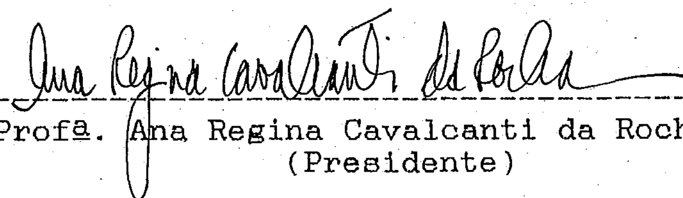
TAXONOMIA DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

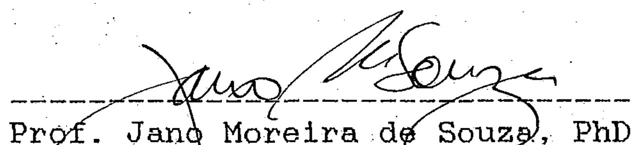
Lygia Maria Ventura Moura

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

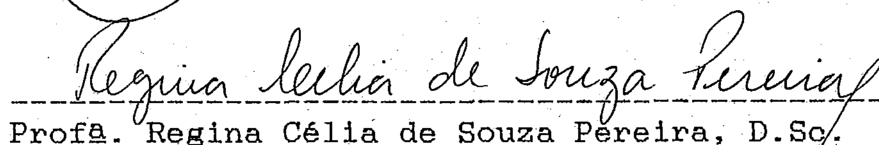
Aprovada por:



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



Prof. Jano Moreira de Souza, PhD



Profª. Regina Célia de Souza Pereira, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MAIO DE 1992

MOURA, LYGIA MARIA VENTURA

Taxonomia de Ambientes de Desenvolvimento de Software
(Rio de Janeiro) 1992.
ix,186p,29,7cm(COPPE/UFRJ, M.Sc. Engenharia de
Sistemas e Computação, 1992).
Tese - Universidade Federal do Rio de Janeiro, COPPE
1. definições 2. classificação de ambientes
3. