordagens baseadas em tarefa existentes para a modelagem de conhecimento, conforme discutido na seção 3.2. Assim, a discussão sobre a resolução de problemas em um universo de discurso deve levar em conta os tipos de problemas freqüentes nesse universo. Tipos de problemas são, de fato, uma importante diretriz para a formulação de modelos que podem ser efetivamente aplicados na construção de SBCs para o domínio em questão. No contexto de Servidores de Conhecimento, estes modelos são a base para a construção dos componentes de conhecimento de tarefa.

Utilizamos a biblioteca de modelos de tarefa de CommonKADS (VALENTE et al., 1994) como referência para a elaboração dos modelos de tarefa que fundamentam os componentes de conhecimento de tarefa do Servidor de Conhecimento de Processo. Basicamente, a biblioteca de CommonKADS está organizada segundo uma coleção de tipos de problemas, contendo oito grupos básicos de tipos de problemas: problemas de modelagem, planejamento, projeto, designação, previsão, avaliação, monitoramento e diagnose (BREUKER, 1994).

Os modelos de tarefa de CommonKADS são descritos em termos de estruturas de funções. Uma função é uma descrição de uma (sub-)tarefa através de sua meta e de uma relação de entrada-saída. A entrada e a saída de uma função são representadas por papéis que o conhecimento de domínio pode desempenhar na resolução de problemas. Por exemplo, um papel de entrada ou saída do tipo *hipótese* significa que algum elemento do domínio terá o papel de hipótese durante o processo de raciocínio. Além disso, funções podem referenciar estruturas de conhecimento que são usadas, mas cujo conteúdo não se altera durante a resolução do problema - os *papéis estáticos*. As únicas referências que uma função faz ao conhecimento de domínio são através dos papéis de conhecimento de entrada, de saída e estático.

CommonKADS utiliza uma linguagem gráfica para descrever modelos de tarefa (VALENTE et al., 1994). Nesta linguagem, funções são representadas por elipses, enquanto os papéis de conhecimento são representados por retângulos.

Ligações com setas indicam a direção do fluxo de conhecimento, sendo que setas duplas são usadas para indicar papéis estáticos.

Estruturas de funções são coleções de funções conectadas por papéis comuns de entrada e saída, representadas como grafos direcionados. Uma estrutura de função especifica como uma tarefa é decomposta em sub-tarefas e como essas estão conectadas, fornecendo, assim, uma visão do fluxo de conhecimento no raciocínio. Cada função em uma estrutura pode ser recursivamente expandida, criando-se uma estrutura de funções em camadas. É importante realçar que, em uma estrutura de funções, não há compromisso com controle e, portanto, ela não é um algoritmo.

De maneira geral, todos os tipos de problema identificados na coleção de CommonKADS ocorrem no desenvolvimento de software. Atividades de construção de um sistema certamente envolvem problemas de modelagem, projeto e designação; atividades de avaliação da qualidade envolvem problemas de previsão, avaliação, monitoramento e diagnose; atividades de gerência, por sua vez, envolvem problemas de planejamento, designação, previsão, monitoramento e diagnose.

Assim, a coleção de tipos de problema de CommonKADS pode ser utilizada como uma base para a modelagem de conhecimento sobre desenvolvimento de software, segundo uma perspectiva de tarefa. Oferecer suporte baseado em conhecimento a todo o processo de software envolve estudar os vários tipos de problema, procurando definir as particularidades destes tipos de problema no domínio em questão.

Com base em sua competência - a definição de processos de software e a instanciação de ADSs para apoiar os processos definidos - o Servidor de Conhecimento de Processo tem de oferecer componentes de conhecimento para, pelo menos, dois tipos de tarefas: planejamento e designação. Assim, estes dois tipos de problemas foram escolhidos para serem tratados neste capítulo.

Problemas de planejamento têm por objetivo construir um plano com base no estado inicial do mundo e no estado que se deseja atingir. O plano resultante é representado, de modo geral, como uma seqüência de ações que, quando executada partindo do estado inicial, deve conduzir ao estado meta (VALENTE, 1994).

Problemas de designação visam preencher uma estrutura com elementos, de modo que determinados requisitos e restrições sejam satisfeitos. A estrutura a ser preenchida pode ser um projeto ou um plano. No primeiro caso, os elementos a serem utilizados para preencher a estrutura de projeto são componentes e o problema é tipicamente um problema de *configuração*. No segundo caso, os elementos a serem designados às atividades do plano são geralmente recursos e o problema é então chamado de *programação* ou *escalonamento* (BREUKER, 1994).

Planejamento

São várias as ocorrências de problemas do tipo planejamento no desenvolvimento de software. Toda elaboração de um plano requerido no desenvolvimento é um problema de planejamento, tais como as elaborações dos Planos de Projeto, de Controle da Qualidade e de Acompanhamento e Controle do Projeto, entre outros. A própria definição de um processo de software é um problema de planejamento. Assim, o modelo de tarefa da biblioteca CommonKADS para este tipo de problema, mostrado na figura 6.1, foi adaptado para o contexto do desenvolvimento de software.

Os papéis de conhecimento descritos na figura 6.1 têm o seguinte significado (VALENTE, 1994):

- Estado Inicial do Mundo: descreve o mundo antes do plano ser executado.
- Estado Meta: descreve aspectos obrigatórios do estado do mundo após a execução do plano.
- Plano: é o resultado do planejamento, geralmente representado como um conjunto ordenado de ações que, partindo do estado inicial, conduz ao estado meta.
- Descrição do Mundo: uma descrição do mundo (domínio) sobre o qual se está planejando. Compreende o conhecimento necessário para representar ou descrever *estados do mundo* e as *mudanças de estado*. As mudanças de estado são, na maioria das vezes, expressas como um conjunto de ações e eventos e, neste sentido, especificam os elementos com os quais um plano é composto.

- Descrição de Plano: envolve todo o conhecimento sobre o plano a ser gerado. Compreende a *estrutura de composição do plano* e o *conhecimento para sua avaliação* (opcional). A composição do plano especifica *como* as partes de um plano (ações ou subplanos) podem ser montadas. O conhecimento de avaliação especifica o conhecimento para definir se um plano é válido, ou que fatores tornam um (sub-)plano melhor que outro.

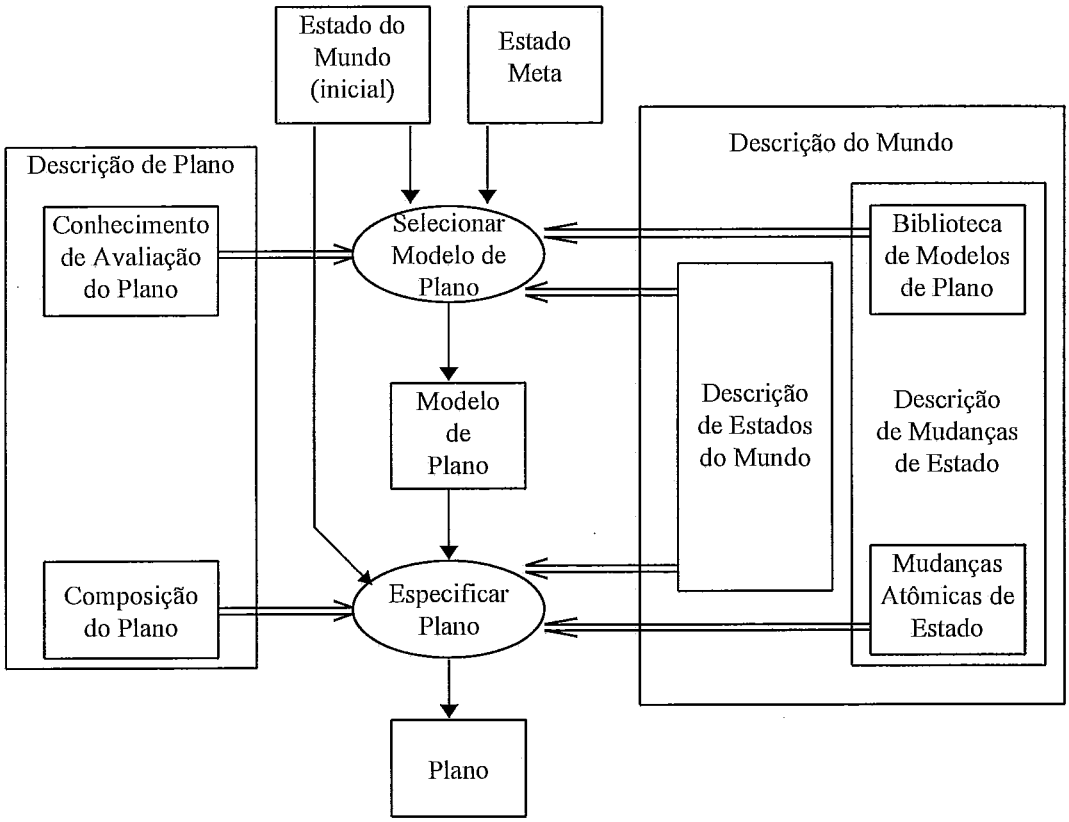


Figura 6.1 - Modelo de Tarefa para o Problema de Planejamento (VALENTE, 1994).

No contexto do desenvolvimento de software, alguns papéis de conhecimento podem ser descritos de forma mais adequada. Uma vez que a maioria dos planos no desenvolvimento de software tem um propósito bem estabelecido, não nos interessa falar em termos de estados inicial e meta do mundo. É mais apropriado pensar em características gerais que o plano deve considerar.

As mudanças de estado são descritas em termos de atividades, muitas vezes agrupadas em modelos de plano. Assim, a descrição de mudanças de estado

compreende dois papéis de conhecimento principais: conhecimento sobre modelos de plano e conhecimento de atividade.

Os estados do mundo são os estados intermediários do desenvolvimento relevantes para o plano em desenvolvimento e, portanto, um sub-conjunto do conhecimento do processo como um todo.

Finalmente, uma vez que um plano é descrito sempre em termos de atividades, a composição do plano no desenvolvimento de software especifica como as atividades podem ser combinadas e, portanto, representa um conhecimento sobre atividades.

A figura 6.2 mostra o modelo de tarefa para o planejamento adaptado ao contexto do desenvolvimento de software.

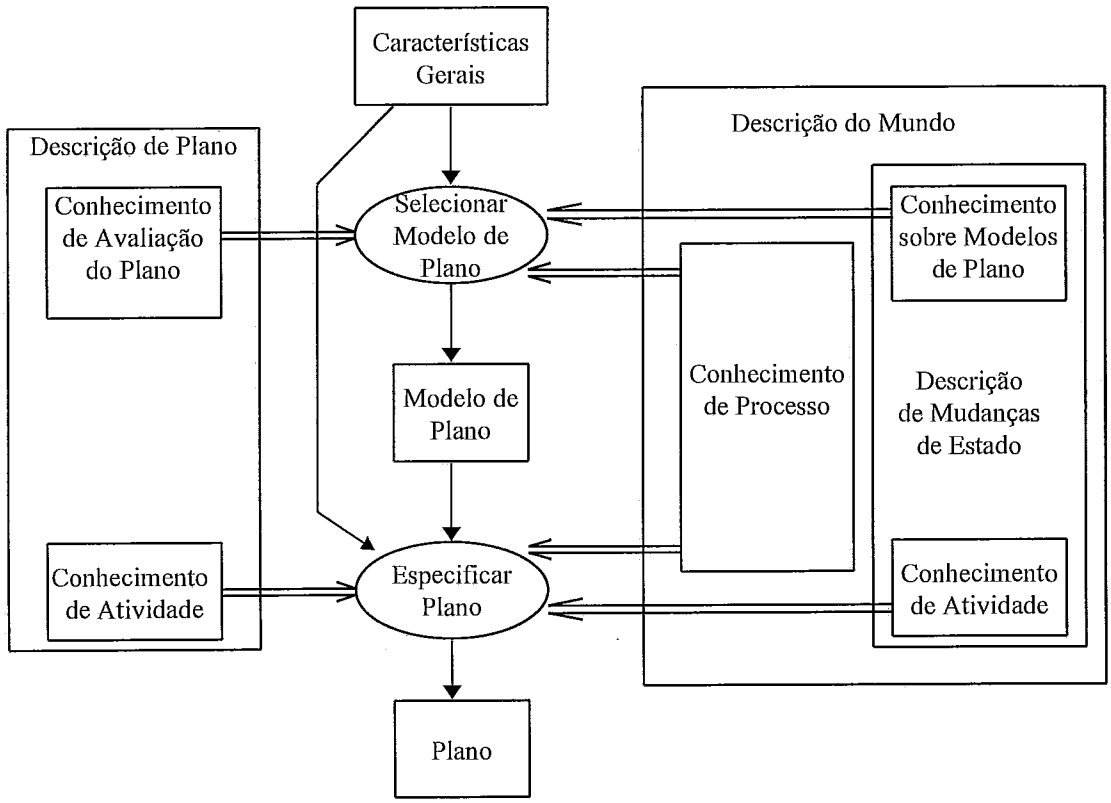


Figura 6.2 - Modelo de Tarefa de Planejamento Adaptado ao Desenvolvimento de Software.

Designação

Da mesma forma que o planejamento, são várias as ocorrências de problemas do tipo designação no desenvolvimento de software. Recursos têm de ser alocados a atividades. Procedimentos têm de ser selecionados para serem adotados na realização

de atividades. Assim, o modelo de tarefa da biblioteca CommonKADS para este tipo de problema foi adaptado para o contexto do desenvolvimento de software. A figura 6.3 mostra o modelo de tarefa original para problemas de designação, descrito na biblioteca de CommonKADS (SUNDIN, 1994), sendo que seus papéis de conhecimento têm o seguinte significado:

- Componentes: conjunto de objetos para os quais recursos devem ser designados.
- Recursos: conjunto de objetos a serem designados para componentes.
- Designação: um conjunto de tuplas $\langle c,r \rangle$, onde c é um componente e r um recurso, que satisfazem um conjunto especificado de restrições e requisitos.
- Designação Parcial: uma designação (parcial) existente, que opcionalmente pode ser dada na entrada, como por exemplo, quando uma designação precisa ser modificada.
- Restrições e Requisitos: restringem o conjunto de possíveis soluções ou agem como um critério para a seleção entre soluções possíveis.

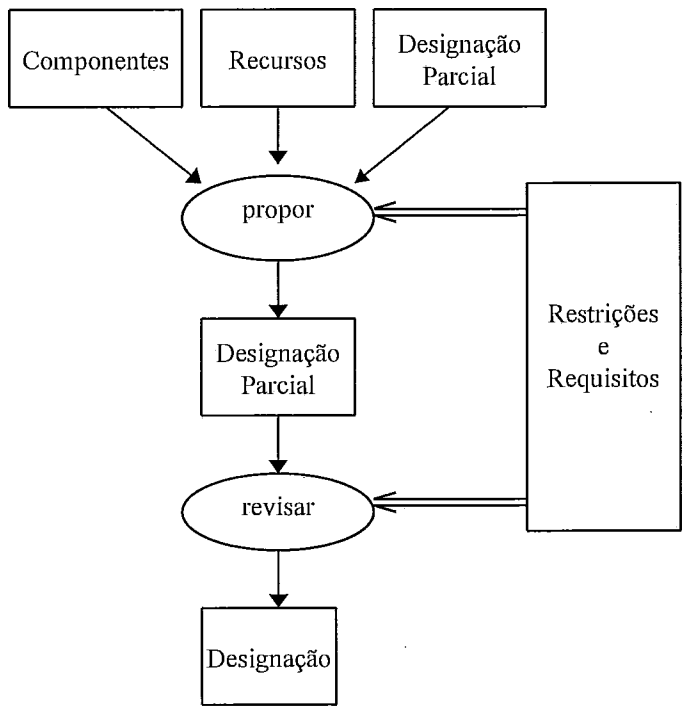


Figura 6.3 - Modelo de Tarefa para o Problema da Designação (SUNDIN, 1994).

Novamente, como fizemos para o modelo de planejamento, o modelo genérico da figura 6.3 deve ser adaptado ao contexto do desenvolvimento de software. Neste caso, as únicas mudanças referem-se aos papéis de conhecimento *Componentes* e *Recursos*. No contexto do desenvolvimento de software, *Componentes* são, de fato, *Atividades*. O papel de conhecimento *Recursos* do modelo original, por sua vez, pode representar mais do que recursos na concepção adotada neste trabalho e, portanto, são tratados por Necessidades, como mostra a figura 6.4.

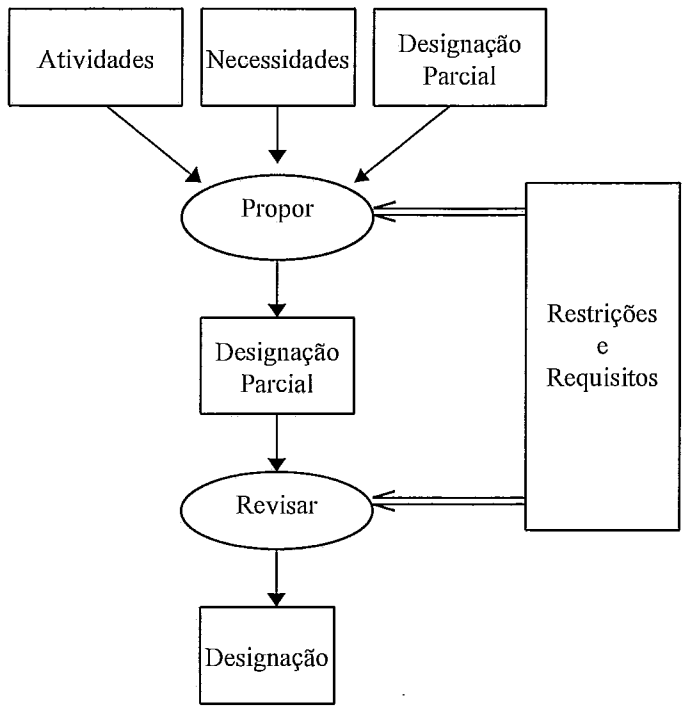


Figura 6.4 - Modelo de Tarefa de Designação Adaptado ao Desenvolvimento de Software.

6.2 - O Protótipo do Servidor de Conhecimento de Processo

Conforme discutido na seção 4.3, a arquitetura de um Servidor de Conhecimento compreende, essencialmente, uma base de conhecimento modular, construída tendo por base uma ontologia, e um conjunto de resolvedores de problemas para os tipos de problemas mais freqüentemente encontrados no universo de discurso.

A competência estabelecida para o protótipo do Servidor de Conhecimento de Processo foi a definição de processos de desenvolvimento de software. Assim, a base

de conhecimento modular foi construída com base na ontologia de processo, apresentada no capítulo 5, e os *templates* de resolvedores de problemas foram desenvolvidos a partir dos modelos de tarefa apresentados na seção anterior.

Para incorporar a arquitetura de Servidores de Conhecimento na Estação TABA, fez-se necessário alterar sua estrutura de conhecimento, discutida na seção 4.2. Assim, uma nova estrutura foi proposta, como mostra a figura 6.5. O modelo completo desta estrutura está apresentado no Apêndice B.

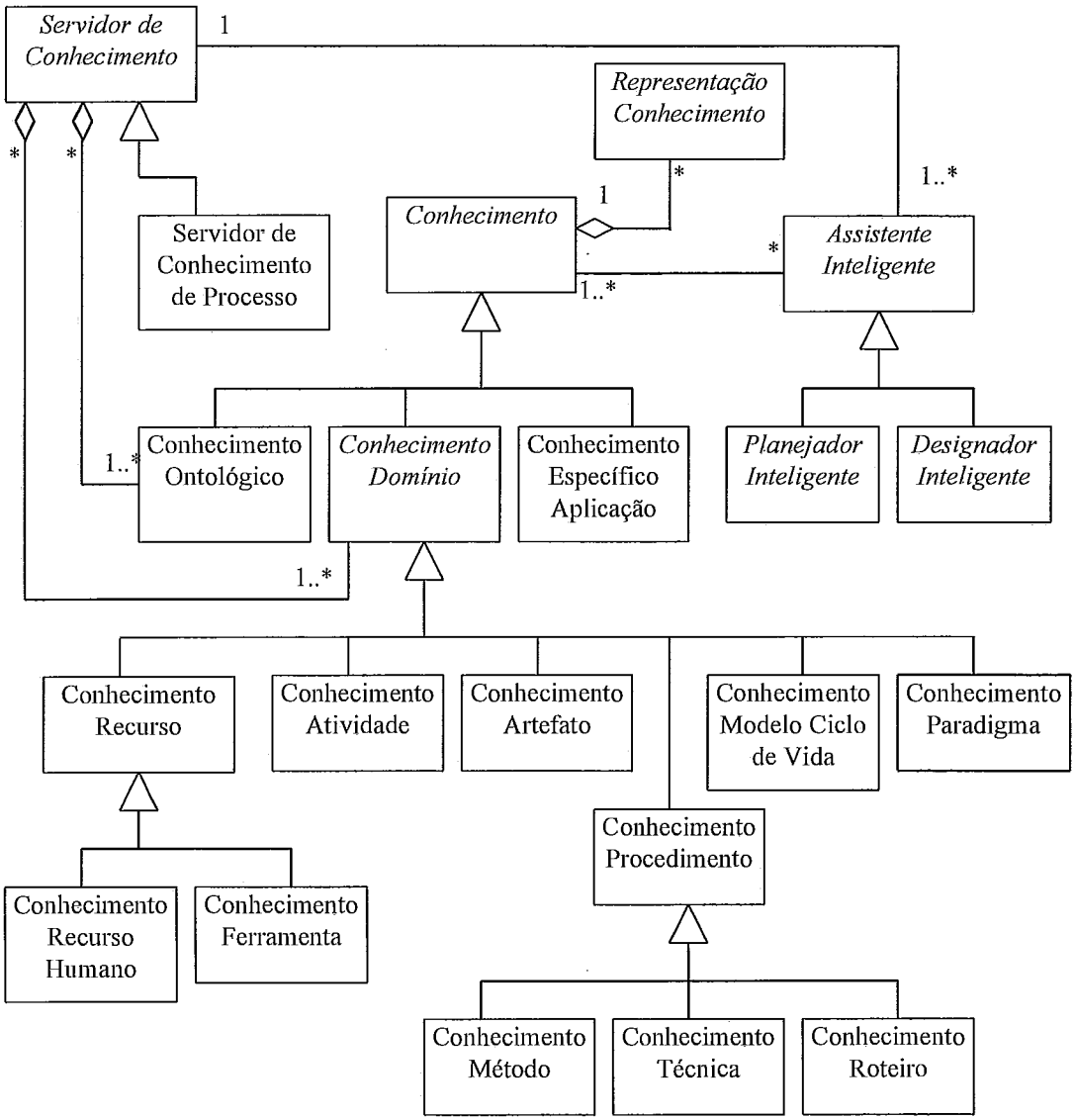


Figura 6.5 - O Novo Modelo de Conhecimento da Estação TABA.

A nova hierarquia das classes *Conhecimento* reflete as necessidades impostas pelos Servidores de Conhecimento. A hierarquia das classes *Conhecimento Domínio* é utilizada para mapear as categorias de conhecimento consideradas nas ontologias. No caso do Servidor de Conhecimento de Processo, esta hierarquia contempla o conhecimento de processos, paradigmas, modelos de ciclo de vida, atividades, artefatos, recursos e procedimentos, como mostra a figura 6.5. Através destas especializações da classe *Conhecimento Domínio*, é possível instanciar as ontologias, conforme discutido na seção 5.7. Em função das alterações na hierarquia das classes *Conhecimento*, foi necessário alterar o menu *Conhecimento* da janela principal da Estação TABA, que passou a apresentar a interface mostrada na figura 6.6. Para uma descrição completa deste ambiente, sua funcionalidade e interfaces, veja (ARAÚJO, 1998). As alterações feitas ao trabalho de ARAÚJO (1998) dizem respeito apenas às alterações do modelo do componente *Conhecimento* da Estação TABA, para comportar o novo modelo construído com base na ontologia de processo.

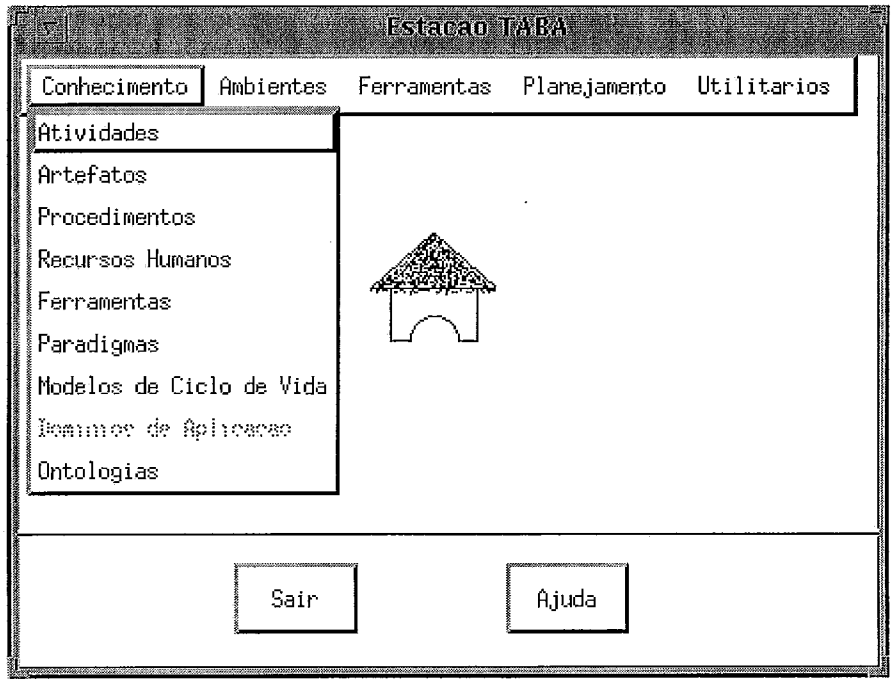


Figura 6.6 - Novo Menu Conhecimento da Estação TABA.

A classe *Conhecimento Domínio* pode, ainda, ser especializada para contemplar domínios de aplicação específicos, como por exemplo Direito e Medicina, permitindo, assim, a criação de Servidores de Conhecimento para estes domínios. Por

esta razão, há uma opção *Domínios de Aplicação* no menu *Conhecimento*, mostrado na figura 6.6. Um trabalho, neste sentido, já está sendo realizado (OLIVEIRA et al., 1998), visando a construção de um Ambiente de Desenvolvimento de Software Orientado a Domínio para Cardiologia.

O conhecimento inerente às ontologias em si, dado por seus axiomas, é descrito na classe *Conhecimento Ontológico.* Ou seja, os objetos desta classe representam o conhecimento capturado nas ontologias propriamente ditas, em contraste com os objetos das classes *Conhecimento Domínio*, que representam instanciações das ontologias e, portanto, conhecimento factual.

Uma vez que a versão corrente do Servidor de Conhecimento não dispõe de uma ferramenta para edição de ontologias, o *Conhecimento Ontológico* deve ser fornecido pelo engenheiro de conhecimento na forma de um arquivo previamente preparado. Assim, a janela correspondente à entrada deste tipo de conhecimento possui *layout* mostrado na figura 6.7.

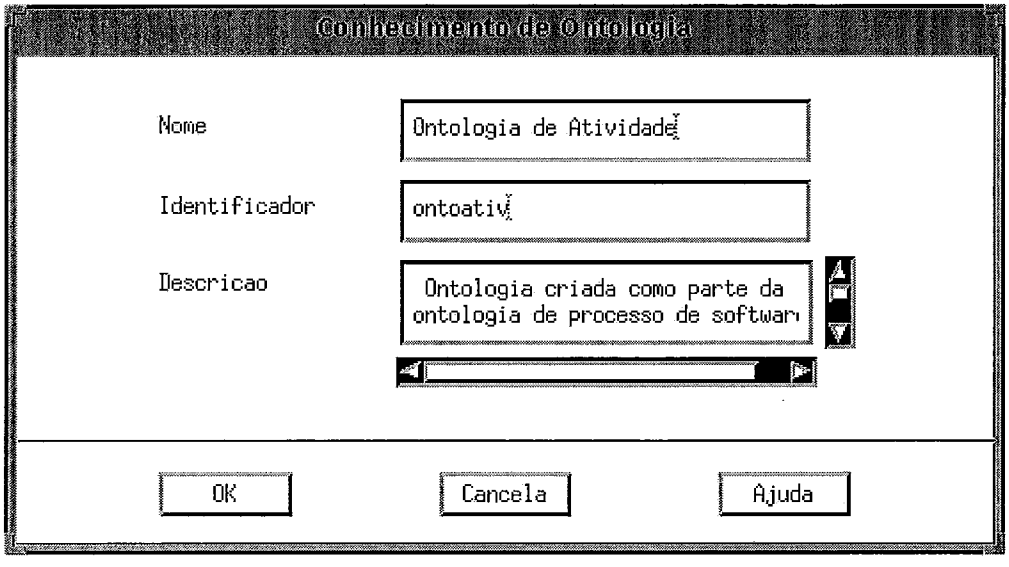


Figura 6.7 - Janela Conhecimento de Ontologias.

Os objetos da classe *Conhecimento Especifico Aplicação* capturam o conhecimento que é específico para uma particular aplicação baseada em conhecimento e, portanto, não integrado ao Servidor de Conhecimento.

A hierarquia das classes *Assistente Inteligente* contempla os dois resolvedores de problemas básicos para o domínio da definição de processos de software: planejamento e designação. De fato, qualquer *template* de resolvedor de problemas deve ser modelado como uma classe abstrata nesta hierarquia. Além disso, um modelo adaptado para uma aplicação específica deve ser modelado como uma sub-classe-folha da hierarquia e, portanto, uma classe concreta.

A classe *Servidor de Conhecimento* agrega a arquitetura dos Servidores de Conhecimento, descrita na figura 4.7. Os módulos de conhecimento, propostos nesta arquitetura, foram modelados como objetos da classe *Conhecimento*; os *templates* de resolvedores de problemas, por sua vez, foram modelados como sub-classes da classe *Assistente Inteligente*. Entretanto, no protótipo atual, um Servidor de Conhecimento não faz referências a resolvedores de problemas, uma vez que estes não são objetos, mas sim classes abstratas do componente *Suporte Inteligente* do modelo da Estação TABA. Assim, na implementação corrente, um Servidor de Conhecimento é uma agregação de módulos de conhecimento, dados pelas ontologias (objetos da classe *Conhecimento Ontológico*) e suas instanciações (objetos das sub-classes *Conhecimento Domínio*). Estes módulos de conhecimento estão implementados em Eiffel (MEYER, 1992), sendo utilizada a representação de conhecimento da classe *Sistema de Programação em Lógica*, discutida na seção 4.2.

A classe *Servidor de Conhecimento de Processo* é uma especialização da classe *Servidor de Conhecimento*, adaptada para o domínio de processos de desenvolvimento de software.

Os *templates* de resolvedores de problemas, conforme mencionado anteriormente, são sub-classes abstratas da classe *Assistente Inteligente*. A figura 6.8 mostra a classe *Designador Inteligente*, escrita em Eiffel, construída com base no modelo de tarefa da figura 6.3.

```
deferred class
        DESIGNADOR_INTELIGENTE
inherit
        ASSISTENTE_INTELIGENTE
feature
        componentes is
                deferred
                end;
        recursos is
                deferred
                end;
        designação_parcial is
                deferred
                end;
        restrições: LINKED_LIST [CONHECIMENTO];

        designar is
                do
                        entrar_dados;
                        propor;
                        revisar;
                end;

        entrar_dados is
                -- entrar componentes e recursos a serem designados
                -- deve ser parte de uma rotina de criação (creation feature)
                deferred
                end;

        propor is
                deferred
                end;
        revisar is
                deferred
                end;

end -- class DESIGNADOR_INTELIGENTE
```

Figura 6.8 - A Classe Abstrata Designador Inteligente.

6.3 - *Assist-Pro*: Assistente Inteligente para a Definição de Processos de Software

A definição de processos de software é, inegavelmente, uma tarefa que requer conhecimento intenso, sendo realizada por especialistas experientes. Para engenheiros de software menos experientes, esta é uma tarefa que potencialmente requer alguma forma de assistência inteligente. Assim, para mostrar como o Servidor de Conhecimento de Processo pode auxiliar a construção de ferramentas baseadas em conhecimento de processo de software, desenvolvemos *Assist-Pro,* um assistente inteligente para apoiar esta tarefa.

É importante realçar que, ao contrário de um sistema especialista - um sistema projetado para agir como um especialista e realizar o trabalho - *Assist-Pro* é projetado para auxiliar um engenheiro de software a definir um processo, fornecendo diretrizes, sugestões e opções, de forma a aumentar a qualidade e a produtividade nesta tarefa. Além disso, não é um objetivo deste trabalho a construção de uma ferramenta completa, capaz de assistir engenheiros de software em situações variadas. Nossa intenção é mostrar como Servidores de Conhecimento podem facilitar o desenvolvimento de assistentes inteligentes. Assim, o escopo de assistência desta ferramenta é limitado: contempla apenas processos de desenvolvimento para projetos de sistemas de informação e desconsidera a possível existência de elementos pré-definidos para o processo, tais como a obrigatoriedade de se empregar um método específico ou uma linguagem de programação particular.

6.3.1 - Definição de Processos de Software

De maneira geral, a definição de um processo de software envolve as seguintes tarefas:

- Definição de um ciclo de vida para o processo;
- Detalhamento das fases do ciclo de vida em atividades;
- Definição de como as atividades devem ser realizadas;
- Definição dos artefatos consumidos e produzidos por uma atividade;
- Definição dos recursos para as atividades;

Diferentes aplicações possuem diferentes perfis e estas diferenças têm influência sobre seus processos de desenvolvimento. Para que as tarefas anteriormente relacionadas possam ser realizadas, é imprescindível que algumas características gerais relacionadas ao processo de software sejam observadas. Desta forma, uma etapa preliminar para tal faz-se necessária. A seguir, os passos anteriormente relacionados são discutidos com mais detalhes.

Determinação das Características do Projeto

Nesta etapa, basicamente, deseja-se obter algumas informações sobre o projeto, que sejam relevantes para a definição de um processo, entre elas:

- *Complexidade do problema a ser tratado*: notadamente a complexidade do software a ser desenvolvido tem um grande impacto no ciclo de vida a ser adotado. Além disso, deve ser observado se o problema a ser tratado é bem definido, isto é, pode ser totalmente especificado no início do desenvolvimento.
- *Características da equipe e da gerência*: a formação, atualização e experiência da equipe são fatores importantes a serem considerados na definição do processo. Além disso, o grau de inovação que a gerência está disposta a enfrentar também tem impacto direto na definição do processo.
- *Responsabilidade pelo desenvolvimento*: deve-se definir quem desenvolverá o software. As seguintes possibilidades devem ser consideradas: o software será desenvolvido com equipe da própria empresa, ou com contratação de terceiros (no todo ou em parte), ou com equipe mixta (equipe composta por consultores e desenvolvedores da própria empresa). Este aspecto tem impacto direto na definição de atividades de gerência e de controle da qualidade.

Determinação do Ciclo de Vida para o Projeto

A definição do ciclo de vida é um dos passos mais importantes na definição de um processo de software. Para auxiliar esta etapa, existem vários modelos de ciclo de vida descritos na literatura. Assim, o ciclo de vida é definido com base em um modelo de ciclo de vida adequado à situação (dada pelas características discutidas anteriormente) e adaptado a ela.

Detalhamento das Fases do Ciclo de Vida em Atividades

Uma vez definido o ciclo de vida do projeto, tem-se apenas as macro-atividades que compõem o processo. É necessário, portanto, refinar estas macro-atividades em sub-atividades para se obter o conjunto total de atividades do processo.

Definição da Realização das Atividades

Para cada atividade do processo, deve-se definir como esta será realizada, isto é, que procedimentos serão adotados. É importante notar que, uma vez que métodos impõem uma particular decomposição de uma atividade em sub-atividades, a escolha dos métodos deve anteceder o completo detalhamento das atividades do processo. Técnicas, roteiros e normas, por sua vez, são aplicáveis apenas a atividades elementares e, portanto, sua escolha pressupõe a conclusão da tarefa anterior.

Definição dos Artefatos Consumidos e Produzidos pelas Atividades

Para cada atividade do processo é necessário definir que artefatos são consumidos pela atividade e que artefatos são produzidos por ela.

Definição dos Recursos Necessários para a Realização das Atividades

Finalmente, uma vez definidas todas as atividades que compõem o processo, deve-se determinar que recursos serão necessários para a realização das mesmas.

6.3.2 - Usando o Servidor de Conhecimento de Processo na Construção de Assist-Pro

Claramente, a definição de um processo de software é uma tarefa que mescla problemas do tipo planejamento e designação. Além disso, grande parte do conhecimento de domínio envolvido nesta tarefa está disponível na ontologia de processo de software, apresentada no capítulo 5. Assim, o Servidor de Conhecimento de Processo foi utilizado na construção de *Assist-Pro*, oferecendo toda a infra-estrutura básica.

É claro que algum conhecimento específico desta aplicação teve que ser elicitado e modelado, mas, ao invés de construir este assistente elaborando uma nova conceituação a partir do nada, o conhecimento disponível no Servidor de Conhecimento de Processo foi reutilizado e, mais importante, a ontologia de processo

de software forneceu um vocabulário básico e guiou a aquisição do conhecimento mais específico.

Adaptação dos Modelos de Tarefa do Servidor de Conhecimento

Analisando a descrição anterior de como proceder para definir um processo, é possível observar que, basicamente, a definição de processos de software envolve duas grandes tarefas: planejamento do processo e designação de procedimentos e recursos para suas atividades, como mostra a figura 6.9. Assim, os modelos de tarefa para estes dois tipos de problemas devem ser adaptados para o problema em questão.

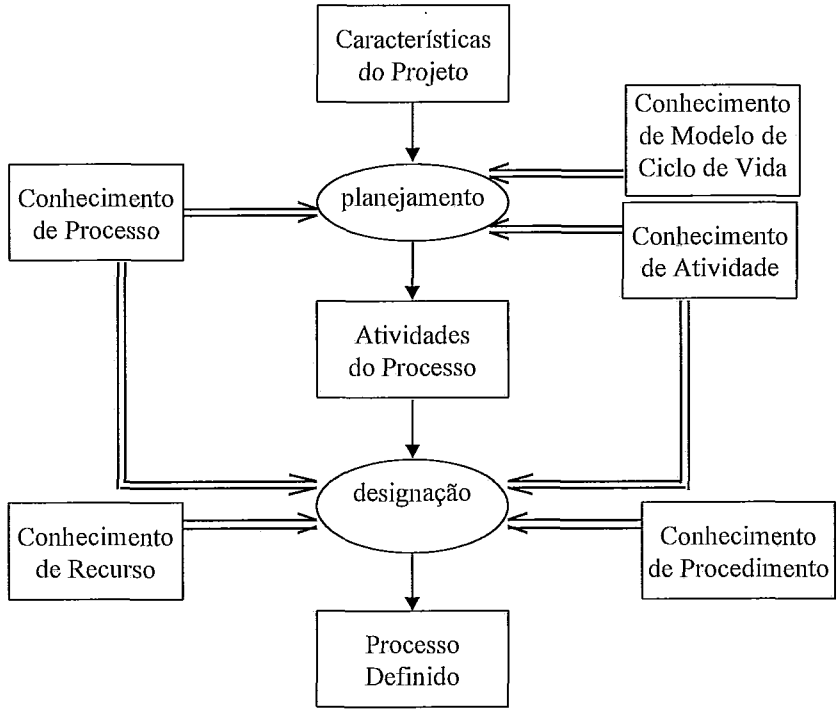


Figura 6.9 - Tipos de Problemas Relacionados com a Definição de um Processo.

Na etapa de planejamento, define-se o ciclo de vida do processo, suas etapas e um detalhamento dessas em atividades. O ciclo de vida é definido tomando por base um modelo de ciclo de vida básico, sendo que o engenheiro de software é livre para adaptá-lo às necessidades e restrições impostas, dadas nas características gerais.

Na etapa de designação, define-se como as atividades serão realizadas e os recursos requeridos. Na verdade, a etapa de designação ocorre em duas sub-etapas: uma designação de procedimentos a atividades, estabelecendo como realizá-las, e uma designação de recursos.

Ainda que a figura 6.9 mostre as tarefas de planejamento e designação isoladas, isto não ocorre na prática. De acordo com a ontologia de procedimento, métodos impõem uma decomposição particular para as atividades por eles apoiadas. Assim, o que acontece, de fato, é uma intercalação de tarefas de planejamento e designação, como mostra a figura 6.10. Nesta figura, não mostramos os papéis de conhecimento estáticos para tornar o modelo mais legível ao leitor.

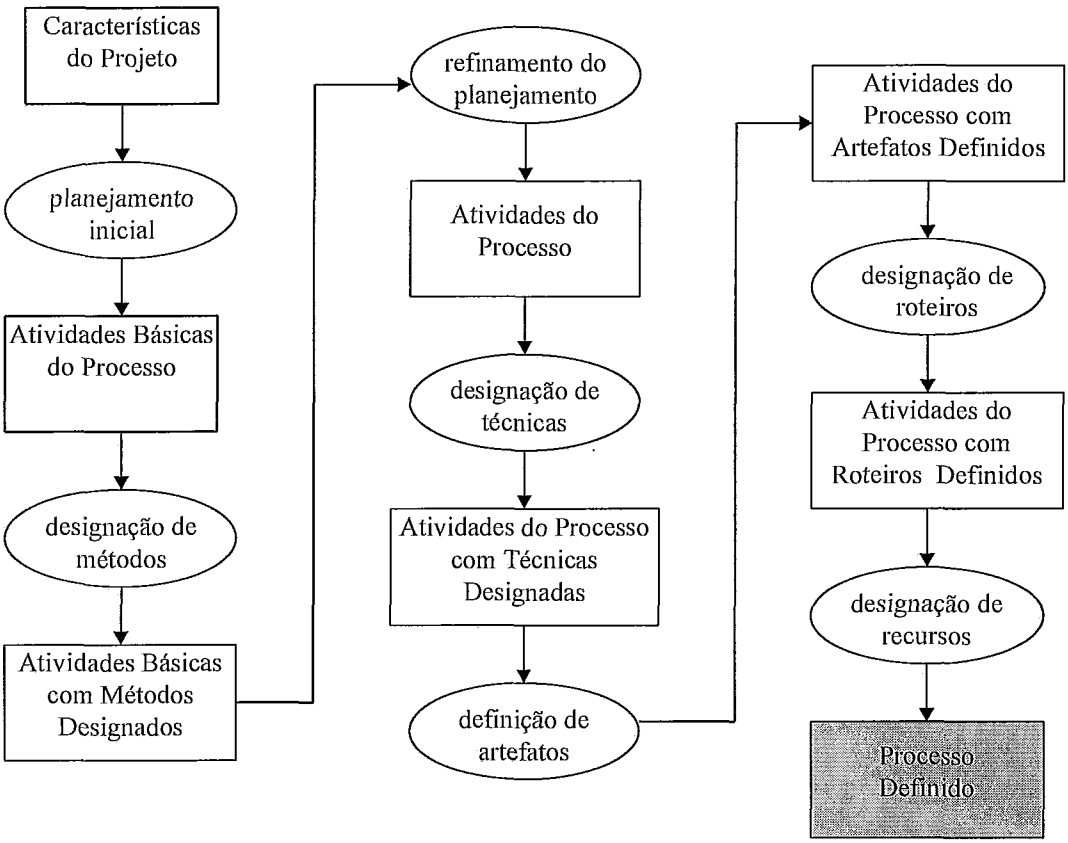


Figura 6.10 - Modelo de Tarefa Global para *Assist-Pro*.

A tarefa de *planejamento inicial* da figura 6.10 segue o modelo da figura 6.2, exceto pela função "especificar plano", que foi desmembrada em duas: a primeira - "especificar processo inicialmente", mostrada na figura 6.11 - é realizada antes da *designação de métodos*; a segunda - "refinar especificação processo"- corresponde à tarefa *refinamento do planejamento* e é realizada após a *designação de métodos*.

As tarefas de *designação de métodos, técnicas, roteiros e normas* compreendem a *designação de procedimentos* e têm, todas elas, a estrutura de funções mostrada na figura 6.12. Esta estrutura foi adaptada da figura 6.4.

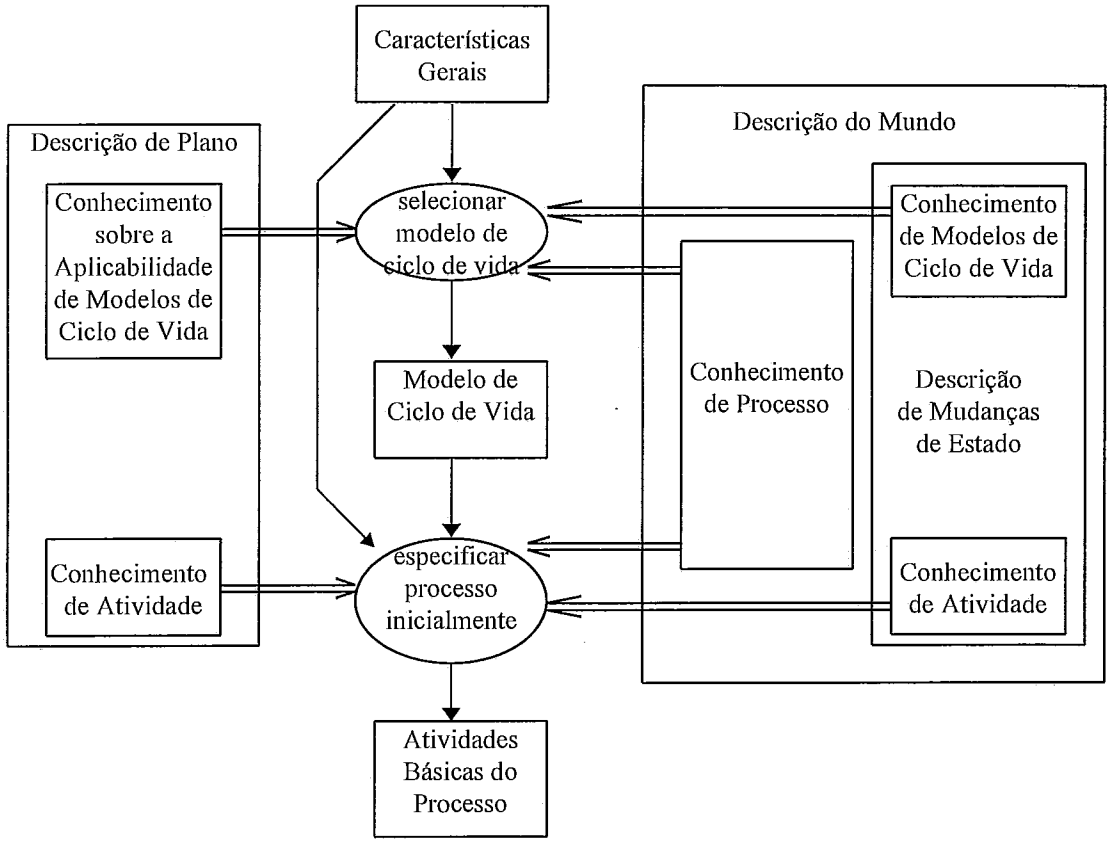


Figura 6.11 - Modelo de Tarefa de Planejamento Adaptado à Definição de Processos.

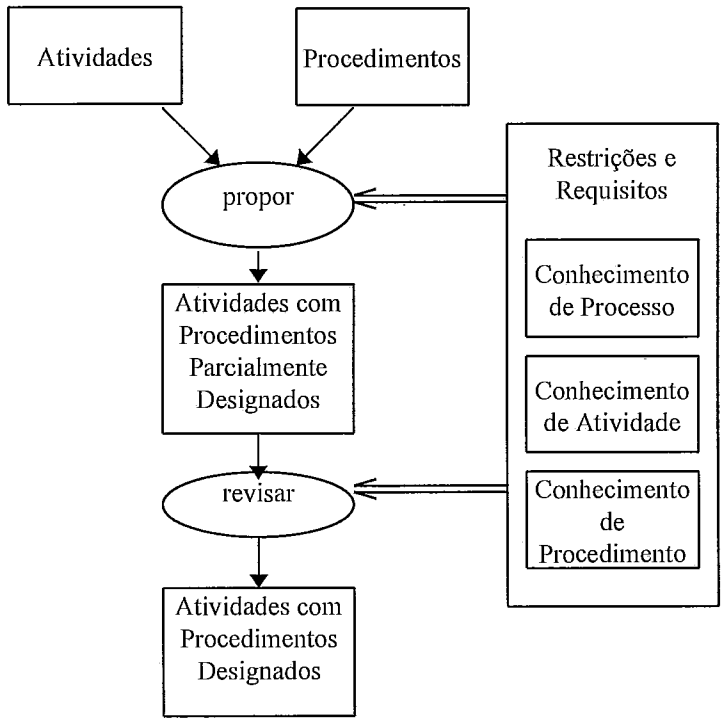


Figura 6.12 - Modelo de Tarefa de Designação Adaptado à Designação de Procedimentos a Atividades do Processo.

Para ilustrar como os templates de resolvedores de problema são adaptados a uma aplicação, a figura 6.13 mostra a especialização da classe *Designador Inteligente*, mostrada na figura 6.9, para a designação de procedimentos. Esta especialização foi construída com base no modelo de tarefa da figura 6.12.

```
class
        DESIGNADOR_INTELIGENTE_PROCEDIMENTOS

inherit
        DESIGNADOR_INTELIGENTE
                rename
                        componentes as atividade,
                        recursos as procedimentos,
                end;
creation
        entrar_dados
feature
        atividade: CONHECIMENTO_ATIVIDADE;
        procedimentos: LINKED_LIST[CONHECIMENTO_PROCEDIMENTO];
        restrições: LINKED_LIST[CONHECIMENTO];
        ontologia_procedimento: CONHECIMENTO_ONTOLÓGICO;
        ontologia_processo: CONHECIMENTO_ONTOLÓGICO;
        ontologia_atividade: CONHECIMENTO_ONTOLÓGICO;

        entrar_dados (ativ: CONHECIMENTO_ATIVIDADE) is
                -- associa os módulos de conhecimento correspondentes ao papéis
                -- de conhecimento do resolvedor de problemas
                do
                        ...
                end;
        propor is
                -- propõem designação de procedimentos à atividade de entrada
                do
                        ...
                end;
        revisar is
                -- revisa a designação proposta
                do
                        ...
                end;

end -- class DESIGNADOR_INTELIGENTE_PROCEDIMENTO
```

Figura 6.13 - A Classe Abstrata Designador Inteligente Adaptada para a Designação de Procedimentos.

Basicamente, os métodos *entrar_dados*, *propor* e *revisar* têm o seguinte comportamento:

- *entrar_dados*: inicializa os papéis de conhecimento descritos pelos atributos *atividade*, *procedimentos* e *restrições*. Este último consiste do conhecimento das ontologias de atividade, procedimento e processo;
- *propor*: usando uma base de conhecimento construída com os módulos de conhecimento associados aos atributos *atividade*, *procedimentos* e *restrições*, é proposto um conjunto inicial de procedimentos passíveis de serem designados à atividade em questão;
- *revisar*: cabe ao engenheiro de software responsável pela definição do processo a decisão de selecionar um dos procedimentos sugeridos pelo assistente, ou de designar um outro procedimento não sugerido.

A tarefa de *definição dos artefatos* requeridos e produzidos por uma atividade foi adaptada também do modelo de designação da figura 6.4. Neste caso, uma vez que a escolha de certos procedimentos determina que artefatos são necessários para a realização da atividade e quais são produzidos por ela, grande parte da definição de artefatos foi feita de forma automática, como mostra o papel de conhecimento de entrada "designação parcial", mostrado na figura 6.14.

A tarefa de *designação de recursos*, mostrada na figura 6.10, compreende, na realidade, a designação de recursos humanos, de hardware e de software. Neste trabalho, contudo, consideramos apenas recursos humanos e ferramentas de software, ambos seguindo o modelo da figura 6.15.

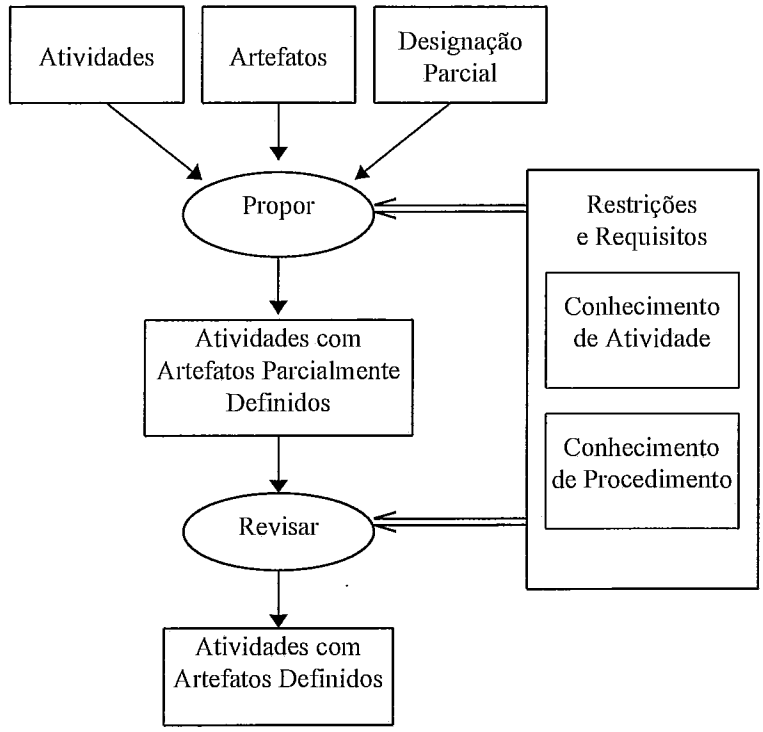


Figura 6.14 - Modelo de Tarefa de Designação Adaptado à Definição de Artefatos Requeridos e Produzidos por Atividades do Processo.

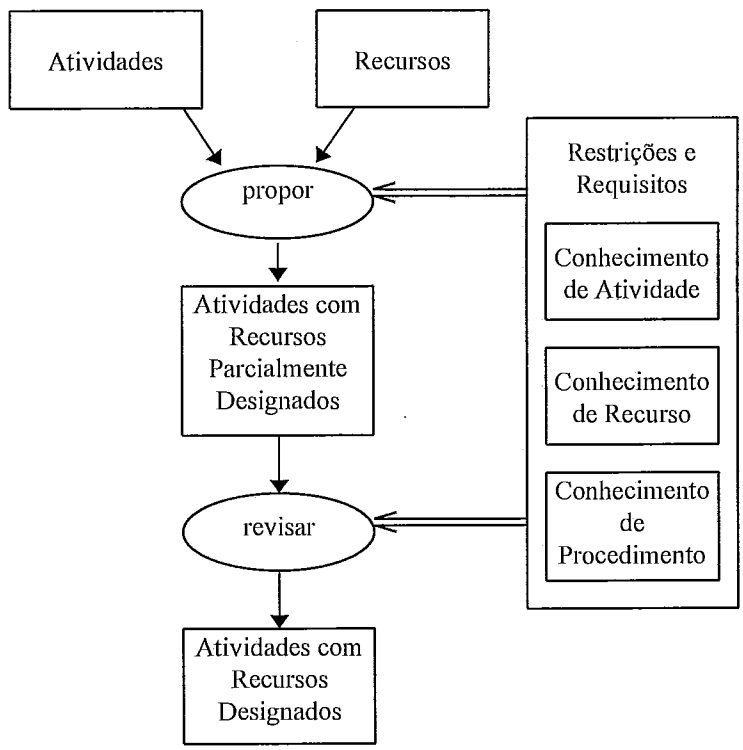


Figura 6.15 - Modelo de Tarefa de Designação Adaptado à Designação de Recursos a Atividades do Processo.

O Conhecimento Necessário

Analisando os modelos de tarefa definidos para *Assist-Pro* nas figuras 6.11, 6.12, 6.14 e 6.15, é possível notar que grande parte dos papéis de conhecimento descritos nestes modelos está disponível no Servidor de Conhecimento de Processo, na forma de módulos de conhecimento oriundos das ontologias e suas instanciações e, portanto, podem ser utilizados diretamente.

De fato, dentre os papéis de conhecimento estáticos, apenas o "Conhecimento sobre Aplicabilidade de Modelos de Ciclo de Vida" não foi capturado pela ontologia de processo e teve de ser elicitado e modelado. Além dele, para especificar inicalmente as atividades do processo, notadamente as atividades de gerência e avaliação da qualidade, foi necessário incorporar conhecimento sobre características gerais do projeto, tais como características da equipe de desenvolvimento e responsabilidades pelo desenvolvimento. Este conhecimento foi incorporado ao papel de conhecimento "Conhecimento de Processo".

Esta é a idéia central de Servidores de Conhecimento: a construção de um assistente inteligente pode ser feita usando toda uma infra-estrutura de conhecimento pré-existente. Modelos de tarefa são adaptados para a aplicação específica; módulos de conhecimento são associados aos papéis de conhecimento dos modelos de tarefa adaptados; e aqueles papéis de conhecimento que não forem completamente preenchidos por módulos de conhecimento do Servidor, deverão ser o alvo da segunda etapa da aquisição do conhecimento.

O Conhecimento Específico da Aplicação

O primeiro passo para a definição de um processo de desenvolvimento de software consiste em selecionar um modelo de ciclo de vida para o projeto. Modelos de ciclo de vida são selecionados em função de fatores como: tecnologia de desenvolvimento e paradigma a serem aplicados no desenvolvimento do software, características do problema a ser resolvido, características do software a ser desenvolvido e características da equipe de desenvolvimento.

Parte deste conhecimento está descrito na ontologia de processo. Entretanto, aspectos como características do problema e da equipe de desenvolvimento não foram capturados. Assim, uma nova etapa de aquisição de conhecimento teve de ser

realizada. Esta etapa foi, fortemente, guiada pela ontologia. Primeiramente, o conhecimento foi elicitado apenas para modelos de ciclo de vida já descritos como instanciações da ontologia, a saber: em cascata, incremental, RAD, evolutivo básico, prototipagem operacional e paralelo/recursivo. Em segundo lugar, o vocabulário básico definido pela ontologia de processo foi usado para descrever o conhecimento, sendo necessário, obviamente, estendê-lo para considerar termos específicos, não previstos na ontologia. As tabelas 6.1 e 6.2 apresentam, de forma compacta, o conhecimento elicitado.

Tabela 6.1 - Características do Desenvolvimento e do Problema a ser resolvido e a adequabilidade de Modelos de Ciclo de Vida.

	Aplicabilidade				
	Características do Desenvolvimento		**Características do Problema**		
Modelo de Ciclo de Vida	Paradigma	Software pode ser colocado em uso rapidamente ? (com funcionalidade total/parcial)	Grau de Definição do Problema	Tamanho	Modularidade
Cascata	Estrutural / OO	não	bem-definido	pequeno	qualquer
Incremental	Estrutural / OO	sim	bem-definido	médio a grande	média a alta
RAD	Estrutural / OO	sim	bem-definido	pequeno a médio	alta
Evolutivo Básico	OO	sim	bem ou mal-definido	qualquer	qualquer
Prototipagem Operacional	OO	não	bem ou mal-definido	pequeno a médio	qualquer
Paralelo / Recursivo	OO	sim	bem ou mal-definido	médio a grande	qualquer

Além de ser útil para a seleção de um modelo de ciclo de vida para o processo, o conhecimento sobre as características da equipe de desenvolvimento e da gerência, mostrado na tabela 6.2, é também importante para a definição de atividades de treinamento a serem incorporadas no processo.

Tabela 6.2 - Características da Equipe de Desenvolvimento e da Gerência e a Adequabilidade de Modelos de Ciclo de Vida.

Modelos de Ciclo de Vida	Aplicabilidade Características da Equipe de Desenvolvimento e da Gerência		
	Nível de Formação	Nível de Atualização	Nível de Experiência
Cascata	baixo	baixo	baixo
Incremental	baixo	médio	médio
RAD	médio	médio	alto
Evolutivo Básico	médio	médio	médio
Prototipagem Operacional	médio	médio	médio
Paralelo / Recursivo	alto	alto	alto

Finalmente, foi necessário capturar algum conhecimento sobre a responsabilidade pelo desenvolvimento. Basicamente, foram consideradas as seguintes possibilidades:

- software desenvolvido na empresa, sendo a equipe composta apenas por membros da empresa;
- software desenvolvido com contratação de terceiros, no todo ou em parte;
- software desenvolvido por equipes mistas.

Estas diferentes possibilidade de responsabilidades pelo desenvolvimento têm impacto na definição de atividades de gerência e controle da qualidade, além de revelar novos papéis de recursos humanos no contexto do desenvolvimento.

6.3.3 - A Implementação de Assist-Pro

Assist-Pro está disponível como uma opção do menu *Utilitários* do meta-ambiente da Estação TABA, como mostra a figura 6.16. Uma vez selecionada esta opção, uma sessão de Assist-Pro é iniciada, apresentando a janela mostrada na figura 6.17. Selecionado um projeto, o primeiro passo consiste em fornecer suas características gerais, como mostra a figura 6.18. Estas características correspondem

às informações necessárias para o planejamento do processo, relacionadas nas tabelas 6.1 e 6.2. A figura 6.19 mostra a janela correspondente às características do desenvolvimento, mostradas na tabela 6.1.

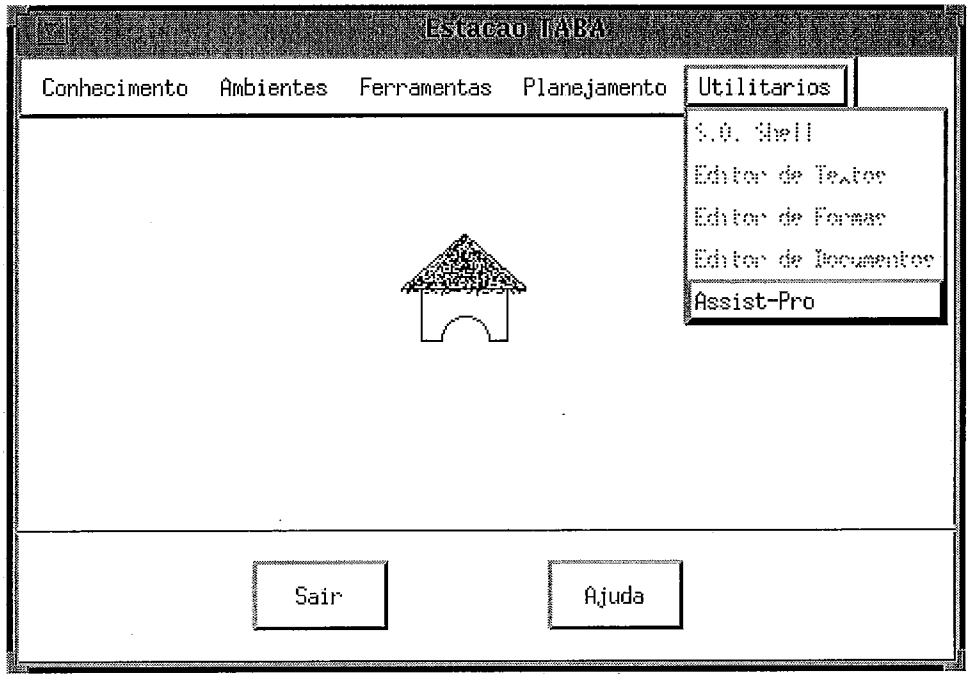


Figura 6.16 - Menu Utilitários da Estação TABA e a Opção Assist-Pro.

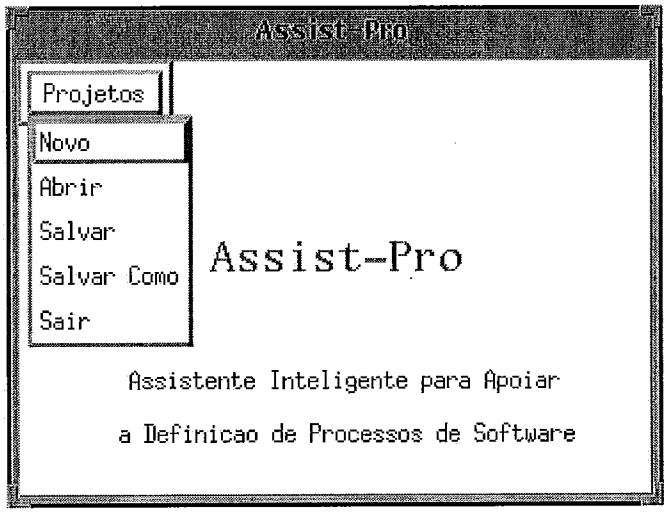


Figura 6.17 - Janela Principal de Assist-Pro.

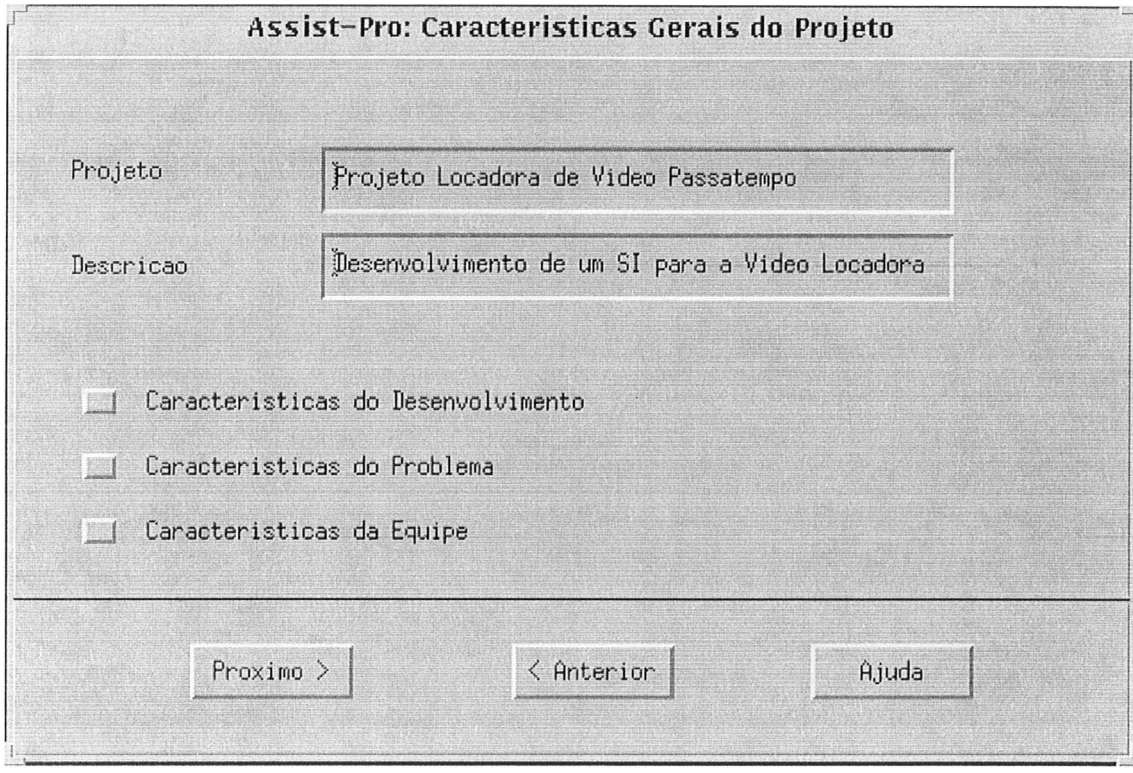


Figura 6.18 - Janela de Características Gerais do Projeto de Assist-Pro.

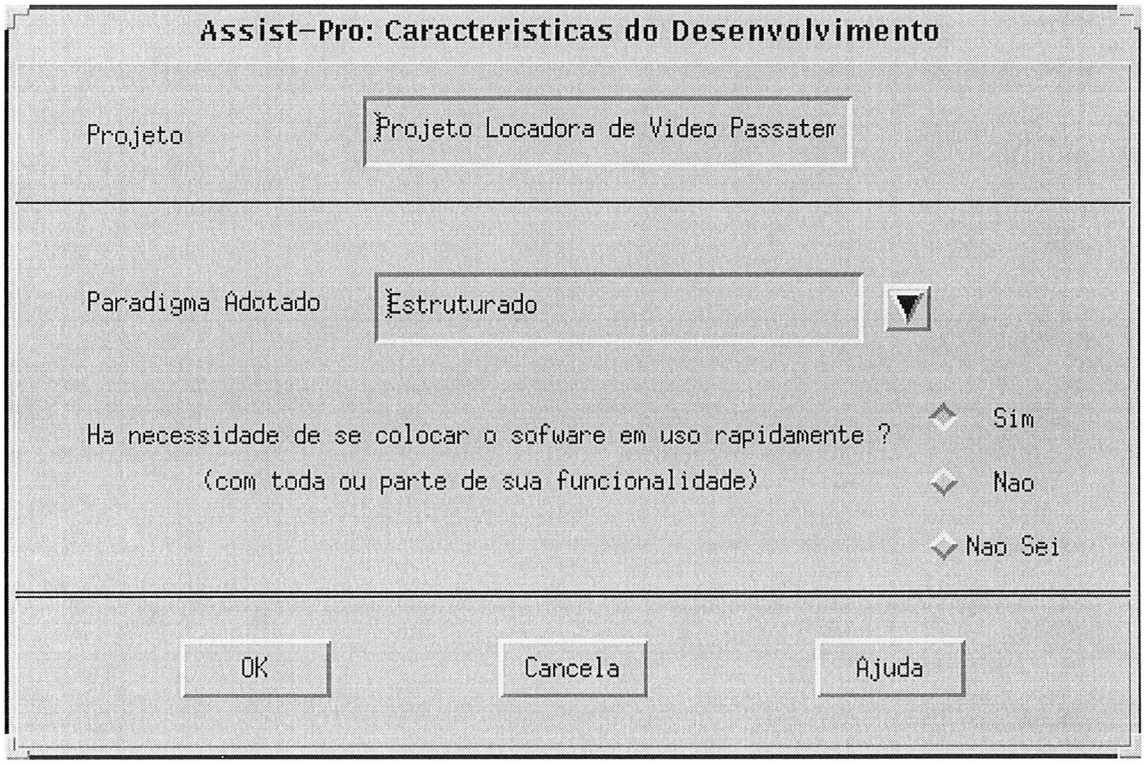


Figura 6.19 - Janela de Características do Desenvolvimento de Assist-Pro.

Uma vez fornecidas as características do projeto, a seqüência de funções do modelo de tarefa global de Assist-Pro, mostrado na figura 6.11, pode ser iniciada. O planejamento inicial corresponde à seleção de um modelo de ciclo de vida e utiliza a janela mostrada na figura 6.20.

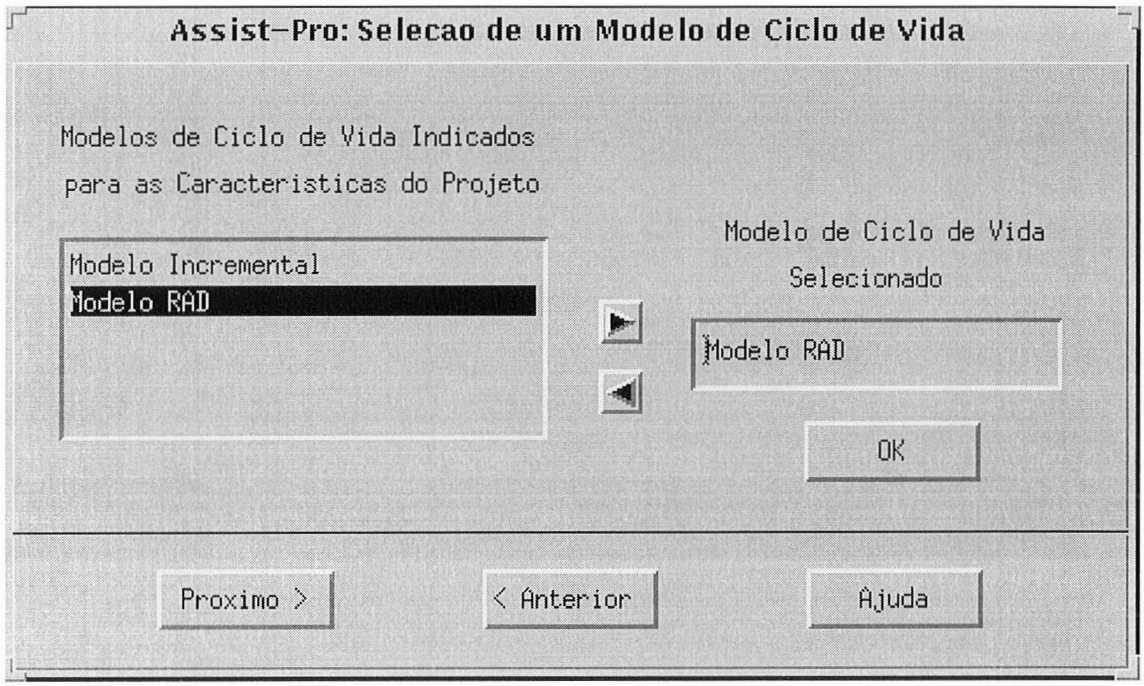


Figura 6.20 - Janela de Seleção de um Modelo de Ciclo de Vida para o Processo.

Nesta janela, são apresentados, para seleção, apenas os modelos de ciclo de vida adequados às características do projeto. Este filtro é feito através da base de conhecimento de Assist-Pro que indica a adequabilidade dos modelos de ciclo de vida.

Nesta etapa, o engenheiro de software pode, ainda, alterar a estrutura básica do modelo de ciclo de vida selecionado, inserindo novas atividades ou removendo atividades propostas pelo modelo de ciclo de vida. Para tal a janela da figura 6.21 é utilizada.

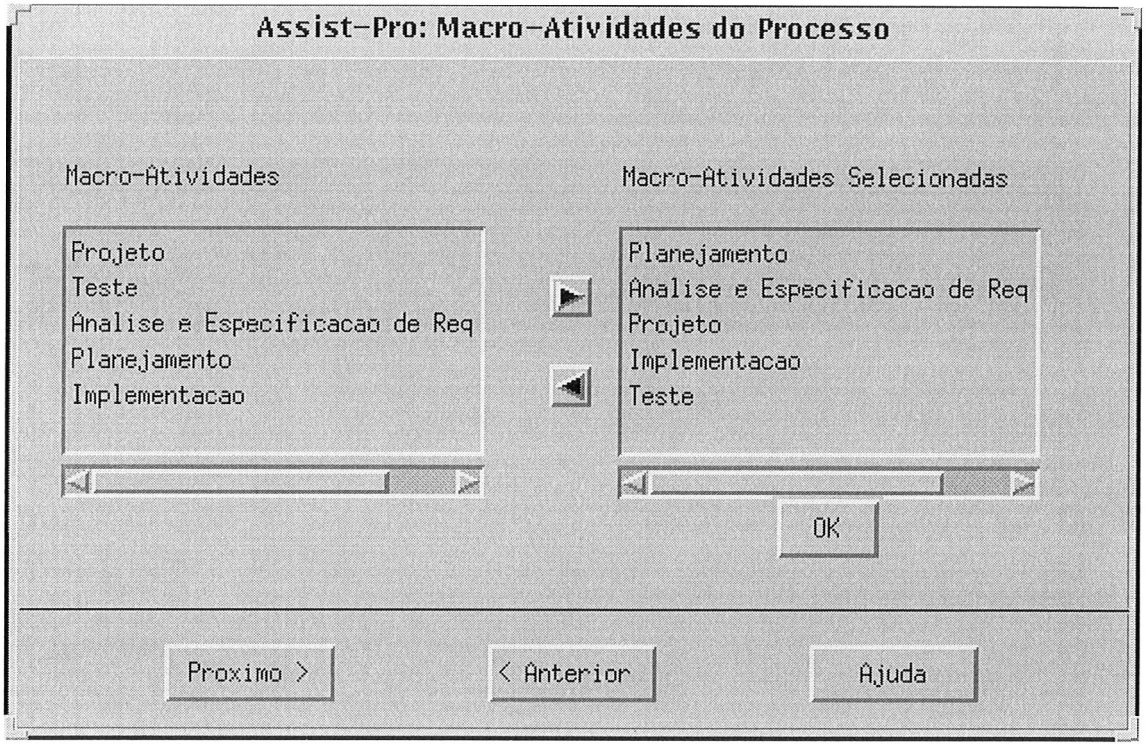


Figura 6.21 - Janela de Adaptação do Modelo de Ciclo Selecionado.

Uma vez selecionado e adaptado o modelo de ciclo de vida, o assistente solicita que sejam designados métodos para algumas de suas atividades, usando a janela mostrada na figura 6.22. Esta tarefa é também guiada por uma base de conhecimento, esta construída utilizando o conhecimento da ontologia de processo e suas instanciações.

A seguir, procede-se o refinamento do planejamento, onde as atividades básicas do processo são refinadas em sub-atividades, usando a janela da figura 6.23. Neste passo, caso ainda não tenha sido feito um detalhamento para a atividade, a lista de sub-atividades selecionadas é preenchida com a indicação provida pela base de conhecimento correspondente às instanciações das ontologias.

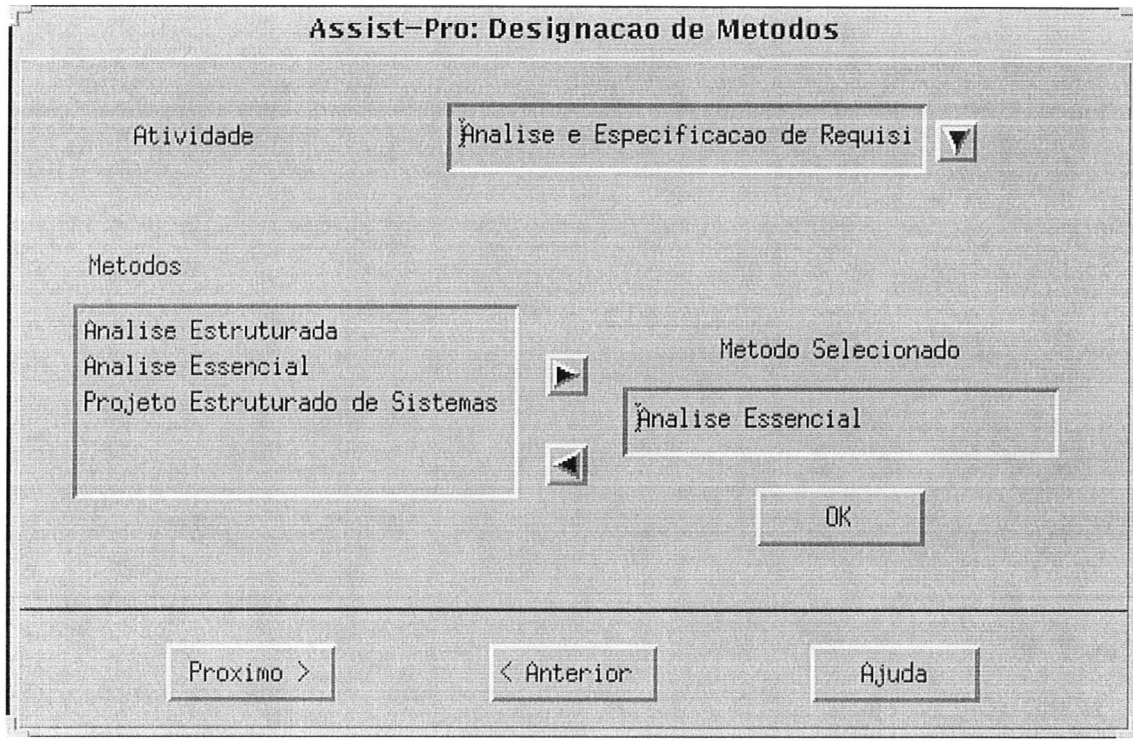


Figura 6.22 - Janela de Designação de Métodos a Atividades.

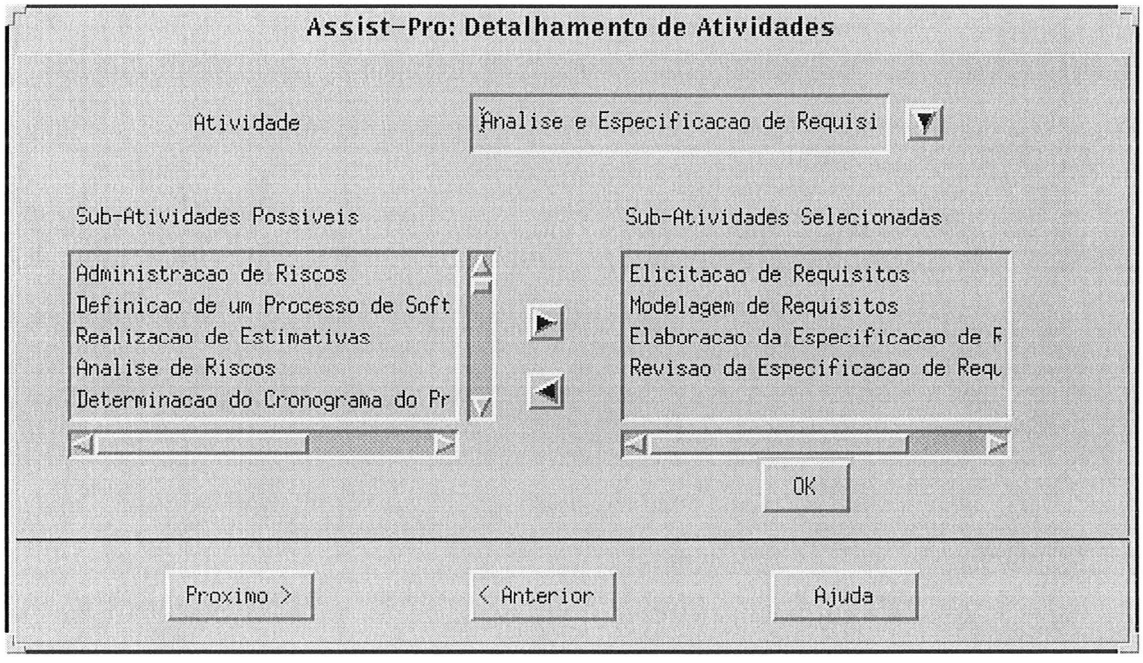


Figura 6.23 - Janela de Refinamento das Atividades.

Na seqüência, são designados técnicas, artefatos, roteiros e recursos às atividades. Em todos estes passos são utilizadas janelas similares à janela de designação de técnicas a atividades, mostrada na figura 6.24.

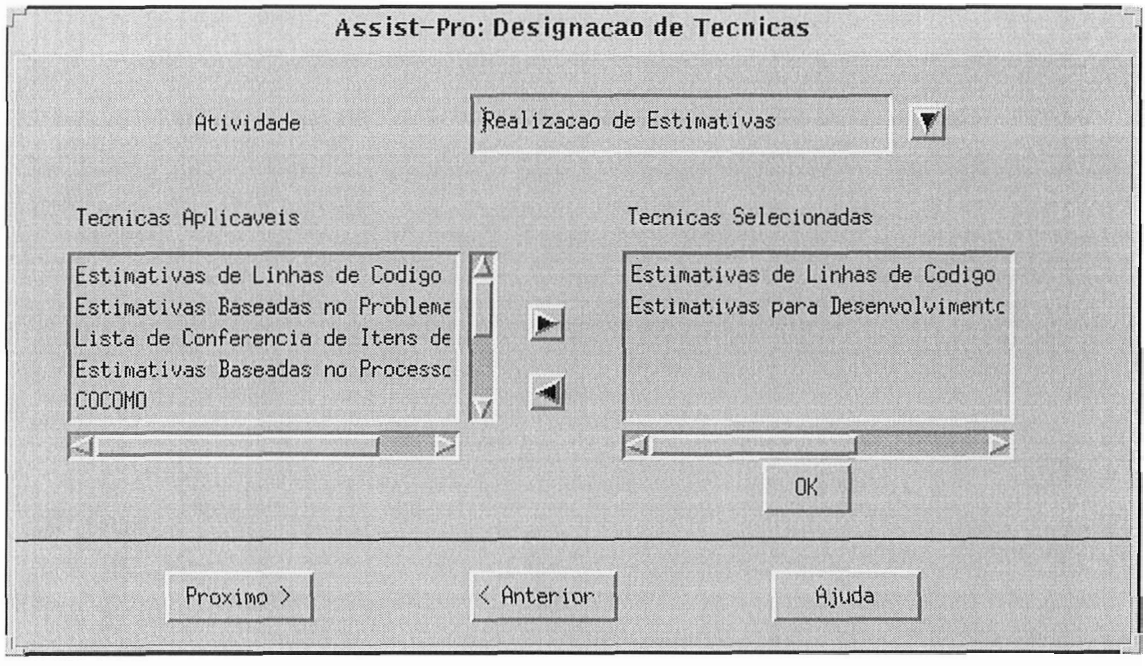


Figura 6.24 - Janela de Designação de Técnicas a Atividades.

Em todos os passos, o engenheiro pode retornar ao passo anterior ou solicitar ajuda. A figura 6.25 mostra a janela de tópicos de ajuda da janela da figura 6.20, onde são apresentados os tópicos de ajuda relevantes para o contexto da seleção de um modelo de ciclo de vida. Selecionado um tópico, o texto de ajuda correspondente é exibido, como mostra a figura 6.26.

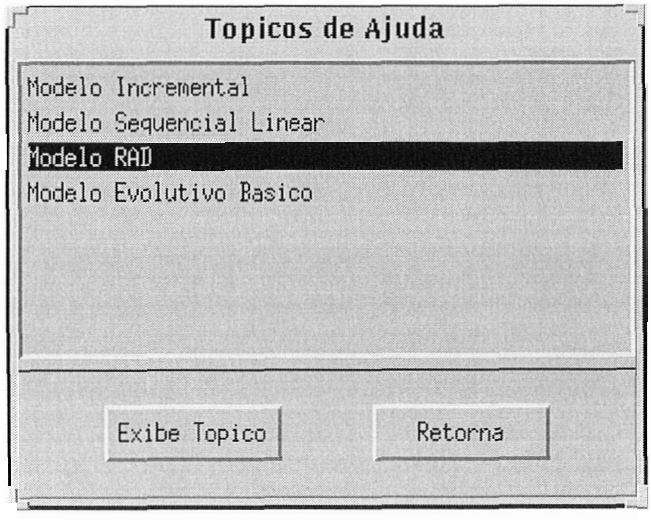


Figura 6.25 - Uma Janela de Tópicos de Ajuda.

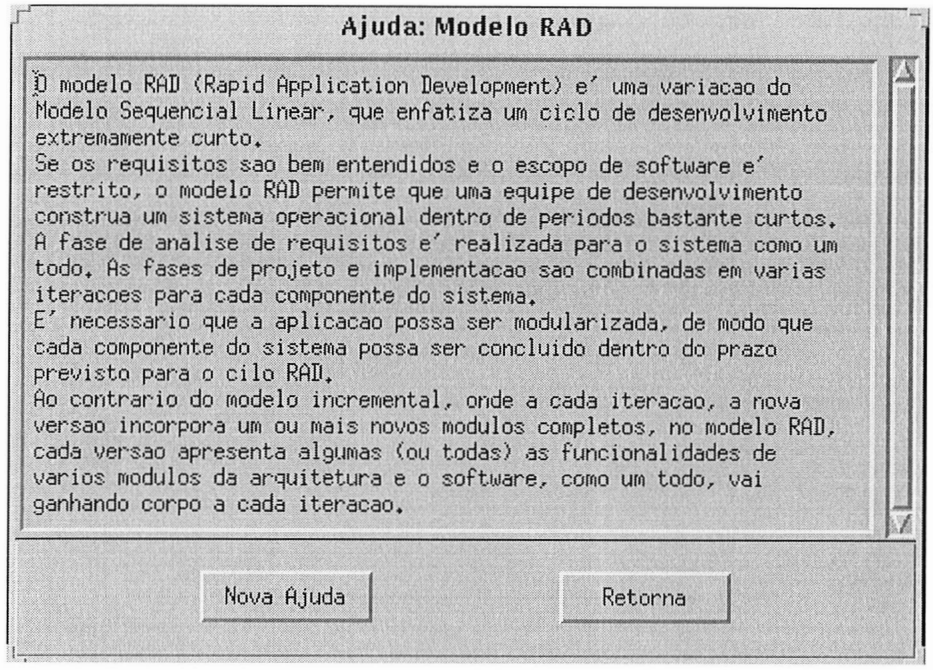


Figura 6.26 - Um Janela de Ajuda Típica.

Seguindo a seqüência de interações com Assist-Pro, mostrada na figura 6.10, tem-se, ao final, um processo definido, como um objeto da classe *Processo* do componente *Controle dos Processos* (ARAÚJO, 1998).

6.4 - Conclusões do Capítulo

Neste capítulo apresentamos o protótipo de um Servidor de Conhecimento de Processo, construído para a Estação TABA. Muitas propostas discutidas nesta tese não foram consideradas no desenvolvimento do protótipo, por requererem a construção de ferramentas de apoio, a serem tratadas em trabalhos futuros. O protótipo do Servidor de Conhecimento, por exemplo, não trata a seleção e adaptação de modelos de tarefa e seus *templates* de resolvedores de problema correspondentes. Na realidade, o protótipo corrente não possui referências aos *templates* de resolvedores porque estes são sub-classes da classe *Assistente Inteligente* e não objetos. Este problema pode ser solucionado de duas formas:

- criando-se uma classe *Modelo de Tarefa*, representando os modelos de tarefa suportados pelo Servidor de Conhecimento. Esta classe deveria oferecer a funcionalidade necessária para gerar o código de um resolvedor

de problema correspondente ao modelo de tarefa em mãos, isto é, para criar uma especialização da classe *Assistente Inteligente*;

- usando uma arquitetura de meta-nível (LISBÔA, 1997), onde classes possam ser tratadas também como objetos.

A primeira abordagem está bastante afinada com o método de Engenharia de Conhecimento baseada em Reutilização proposto na seção 3.4.3, utilizado neste capítulo para a construção de um assistente inteligente para a definição de processos. Entretanto, requer uma ferramenta para edição de modelos de tarefa e um gerador de código de templates de resolvedores de problema. Uma possível implementação da classe *Servidor de Conhecimento* adotando esta abordagem é mostrada na figura 6.24.

A despeito das limitações do protótipo do Servidor de Conhecimento de Processo, a meta principal deste trabalho, a integração de conhecimento em Ambientes de Desenvolvimento de Software, pode ser estudada. Oferecendo um conjunto de módulos de conhecimento, implementados como objetos da classe *Conhecimento* e suas especializações, e um conjunto de templates de resolvedores de problemas, implementados como especializações da classe *Assistente Inteligente*, associados a uma documentação consistente, capaz de guiar o uso dos componentes de conhecimento, foi possível notar que a construção de ferramentas baseadas em conhecimento pode ser facilitada com o suporte de um Servidor de Conhecimento. Esta constatação foi corroborada pela construção de *Assist-Pro*, um assistente inteligente para apoiar a definição de processos de software no meta-ambiente da Estação TABA. Entretanto, a real utilidade desta infra-estrutura só poderá ser sentida na medida em que um maior número de aplicações baseadas em conhecimento a utilize.

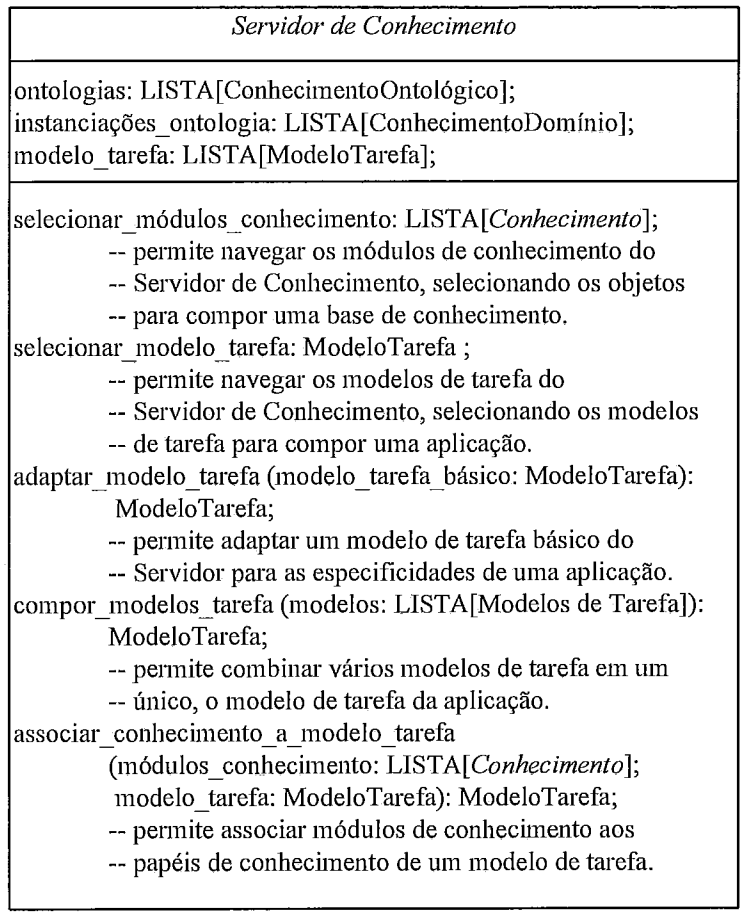


Figura 6.24 - Servidores de Conhecimento Considerando Modelos de Tarefa.

Capítulo 7

Conclusões e Perspectivas Futuras

7.1 - Conclusões

A integração em Ambientes de Desenvolvimento de Software (ADSs) tem sido tratada como uma questão de três dimensões: dados, controle e apresentação. Entretanto, com o aumento da complexidade dos produtos de software desenvolvidos, e conseqüentemente de seus processos de desenvolvimento, faz-se necessário oferecer assistência inteligente para que o desenvolvedor possa realizar suas tarefas com a qualidade e a produtividade desejadas. À medida que aumenta o número de ferramentas baseadas em conhecimento em um ADS, passa a ser necessário considerar mais uma dimensão na integração: a integração de conhecimento (TRAVASSOS, 1994).

A integração de conhecimento em um ADS requer uma nova abordagem para o desenvolvimento de assistentes inteligentes. O conhecimento tem de ser elicitado, modelado e implementado de forma a poder ser reutilizado e compartilhado.

Este trabalho teve como objetivo básico propor um modelo de integração de conhecimento para ADSs, tendo como referencial a Estação TABA (ROCHA et al., 1990). Em essência, defendemos a tese de que é necessário prover uma infra-estrutura de conhecimento capaz de apoiar o desenvolvimento de ferramentas baseadas em conhecimento em um universo de discurso. A esta infra-estrutura demos o nome de *Servidor de Conhecimento*.

Inicialmente, concentramos esforços na questão da representação de conhecimento em um ADS (FALBO, 1997a). Entretanto, logo percebemos que a integração não poderia ser atingida apenas com uma abordagem de nível simbólico (NEWELL, 1982). Assim, mudamos o enfoque para o nível de conhecimento, onde passamos a considerar o uso de ontologias como o aspecto central.

Ontologias têm por objetivo firmar um acordo sobre o vocabulário do domínio de interesse, a ser compartilhado por agentes que nele atuam. Usando ontologias para capturar o conhecimento sobre um universo de discurso e tornando este conhecimento disponível em um Servidor de Conhecimento, foi possível estabelecer uma abordagem prática para a integração de conhecimento.

Entretanto, quando falamos em reuso e compartilhamento de conhecimento, temos que considerar, além do conhecimento de domínio capturado pelas ontologias, o conhecimento sobre tarefas. Para tal, utilizamos os modelos de tarefa de CommonKADS (VALENTE et al., 1994) para desenvolvermos *templates* de resolvedores genéricos de problemas. Assim, basicamente, um Servidor de Conhecimento para um domínio específico oferece:

- um conjunto de bases de conhecimento modulares e reutilizáveis, construídas a partir de ontologias do domínio e,
- um conjunto de máquinas de inferência customizadas para os tipos de problema mais comumente encontrados neste domínio.

Para materializar as discussões a cerca da integração de conhecimento através de Servidores de Conhecimento, desenvolvemos um protótipo de um Servidor de Conhecimento de Processo para a Estação TABA. Além disso, para mostrar como um Servidor de Conhecimento pode apoiar a construção de ferramentas baseadas em conhecimento, utilizamos o protótipo desenvolvido na construção de *Assist-Pro*, um assistente inteligente para apoiar a definição de processos de software no meta-ambiente TABA.

A concepção de Servidores de Conhecimento proposta neste trabalho mostrou ser muito útil para a integração de conhecimento em ADSs. Entretanto, para ganhar um caráter realmente prático, é necessário que o conhecimento tornado disponível possa ser explorado. Neste contexto, é imprescindível a construção de um ambiente

para processamento de conhecimento que permita, entre outras coisas: (i) construir, estender e navegar ontologias e modelos de tarefa, (ii) selecionar e adaptar os componentes de conhecimento para uma aplicação específica, e (iii) assistir o desenvolvedor na construção de uma aplicação a partir dos componentes adaptados.

O uso de ontologias mostrou ser uma ferramenta útil na aquisição de conhecimento. Enquanto na Engenharia de Conhecimento tradicional, para cada nova aplicação a ser construída, uma nova conceituação é elaborada, em uma abordagem baseada em ontologias, o conhecimento geral do domínio, relevante para uma grande variedade de sistemas, é capturado e especificado na forma de ontologias. Estas, por sua vez, são usadas para guiar a aquisição do conhecimento específico de uma aplicação. Desta forma, uma mesma ontologia pode ser usada para guiar o desenvolvimento de várias aplicações, diluindo os custos da aquisição e permitindo reutilização e compartilhamento de conhecimento.

Além do uso de ontologias de domínio, o uso de modelos de tarefa também se mostrou bastante útil para a elaboração de *templates* de resolvedores de problemas. Uma abordagem mais uniforme, contudo, poderia considerar o uso de ontologias de tarefa para guiar a construção destes *templates*.

Uma vez que o uso de ontologias foi o ponto-chave de nossa proposta para a integração de conhecimento em ADSs e a construção de ontologias é ainda um campo de estudos em aberto, foi proposto também um método para a Engenharia de Ontologias. A ontologia de processo de software, desenvolvida como base para o Servidor de Conhecimento de Processo, foi construída usando este método. Além disso, tendo em vista que um Servidor de Conhecimento é, de fato, uma infra-estrutura para apoiar a construção de ferramentas baseadas em conhecimento no ADS, fez-se necessário estabelecer como utilizar esta infra-estrutura no desenvolvimento de assistentes inteligentes. Assim, uma outra contribuição deste trabalho foi a proposta de uma abordagem de Engenharia de Conhecimento baseada em Reutilização, a ser utilizada sempre que houver uma infra-estrutura de componentes reutilizáveis de conhecimento, como são os Servidores de Conhecimento. Esta abordagem foi aplicada no desenvolvimento de Assist-Pro, que foi construído utilizando o Servidor de Conhecimento de Processo.

A abordagem de Servidores de Conhecimento proposta pode ser muito útil para o estudo de ADSs Orientados a Domínio (FISCHER, 1994). Usando ontologias para capturar o conhecimento sobre um domínio e tornando este conhecimento disponível, na forma de Servidores de Conhecimento para o ADS e suas várias ferramentas, é possível avançar em direção a ADSs baseados em conhecimento e orientados a domínio (OLIVEIRA et al., 1998).

Ainda que a integração no nível de conhecimento (NEWELL, 1982) tenha se mostrado bem mais satisfatória do que no nível simbólico, idealmente devemos considerar ambos os níveis para obtermos a integração de conhecimento em ADSs. Não existe uma forma de representar conhecimento que possa ser considerada a melhor. O que existe é uma maior adequação ao problema que se quer resolver. Assim, uma representação na forma de Sistemas de Programação em Lógica, como a utilizada neste trabalho, pode ser bastante adequada para um domínio como Direito, porém menos adequada para outro domínio, como Medicina. Além disso, a representação de conhecimento usando o Prolog foi insuficiente para representar todo o conhecimento expresso nas ontologias. Deste modo, é necessário estudar formas de representação de conhecimento e a interoperabilidade entre elas, de modo a melhor representar cada porção de conhecimento, tornando disponíveis corpos de conhecimento para finalidades diferentes.

Em suma, foram contribuições deste trabalho:

- a proposta de um modelo de integração de conhecimento em ADSs, concretizada na forma de Servidores de Conhecimento;
- a definição de um método para a Engenharia de Ontologias;
- a definição de um método de Engenharia de Conhecimento baseada em Reutilização;
- o desenvolvimento de uma ontologia de processos de desenvolvimento de software, usando o método de Engenharia de Ontologias proposto;
- o desenvolvimento de um protótipo de um Servidor de Conhecimento de Processo;
- o desenvolvimento de um assistente inteligente para apoiar a definição de processos de desenvolvimento de software, usando o Servidor de

Conhecimento de Processo e aplicando o método de Engenharia de Conhecimento proposto.

7.2 - Perspectivas Futuras

Visando tornar prático o uso de Servidores de Conhecimento, é necessário facilitar a construção, o acesso, a compreensão, a instanciação e a adaptação de seus componentes: ontologias, e os módulos de conhecimento delas derivados direta ou indiretamente (instanciações das ontologias), e modelos de tarefa, e os *templates* de resolvedores de problemas resultantes. Assim, um primeiro trabalho a ser realizado como extensão deste, consiste no desenvolvimento de um Ambiente para Processamento de Conhecimento. Em essência, um ambiente desta natureza terá de compreender, entre outras, as seguintes ferramentas:

- Editor de Ontologias: para permitir a construção, navegação e extensão de ontologias. Uma ferramenta desta natureza deve gerar os axiomas epistemológicos automaticamente e apoiar o engenheiro de software na elaboração dos axiomas de consolidação e ontológicos. Com base em uma ontologia descrita, este editor deve, ainda, apoiar a criação de instanciações.
- Editor de Modelos de Tarefas: para permitir a construção, compreensão e adaptação dos modelos de tarefa suportados por um Servidor de Conhecimento. Esta ferramenta deve ser capaz de gerar, automaticamente, um *template* de resolvedor de problema, a partir de um modelo de tarefa.
- Ferramenta de Apoio à Construção de Ferramentas Baseadas em Conhecimento: para apoiar engenheiros de software na construção de ferramentas baseadas em conhecimento, utilizando a infra-estrutura do Servidor de Conhecimento.

Os editores de ontologias e de modelos de tarefa devem possuir dois modos de operação: modo construção e modo navegação. No modo construção, tais ferramentas permitiriam o desenvolvimento de ontologias e modelos de tarefa, respectivamente. No modo navegação, permitiriam ao engenheiro de software navegar estes modelos para ter uma compreensão do domínio e dos componentes disponíveis. A ferramenta de apoio à construção de ferramentas baseadas em conhecimento utilizaria ambos os

editores no modo navegação, permitindo a seleção de componentes e sua adaptação, auxiliando o engenheiro de software no processo de composição de uma aplicação baseada no conhecimento do domínio do Servidor de Conhecimento.

Além destas ferramentas, o Ambiente para Processamento de Conhecimento deve tornar disponíveis diferentes formas de representação do conhecimento, permitindo que o engenheiro de software selecione aquela que se mostrar mais adequada para o problema em mãos. Uma característica desejável, também, é prover interoperabilidade entre as formas de representação. Para tal, o uso de uma linguagem intermediária para a descrição do conhecimento, tal como Ontolingua (GRUBER, 1992), pode ser uma boa opção.

Conforme mencionado anteriormente, a abordagem de Servidores de Conhecimento pode ser muito útil para o estudo de ADSs Orientados a Domínio (FISCHER, 1994). De fato, esta linha de pesquisa já está sendo conduzida no contexto do Projeto TABA, na concepção de um Ambiente de Desenvolvimento para Cardiologia (OLIVEIRA et al., 1998).

O uso de ontologias como uma ferramenta para analisar e compreender um domínio de estudo mostrou ser bastante interessante e promissor. Observando o papel desempenhado pelas ontologias na abordagem de Engenharia de Conhecimento utilizada neste trabalho, é possível notar uma forte analogia entre a Engenharia de Ontologias e a Engenharia de Domínio. Esta constatação abre espaço para pesquisas no sentido de buscar uma unificação de ambas as abordagens, incorporando ontologias como uma ferramenta para a Engenharia de Domínio.

Os recentes avanços da tecnologia de informação têm tido um significativo impacto nos Sistemas de Informação (SIs). Em contraste aos SIs atuais, onde cada sistema é construído e utilizado com uma certa independência em relação aos demais sistemas de uma organização, começa a despontar uma demanda por sistemas organizacionais complexos, cuja característica básica é a necessidade de integração da informação (dados e conhecimento).

Muitos dos problemas a serem enfrentados na integração de informação estão relacionados à heterogeneidade estrutural e de implementação (incluindo diferenças em plataformas de hardware, SGBDs, linguagens de representação, ...) e à falta de

uma ontologia comum (o que leva a uma heterogeneidade semântica da informação) (BERGAMASCHI et al., 1998). De fato, a integração de informação é a principal área de aplicação para ontologias. Mesmo se dois sistemas adotarem um mesmo vocabulário, não há garantia de que eles estejam de acordo em relação ao significado de certa informação, a menos que se comprometam com a mesma conceituação. Todo SI possui, pelo menos implicitamente, sua própria ontologia, já que atribui significado aos símbolos usados, de acordo com uma visão particular do mundo. Assim, um importante passo na construção de um SI organizacional consiste em explicitar essa ontologia e utilizá-la como elemento principal para a definição da arquitetura do sistema e para a comunicação entre seus subsistemas, em uma abordagem de Sistemas de Informação dirigidos por ontologias (GUARINO, 1998).

Assim, acreditamos que a próxima geração de Sistemas de Informação de nível organizacional terá como ponto de partida uma arquitetura multi-agente, centrada em ontologias, onde cada subsistema pode ser visto como um agente com uma meta objetiva e a comunicação entre os subsistemas dar-se-á através de protocolos bem-estabelecidos, fundamentados na ontologia da organização.

Esta é uma linha de pesquisa que pretendemos seguir: explorar a utilização de ontologias no desenvolvimento de sistemas de informação complexos. A noção de Servidores de Conhecimento, utilizada no contexto de Ambientes de Desenvolvimento de Software, pode, então, ter suas fronteiras ampliadas para o desenvolvimento de sistemas complexos, usando uma abordagem de múltiplos agentes. De fato, um Ambiente de Desenvolvimento de Software é apenas um tipo de sistema complexo. Um Servidor de Conhecimento pode prover uma infra-estrutura para o desenvolvimento de agentes em um domínio, provendo um vocabulário básico para os agentes e para seus protocolos de comunicação.

Referências Bibliográficas

AGUIAR, T.C., 1992, *Um Sistema Especialista de Suporte à Decisão para Planejamento de Ambientes de Desenvolvimento de Software*. Tese de D.Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

ARAÚJO, M.A., 1998, *O Modelo de Integração de Ferramentas da Estação TABA*. Tese de M.Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

BAILOR, P.D., 1992, "Educating Knowledge-Based Software Engineers". In: *Proceedings of the Seventh Knowledge Based Software Engineering Conference - KBSE'92.*

BARLEY, M., et al., 1997, "The Neutral Representation Project". *Ontological Engineering - Working Notes*, Stanford, California, March.

BAZZANA, G., et al., 1993, "ISO 9126 and ISO 9000: Friends or Foes?". In: *Proceedings of the Software Engineering Standards Symposium*, Brighton, England, August.

BEN-SHAUL, I.Z., KAISER, G.E., 1998, "Federating Process-Centered Environments: The Oz Experience", *Automated Software Engineering,* v. 5, n. 1 (January).

BERGAMASCHI, S., CASTANO, S., VIMERCATI, S.C., 1998, "An Intelligent Approach to Information Integration". In: *Proceedings of the First International Conference on Formal Ontology in Information Systems (FOIS'98)*, Trento, Italy, June.

BERSOFF, E.H., DAVIS, A.M., 1991, "Impacts of Life Cycle Models on Software Configuration Management", *Communications of the ACM*, v. 34, n. 8 (August).

BOEHM, B., 1981, *Software Engineering Economics*, Prentice-Hall.

BOLCER, G.A., 1995, "User Interface Design Assistance For Large-Scale Software Development", *Automated Software Engineering*, v. 2, n. 3 (September).

BOOCH, G., 1994, *Object-Oriented Analysis and Design with Applications*, 2nd edition, Benjamin/Cummings Publishing Company Inc.

BORGO, S., GUARINO, N., et al., 1997, "Using a Large Linguistic Ontology for Internet-Based Retrieval of Object-Oriented Components". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

BORST, P., AKKERMANS, H., 1997, "Engineering ontologies", *Int. J. Human-Computer Studies*, v. 46, pp. 365-406.

BRACHMAN, R., SCHMOLZE, J., 1985, "An Overview of the KL-ONE Knowledge Representation System", *Cognitive Science*, n. 9.

BREUKER, J., 1994, "A Suite of Problem Types". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp.57-88.

BREUKER, J., VAN DE VELDE, W., 1994, *CommonKADS Library for Expertise Modelling*, IOS Press.

BUNGE, M., 1977, *Ontology I: The Furniture of the World*, volume 3 of *Treatise on Basic Philosophy*, D.Reidel, Dordrecht, Holland.

BURG, J.F.M., van de RIET, R.P., 1997, "Truly Intelligent CASE Environments profit from Linguistics". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

CANFORA, G., LUCIA, A., MUNRO, M., 1998, "An integrated environment for reuse reengineering C code", *The Journal of Systems and Software*, n. 42, pp 153-164.

CARNAP, R., 1958, *Introduction to Symbolic Logic and Its Application*, Dover Publications, Inc., New York.

CHANDRASEKARAN, B., et al., 1993; "Task-Structure Analysis for Knowledge Modeling". In: Cuena (ed.), *Knowledge Oriented Software Design*, Elsevier Science Publishers B.V.

CHANDRASEKARAN, B., JOSEPHSON, J.R., 1997, "The Ontology of Tasks and Methods". Ontological *Engineering - Working Notes*, Stanford, California, March.

CHATZOGLOU, P.D., MACAULAY, L.A., 1998, "A Rule-Based Approach to Developing Software Development Prediction Models", *Automated Software Engineering*, v. 5, n. 2 (April), pp. 211-243.

COAD, P., YOURDON, E., 1992, *Análise Baseada em Objetos*, Editora Campus.

COAD, P., YOURDON, E., 1993, *Projeto Baseado em Objetos*, Editora Campus.

CLANCEY, W.J., 1993, "The knowledge level reinterpreted: modelling socio-technical systems", *International Journal of Intelligent Systems*, n. 8.

CYBULSKI, J.L., REED, K., 1992, "A Hypertext Based Software Environment", *IEEE Software*, March.

DAVIS, A.M., BERSOFF, E.H., COMER, E.R., 1988, "A Strategy for Comparing Alternative Software Development Life Cycle Models", *IEEE Transaction on Software Engineering*, v. 14, n. 10 (October).

DAVIS, R., SHROBE, H., SZOLOVITS, P., 1993, "What is a Knowledge Representation?", *AI Magazine*, Spring.

DEMIRÖRS, E., 1997, "A Blackboard Framework for Supporting Teams in Software Development". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

DORLING, A., 1993, "SPICE: Software Process Improvement and Capability dEtermination", *Information and Software Technology*, v.35, n. 6/7 (June/July).

DRUMMOND, C., HOLTE, R., IONESCU, D., 1993, "Accelerating Browsing by Automatically Inferring a User's Search Goal", In: *Proceedings of the 8th Knowledge Based Software Engineering Conference - KBSE'93*, September.

ESA, 1991, "The Software Life Cycle". *ESA Software Engineering Standards*, ESA PSS-05-0, Issue 2, February.

FALBO, R.A., TRAVASSOS, G.H., 1995, "Um Estudo sobre Ambientes de Desenvolvimento de Software com Suporte Baseado em Conhecimento". In: *Anais da XXI Conferência Latino-Americana de Informática (CLEI'95)*, Canela, Rio Grande do Sul, Brasil, Julho.

FALBO, R.A., ROCHA, A.R.C., 1996a, "Requisitos de Ambientes de Desenvolvimento para Suportar Processos de Software". In: *Anais da VII Conferência Internacional de Tecnologia de Software (VII CITS)*, Curitiba, Paraná, Brasil, Junho.

FALBO, R.A., TRAVASSOS, G.H., 1996b, "A Integração de Conhecimento em um Ambiente de Desenvolvimento de Software". In: *Anais do 2do. Congreso Argentino de Ciencias de la Computación*, Universidad Nacional de San Luis, Noviembre.

FALBO, R.A., 1996c, "Automatização do Processo de Desenvolvimento de Software". In: Rocha, A.R.C. e Weber, K.C. (eds.), *Qualidade de Software: Seleção de Textos*, CITS, Dezembro.

FALBO, R.A., TRAVASSOS, G.H., 1997a, "Improving Tool's Integration on Software Engineering Environments Using Objects and Knowledge". In: *Proceedings of the World Multiconference on Systemic, Cybernetics and Informatics / 3th International Conference on Information Systems Analysis and Synthesis (SCI'97/ISAS'97)*, Caracas, Venezuela, July.

FALBO, R.A., MENEZES, C.S., ROCHA, A.R.C., 1997b, "Um Servidor de Conhecimento de Processo". In: *Anais do Workshop de Teses em Engenharia de Software - SBES'97*, Fortaleza, Ceará, Brasil, Outubro.

FALBO, R.A., MENEZES, C.S., ROCHA, A.R.C., 1998a, "Integração de Conhecimento sobre Processos de Software em um Ambiente de Desenvolvimento". In: *Anais da IX Conferência Internacional de Tecnologia de Software (IX CITS)*, Curitiba, Paraná, Brasil, Junho.

FALBO, R.A., MENEZES, C.S., ROCHA, A.R.C., 1998b, "Using Ontologies to Improve Knowledge Integration in Software Engineering Environments". In: *Proceedings of Proceedings of the World Multiconference on Systemic, Cybernetics and Informatics / 4th International Conference on Information Systems Analysis and Synthesis SCI'98/ISAS'98*, Orlando, USA, July.

FALBO, R.A., MENEZES, C.S., ROCHA, A.R.C., 1998c, "A Systematic Approach for Building Ontologies". In: *Proceedings of the IBERAMIA'98*, Lisbon, Portugal, October.

FÉRNANDEZ, M., GÓMEZ-PÉREZ, A., JURISTO, N., 1997, "METHONTOLOGY: From Ontological Art Towards Ontological Engineering". *Ontological Engineering - Working Notes*, Stanford, California, March.

FICKAS, S., SELFRIDGE, P.G., 1994, "Software Engineering and Artificial Intelligence". In: *Proceeedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May.

FINNIE, G.R., WITTIG, G.E., 1997, "A Comparison of Software Effort Estimation Techniques: Using Function Points with Neural Networks, Case-Based Reasoning and Regression Models", *The Journal of Systems and Software*, n. 39, pp 281-289.

FISCHER, G., et al., 1992, "Supporting Software Designers with Integrated Domain-Oriented Design Environments", *IEEE Transactions on Software Engineering*, v. 18, n. 6 (June).

FISCHER, G., 1994, "Domain-Oriented Design Environments", *Automated Software Engineering - The International Journal of Automated Reasoning and Artificial Intelligence in Software Engineering*, v. 1, n. 2 (June), pp 177-203.

FOWLER, M., SCOTT, K., 1997, *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley Object Technology Series.

FOX, M.S., et al., 1993, "A Common-Sense Model of the Enterprise". In: *Proceedings of the 2nd Industrial Engineering Research Conference.*

FRANCH, X., BOTELLA, P., et al., 1997, "ComProLab: A Component Programming Laboratory". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

FREITAS, A.L.C., PRICE, A.M.A., 1998, "Formalização de Heurísticas para o Apoio a Modelagem de Sistemas Orientados a Objetos". In: *Anais do XII Simpósio Brasileiro de Engenharia de Software - SBES'98*, Maringá, Paraná, Brasil, Outubro.

GARG, P.K., et al., 1994, "The SMART Approach for Software Process Engineering". In: *Proceedings. of the 16th International Conference on Software Engineering*, Sorrento, Italy, May.

GOMAA, H., et al., 1996, "A Knowledge-Based Software Engineering for Reusable Software Requirements and Architectures", *Automated Software Engineering*, v. 3, n. 3/4 (August).

GÓMEZ-PÉREZ, A., FERNÁNDEZ, M., VICENTE, A.J., 1996, "Towards a Method to Conceptualize Domain Ontologies", *ECAI'96 - Workshop on Ontological Engineering*, Budapest, August.

GONZÁLEZ, P.A., FERNÁNDEZ, C., 1997, "A Knowledge-based Approach to Support Software Reuse in Object-oriented Libraries". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

GRESSE, C., BRIAND, L.C., 1997, "Requirements for the Knowledge-Based Support of Software Engineering Measurement Plans". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

GRUBER, T.R., 1992, *Ontolingua: A mechanism to support portable ontologies, version 3.0.* Technical Report, Knowledge Systems Laboratory, Stanford University, California.

GRUBER, T.R., 1995, "Towards principles for the design of ontologies used for knowledge sharing", *Int. J. Human-Computer Studies*, v. 43, n. 5/6.

GRÜNINGER, M., FOX, M.S., 1994, *The Design and Evaluation of Ontologies for Enterprise Engineering.* Technical Report, University of Toronto, May.

GRÜNINGER, M., FOX, M.S., 1995, *Methodology for the Design and Evaluation of Ontologies.* Technical Report, University of Toronto.

GUARINO, N., 1993, "The Ontological Level". In: *Proceedings of the 16th Wittgenstein Symposium*, Kirchberg, Austria, August.

GUARINO, N, 1995, "Formal ontology, conceptual analysis and knowledge representation", *Int. Journal of Human-Computer Studies*, v. 43.

GUARINO, N., 1997, "Understanding, building and using ontologies", *Int. Journal Human-Computer Studies*, v. 45, n. 2/3 (Feb/Mar).

GUARINO, N., 1998, "Formal Ontology and Information Systems". In: *Proceedings of the First International Conference on Formal Ontology in Information Systems (FOIS'98)*, Trento, Italy, June.

HEIJST, G., 1995, *The Role of Ontologies in Knowledge Sharing*, Doctoral Thesis, University of Amsterdam.

HEIJST, G., SCHREIBER, A.T., WIELINGA, B.J., 1997a, "Using explicit ontologies in KBS development", *Int.J. Human-Computer Studies*, v. 45, n. 2/3 (Feb/Mar).

HEIJST, G., SCHREIBER, A.T., WIELINGA, B.J., 1997b, "Roles are not classes: a reply to Nicola Guarino", *Int. J. Human-Computer Studies*, v. 45, n. 2/3.

HOBBS, J.R., 1995, "Sketch of an ontology underlying the way we talk about the world", *Int. Journal of Human-Computer Studies*, v. 43.

HUMPHREYS, B.L., LINDBERG, D.A.B., 1993, "The UMLS project: making the conceptual connection between users and the information they need", *Bulletin of the Medical Library Association*, v. 81, n. 2.

HURLEY, W.D., KOVACEVIC, S., 1997, "Intelligent Tutoring Systems for Software Development". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

IBRAHIM, O., BERZINS, V., LUQI, 1997, "A Requirements Evolution Model for Computer Aided Prototyping". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

ISO 9000-3, 1991, *Quality management and quality assurance standards - Part 3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software*, ISO.

JACCHERI, M., CONRADI,R., 1993, "Techniques for Process Model Evolution in EPOS", *IEEE Transactions on Software Engineering*, v. 19, n. 12 (December).

KARP, P.D., 1993, "A Qualitative Biochemistry and Its Application to the Regulation of the Trytophan Operon". In: *Artificial Intelligence and Molecular Biology*, AAAI Press.

KIM, H.M., FOX, M.S., 1994, "Formal Models of Quality and ISO-9000 Compliance: An Information Systems Approach". *American Quality Congress Conference*..

KONTOGIANNIS, K.A., 1995, "Workshop Report: The Two-Day Workshop on Research Issues in the Intersection between Software Engineering and Artificial Intelligence", *Automated Software Engineering*, v. 2, n.1 (Mar).

LENAT, D.B, GUHA, R.V., PITTMAN, K., 1990, "Cyc: toward programs with common sense", *Communications of the ACM*, August.

LISBÔA, M.L.B., 1997, *Arquitetura de Meta-Nível*, Tutorial do XI Simpósio Brasileiro de Engenharia de Software, Fortaleza, Ceará.

LIU, X.F., 1998, "A quantitative approach for assessing the priorities of software quality requirements", *The Journal of Systems and Software*, n. 42, pp 105-113.

MADACHY, R.J., 1995, "Knowledge-Based Risk Assessment and Cost Estimation", *Automated Software Engineering*, v. 2, n. 3 (September).

MEYER, B., 1992, *Eiffel: The Language*, Prentice Hall Object-Oriented Series.

MONTERO, L., SCOTT, C.T., 1997, "Using Knowledge Transformation to Improve the Software Development Process". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

MUSEN, M.A., et al., 1995, "PROTEGE-II: Computer support for development of intelligent systems from libraries of components", In: *Proceedings of MEDINFO'95 - Eighth World Congress on Medical Informatics*.

NECHES, R., et al., 1991, "Enabling Technology for Knowledge Sharing", *AI Magazine*, Fall.

NEWELL, A., 1982, "The Knowledge Level", *Artificial Intelligence*, n. 18.

NGUYEN, M.N., WANG, A.I., CONRADI, R., 1997, "Total Software Process Model Evolution in EPOS: Experience Report", In: *Proceedings of the 19th International Conference on Software Engineering*, Massachusetts, USA, May.

O'LEARY, D.E., 1997, "Impediments in the use of explicit ontologies for KBS development", *Int. J. Human-Computer Studies*, v. 46, n. 2/3.

OLIVEIRA, K., ROCHA, A.R., TRAVASSOS, G.H., MATWIN, S, 1998, "Towards a Domain-Oriented Software Development Environment for Cardiology", *CAiSE'98, 5th Doctoral Consortium*, Pisa, Italy, June.

OSTERTAG, E., et al., 1992, "Computing Similarity in a Reuse Library System: An AI-Based Approach", *ACM Transactions on Software Engineering and Methodology*, v. 1, n. 3 (July).

OSTERWEIL, L., 1987, "Software Processes are Software Too", In: *Proceedings of the 9th International Conference on Software Engineering*, California, USA, March.

OSTERWEIL, L., 1997, "Software Processes are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9", In: *Proceedings of the 19th International Conference on Software Engineering*, Massachusetts, USA, May.

PAULK, M.C., CURTIS, B., CHRISSIS, M.B., 1993, "Capability Maturity Model, Version 1.1", *IEEE Software*, July.

PERRY, D.E., POPOVICH, S.S., 1993, "Inquire: Predicate-Based Use and Reuse". In: *Proceedings of the Eighth Knowledge-Based Software Engineering Conference (KBSE'93)*, Chicago, Illinois, September.

PRESSMAN, R.S., 1997, *Software Engineering: A Practitioner's Approach*, 4th Edition, McGraw-Hill.

ROBBINS, J.E., HILBERT, D.M., REDMILES, D.F., 1998, "Extending Design Environments to Software Architecture Design", *Automated Software Engineering*, v. 5, n. 3 (July), pp. 261-290.

ROCHA, A.R.C., 1987, *Análise e Projeto Estruturado de Sistemas*, Editora Campus, Rio de Janeiro.

ROCHA, A.R.C., AGUIAR, T.C., BLASCHEK, J.R.S, 1987, *Ambientes para Desenvolvimento de Software: Definição de Termos.* Relatório Técnico do Programa de Engenharia de Sistemas e Computação ES-137/87, COPPE/UFRJ.

ROCHA, A.R.C., AGUIAR, T.C., SOUZA, J.M., 1990, "TABA: a heuristic workstation for software development". In: *Proceedings of COMPEURO'90*, Tel Aviv, Israel, May.

ROCHA, A.R.C., WERNER, C.M.L, TRAVASSOS, G.H., WERNECK, V.M.B., XEXÉO, G.B., 1994, *Uma Experiência na Definição do Processo de Desenvolvimento e Avaliação de Software segundo as Normas ISO.* Relatório Técnico do Programa de Engenharia de Sistemas e Computação ES-302/94, COPPE/UFRJ.

ROCHA, A.R.C., WERNER, C.M.L., TRAVASSOS, G.H., WERNECK, V.M., 1996, *Processo de Desenvolvimento de Software Baseado em Reutilização*. Relatório Técnico 1/96, COPPE/UFRJ.

RUSSELL, S., NORVIG, P., 1995, *Artificial Intelligence - A Modern Approach*, Prentice Hall Series in AI.

SCHREIBER, G., WIELINGA, B., BREUKER, J., 1993, *KADS: A Principled Approach to Knowledge-Based System Development*, Academic Press.

SELBY, R.W., et al., 1991, "Metric-Driven Analysis and Feedback Systems for Enabling Empirically Guided Software Development", In: *Proceeedings of the 13th International Conference on Software Engineering*, May.

SHEPPERD, M., SCHOFIELD, C., 1997, "Estimating Software Project Effort Using Analogies", *IEEE Transactions on Software Engineering*, v.23, n.12 (November).

SMITH, H., 1996, "Establishing the Foundations for the Specifications of the Next Generation (Advanced) Air Traffic Management Systems", EUROCONTROL EATMS Architecture Workshop, June.

SOWA, J.F., 1995, "Top-level ontological categories", *International Journal of Human-Computer Studies*, v. 43.

STEELS, L., 1990, "Components of Expertise", *AI Magazine*, Summer.

SUNDIN, U., 1994, "Assignment and Scheduling". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp. 231-264.

SWARTOUT, B., PATIL, R., KNIGHT, K., RUSS, T., 1997, "Toward Distributed Use of Large-Scale Ontologies". *Ontological Engineering - Working Notes*, Stanford, California, March.

TAKAAI, M., TAKEDA, H., NISHIDA, T., 1997, "Distributed Ontology Development Environment for Multi-agent Systems". In: *Ontological Engineering - Working Notes*, Stanford, California, March..

THOMAS, I., NEJMEH, B.A., 1992, "Definitions of Tool Integration for Environments", *IEEE Software*, March.

TOTH, G.A., 1995, "Automated Method for Identifying and Prioritizing Project Risk Factors", *Automated Software Engineering*, v. 2, n. 3 (September).

TRAVASSOS, G.H., 1994, *O Modelo de Integração de Ferramentas da Estação TABA*. Tese de D.Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

TRAVASSOS, G.H., WERNER, C.M.L., et al., 1995, *Extensões ao Modelo TABA*. Relatório Técnico, COPPE/UFRJ.

USCHOLD, M., KING, M., 1995, "Towards a Methodology for Building Ontologies", *Workshop on Basic Ontological Issues in Knowledge Sharing, IJCAI'95*.

VALENTE, A., 1994, "Planning". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp. 213-230.

VALENTE, A., 1995, *Legal Knowledge Engineering - A Modelling Approach*. IOS Press.

VALENTE, A., BREUKER, J., VAN DE VELDE, W., 1994, "The CommonKADS Expertise Modeling Library". In: Breuker, J., Van de Velde, W. (eds), *CommonKADS Library for Expertise Modelling*, IOS Press, pp. 31-56.

VALENTE, A., BREUKER J., 1997, "Ontological Engineering with Principled Core Ontologies". *Ontological Engineering - Working Notes*, Stanford, California, March.

VESSEY, I., et al., 1992, "Evaluation of Vendors Products: CASE Tools as Methodology Companions", *Communications of the ACM*, v. 35, n. 4 (April).

VAN DER VET, P.E., MARS, N.J.I., 1993, "Structured System of Concepts for Storing, Retrieving and Manipulating Chemical Information", *Journal of Chemical Information and Computer Sciences*, n. 33.

WERNECK, V.M.B., 1995, *Ambiente de Desenvolvimento de Sistemas Baseados em Conhecimento*. Tese de D. Sc., Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

WERNER, C.M.L., TRAVASSOS, G.H., ROCHA, A.R.C., WERNECK, V.M., 1996, *Memphis: Um Ambiente para Desenvolvimento de Software Baseado em Reutilização*. Relatório Técnico 3/96, COPPE/UFRJ.

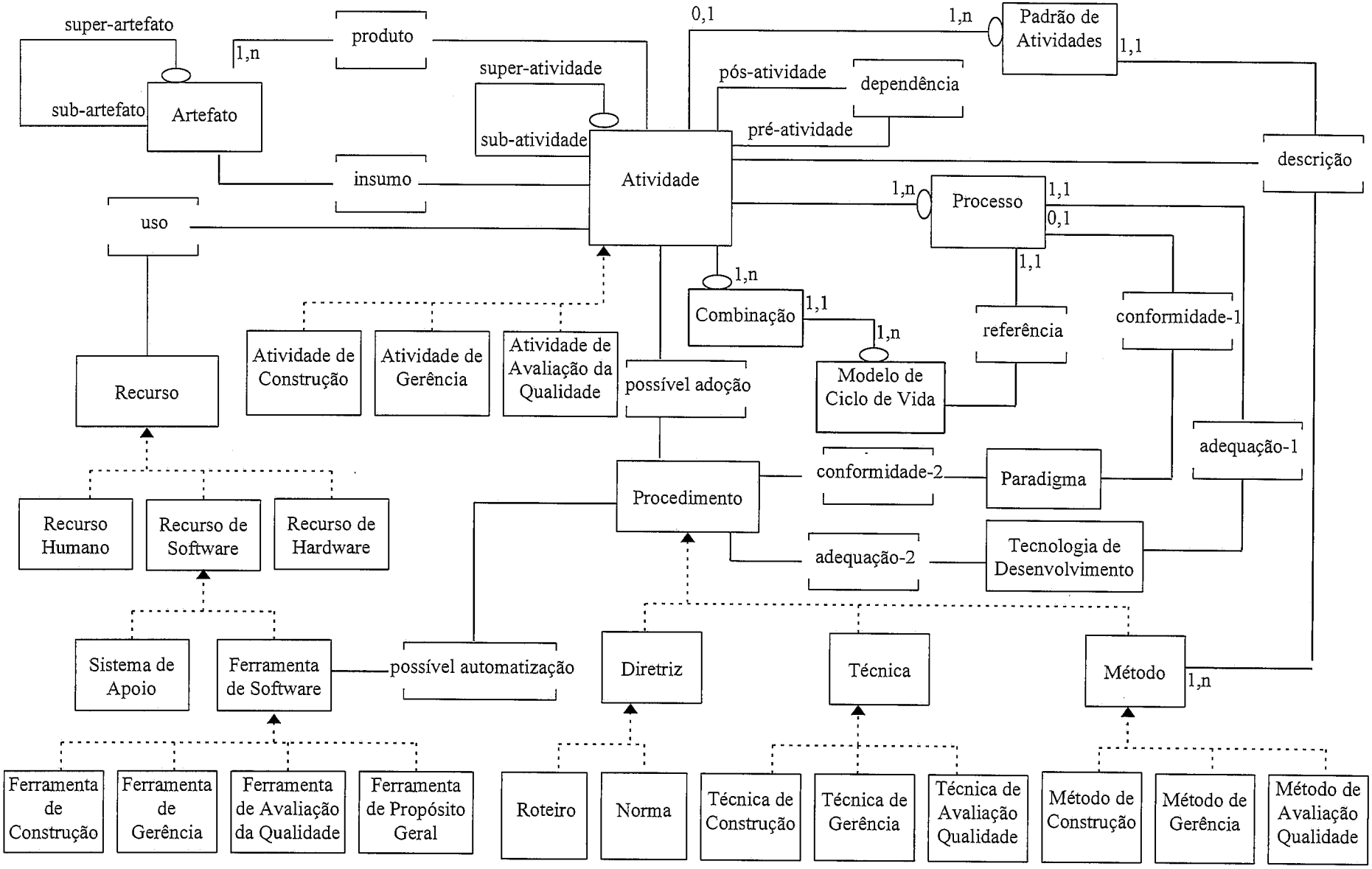
WIELEMAKER, J., 1998, *SWI-Prolog 2.9 - Reference Manual.*

WIELINGA, B., SCHREIBER, G., BREUKER, J., 1992, "KBS Development through Knowledge Modelling". In: Steels and Lepape (eds.), *Enhancing the Knowledge Engineering Process*, Elsevier Science Publishers B.V.

ZHANG, D.A., ZHANG, K., 1997, "Applying Graph Rewriting Rules in Tool Construction and Integration". In: *Proceedings of the 9th International Conference on Software Engineering and Knowledge Engineering - SEKE'97*, Madrid, Spain, June.

Anexo A

Modelo Completo da Ontologia de Processo de Software


super-artefato
sub-artefato
Artefato
1,n
produto
insumo
uso
super-atividade
sub-atividade
Atividade
0,1
1,n
Padrão de Atividades
1,1
pós-atividade
dependência
pré-atividade
descrição
1,n
Processo
1,1
0,1
1,1
referência
conformidade-1
adequação-1
1,n
Combinação
1,1
1,n
Modelo de Ciclo de Vida
possível adoção
Atividade de Construção
Atividade de Gerência
Atividade de Avaliação da Qualidade
Recurso
Recurso Humano
Recurso de Software
Recurso de Hardware
Procedimento
conformidade-2
Paradigma
adequação-2
Tecnologia de Desenvolvimento
Sistema de Apoio
Ferramenta de Software
possível automatização
Diretriz
Técnica
Método
1,n
Ferramenta de Construção
Ferramenta de Gerência
Ferramenta de Avaliação da Qualidade
Ferramenta de Propósito Geral
Roteiro
Norma
Técnica de Construção
Técnica de Gerência
Técnica de Avaliação Qualidade
Método de Construção
Método de Gerência
Método de Avaliação Qualidade

Dicionário de Termos

Adequação-1	relação entre um **processo** e uma **tecnologia de desenvolvimento**, indicando que o **processo** é definido para a **tecnologia de desenvolvimento**.
Adequação-2	relação entre um **procedimento** e uma **tecnologia de desenvolvimento**, indicando que o **procedimento** é adequado ao desenvolvimento usando a **tecnologia de desenvolvimento**.
Artefato	um **insumo** para, ou um **produto** de, uma **atividade**, no sentido de ser um objeto de transformação da atividade. Uma importante propriedade de um artefato é o seu tipo. Os tipos de **artefatos** incluem: **artefatos de código**, **componentes de software** e **documentos**.
Artefato de Código	porção de código-fonte, passível de execução, gerada no próprio desenvolvimento. Ex: programas, sub-programas, classes, frameworks, rotinas, funções, etc.
Atividade	ação que transforma **artefatos** de entrada (**insumos**) em **artefatos** de saída (**produtos**). Em função de sua natureza, atividades podem ser classificadas em **atividades de gerência**, **atividades de construção** e **atividades de avaliação da qualidade**.
Atividade de Avaliação da Qualidade	**atividade** relacionada com a garantia da qualidade do produto em desenvolvimento ou do processo utilizado neste desenvolvimento. Ex: revisões e inspeções de produtos (intermediários ou finais) do desenvolvimento e testes.
Atividade de Construção	**atividade** diretamente relacionada ao processo de construção do software. Ex: especificação de requisitos, modelagem, projeto, codificação, etc.
Atividade de Gerência	**atividade** relacionada ao planejamento e acompanhamento gerencial do projeto. Ex: elaboração, execução, monitoramento, controle e revisão do plano de projeto.
Atividade Elementar	**atividade** que não pode mais ser decomposta, isto é, não é **super-atividade** de nenhuma outra e, portanto, é passível de realização direta.
Combinação	conjunto de **atividades** realizadas segundo uma abordagem seqüencial ou iterativa, que é parte da descrição de um **modelo de ciclo de vida**.
Componente de Software	**artefato** oriundo de uma biblioteca de componentes. Ex: uma classe de uma biblioteca de classes, um framework de reuso, um padrão gerativo, etc.
Conformidade-1	relação entre um **processo** e um **paradigma**, indicando que o **processo** adota o **paradigma**.
Conformidade-2	relação entre um **procedimento** e um **paradigma**, indicando que o **procedimento** segue os princípios do **paradigma**.
Diretriz	**procedimento** que visa estabelecer um padrão para a realização de **atividades**. Diretrizes podem ser divididas em **roteiros** e **normas**.
Documento	**artefato** de software não passível de execução. Ex: documento de especificação de requisitos, plano de projeto, plano de qualidade, relatório de avaliação da qualidade, etc.

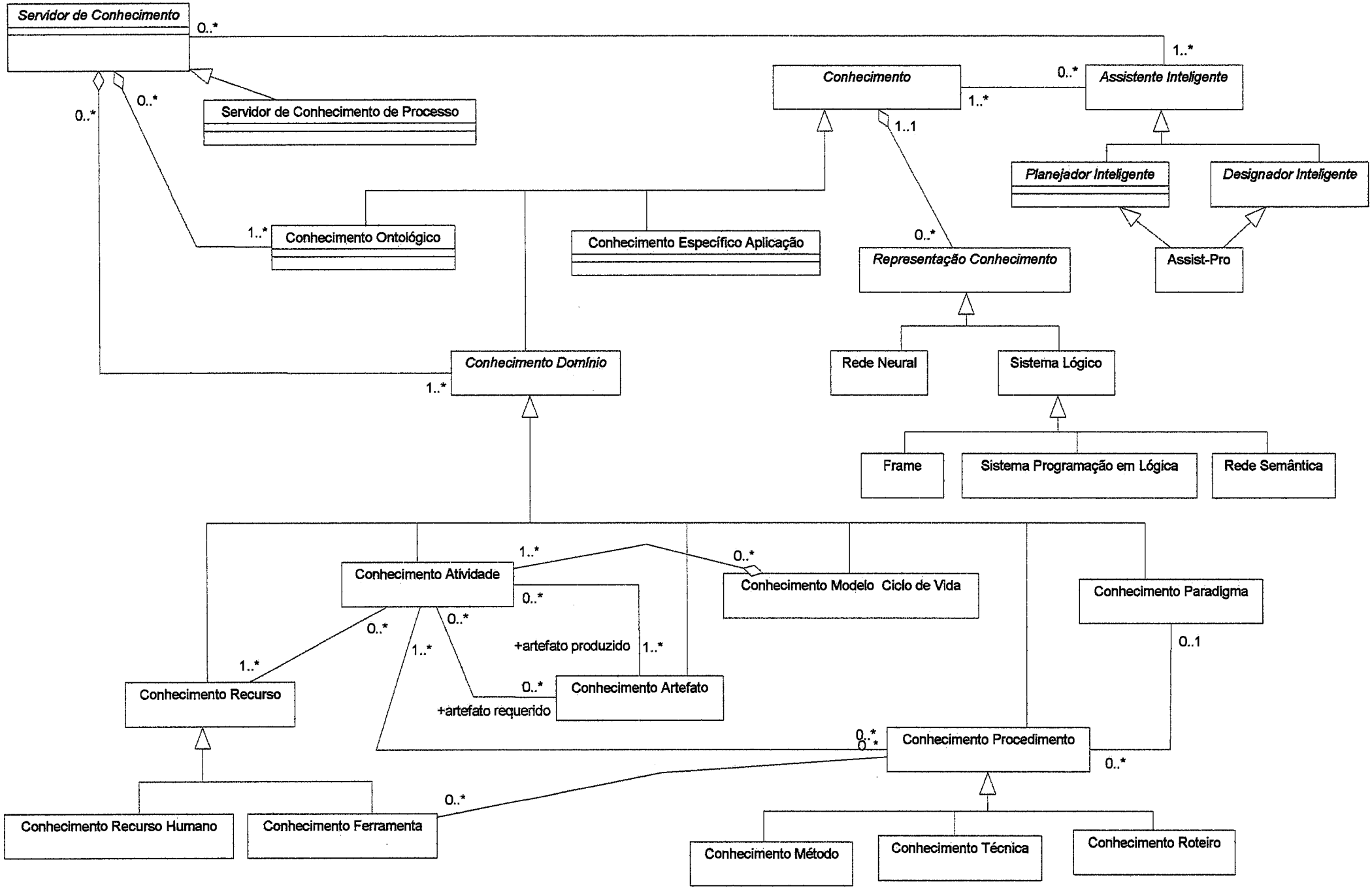
Ferramenta (de Software)	**recurso de software** utilizado para (semi-)automatizar um **procedimento** adotado na realização de uma **atividade**. Quanto ao tipo de **procedimento** que podem (semi-)automatizar, **ferramentas de software** podem ser classificadas em **ferramentas de construção, ferramentas de gerência, ferramentas de avaliação da qualidade** e **ferramentas de propósito geral.**
Ferramenta de Avaliação da Qualidade	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de avaliação da qualidade**. Ex: ferramentas para coleta automática de dados de métricas, geradores de massas de teste, geradores de casos de teste, etc.
Ferramenta de Construção	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de construção**. Ex: ferramentas CASE, linguagens de programação, SGBDs, etc.
Ferramenta de Gerência	**ferramenta de software** que (semi-)automatiza um **método** ou **técnica de gerência**. Ex: sistema de controle de versões, etc.
Ferramenta de Propósito Geral	**ferramenta de software** que não apoia um tipo específico de **procedimento**, podendo ser utilizada para (semi-)automatizar diversos tipos de procedimentos. Ex: editor de texto, editor de figura, editor de formulário, etc.
Insumo	relação entre um **artefato** e uma **atividade**, indicando que o **artefato** é utilizado como "matéria-prima" pela **atividade**, sendo de alguma forma incorporado a outro **artefato**, o **produto** da **atividade**.
Macro-atividade	**atividade** que não é parte de outra, isto é, não é **sub-atividade** de nenhuma outra **atividade**.
Método	**procedimento** *sistemático*, definindo passos e heurísticas para a realização de uma ou mais **atividades**. Quanto ao tipo de atividade que podem apoiar, os métodos podem ser classificados em **métodos de construção, métodos de gerência** e **métodos de avaliação da qualidade.**
Método de Avaliação da Qualidade	**método** para apoiar **atividades de avaliação da qualidade**. Ex: O método Rocha para avaliação da qualidade de produtos de software.
Método de Construção	**método** para apoiar **atividades de construção**. Ex: O método de Booch para desenvolvimento orientado a objetos.
Método de Gerência	**método** para apoiar **atividades de gerência**. Ex: Métodos de estimativa de custo.
Modelo de Ciclo de Vida	estrutura que define as **macro-atividades** de um **processo**, organizadas na forma de **combinações** que são realizadas de forma seqüencial ou iterativa.
Norma	**diretriz** que visa estabelecer padrões para a realização de **atividades** que não sejam de elaboração de documentos. Ex: normas de programação.
Paradigma	filosofia adotada na construção do software, abrangendo um conjunto de princípios e conceitos que norteiam o desenvolvimento. Ex.: paradigma estrutural, orientado a objetos, etc.
Pós-atividade	faceta da relação de dependência entre duas **atividades** a_1 e a_2. Se a realização de a_2 depende da realização de a_1 então a_2 é dita uma pós-atividade de a_1.

Possível Adoção	uma relação entre um **procedimento** e uma **atividade**, indicando que o **procedimento** pode ser adotado na realização da **atividade**.
Possível Automatização	uma relação entre uma **ferramenta de software** e um **procedimento**, indicando que a **ferramenta de software** pode ser usada para (semi-) automatizar o **procedimento**.
Pré-atividade	faceta da relação de dependência entre duas **atividades** a_1 e a_2. Se a_1 precisa ser realizada para que a_2 o seja então, a_1 é dita uma pré-atividade para a_2.
Procedimento	conduta bem estabelecida e ordenada para a realização de uma **atividade**. Quanto à sua natureza, procedimentos podem ser classificados em **métodos**, **técnicas** e **diretrizes**.
Processo	a atividade global de desenvolvimento. É, de fato, uma infra-estrutura contendo as **atividades** das diversas naturezas envolvidas no desenvolvimento de software. A definição de um processo de software depende fortemente da **tecnologia de desenvolvimento** e do **paradigma** a serem adotados no desenvolvimento e pode ser facilitada pela adoção de um **modelo de ciclo de vida** como **referência**.
Produto	relação entre um **artefato** e uma **atividade**, indicando que o **artefato** é produzido pela **atividade**.
Recurso	qualquer coisa que seja necessária para a realização de uma **atividade**, mas que não seja um **insumo** para a **atividade**, no sentido de não ser objeto de transformação por parte da **atividade**. Recursos podem ser classificados em **recursos de hardware**, **recursos de software** e **recursos humanos**.
Recurso de Hardware	equipamento de hardware necessário para a realização de uma **atividade**. Ex: computadores, kit multimídia, etc.
Recurso de Software	software necessário para a realização de uma **atividade**, mas que não é incorporado ao produto desta.
Recurso Humano	agente humano necessário para a realização de uma **atividade**. Ex: engenheiro de software, programador, especialista de domínio, etc.
Referência	relação entre um **processo** e um **modelo de ciclo de vida**, indicando que o **processo** é definido tem por base o **modelo de ciclo de vida**.
Roteiro	**diretriz** para a elaboração de documentos. Ex: roteiro de plano de projeto.
Sistema de Apoio	recurso de software requerido por uma **atividade**, mas que não (semi-) automatiza um **procedimento**. Ex: sistema de gerenciamento de redes.
Sub-artefato	faceta da relação de composição entre dois **artefatos** s_1 e s_2. Se s_2 é parte de s_1 então s_2 é dito um sub-artefato de s_1.
Sub-atividade	faceta da relação de composição entre duas **atividades** a_1 e a_2. Se a_2 é parte de a_1 então a_2 é dita uma sub-atividade de a_1.
Super-artefato	faceta da relação de composição entre dois **artefatos** s_1 e s_2. Se s_1 é decomposto em outros artefatos, dentre eles s_2, então s_1 é dito um super-artefato de s_2.
Super-atividade	faceta da relação de composição entre duas **atividades** a_1 e a_2. Se a_1 é decomposta em outras atividades, dentre elas a_2, então a_1 é dita uma super-atividade de a_2.

Técnica	**procedimento** para a realização de uma **atividade**, que não descreve como realizá-la em termos de **sub-atividades**. Técnicas, em função da natureza das **atividades** que podem apoiar, são classificadas em: **técnicas de construção, técnicas de avaliação da qualidade** e **técnicas de gerência**.
Técnica de Avaliação da Qualidade	**técnica** para avaliar a qualidade do **processo de desenvolvimento** ou dos **artefatos** nele gerados. Ex: inspeções, walkthroughs e testes.
Técnica de Construção	**técnica** para apoiar **atividades de construção**. Ex: técnicas top-down e bottom-up.
Técnica de Gerência	**técnica** para apoiar **atividades de gerência**. Ex: técnicas para estimar tempo e esforço, tal como COCOMO.
Tecnologia de Desenvolvimento	tecnologia a ser empregada no desenvolvimento do software. Ex.: tecnologia convencional de processamento de dados, tecnologia de sistemas baseados em conhecimento, etc.
Uso	uma relação entre um **recurso** e uma **atividade**, indicando que o **recurso** é necessário para a realização da **atividade**.

Anexo B

Modelo de Objetos do Componente Conhecimento


Servidor de Conhecimento
0..*
1..*
Conhecimento
0..*
Assistente Inteligente
1..*
0..*
0..*
Servidor de Conhecimento de Processo
1..1
Planejador Inteligente
Designador Inteligente
1..*
Conhecimento Ontológico
Conhecimento Específico Aplicação
0..*
Representação Conhecimento
Assist-Pro
Conhecimento Domínio
1..*
Rede Neural
Sistema Lógico
Frame
Sistema Programação em Lógica
Rede Semântica
1..*
0..*
Conhecimento Atividade
Conhecimento Modelo Ciclo de Vida
Conhecimento Paradigma
0..*
0..*
0..*
+artefato produzido
1..*
1..*
1..*
0..*
Conhecimento Artefato
Conhecimento Recurso
+artefato requerido
0..*
0..*
Conhecimento Procedimento
0..1
0..*
0..*
Conhecimento Recurso Humano
Conhecimento Ferramenta
Conhecimento Método
Conhecimento Técnica
Conhecimento Roteiro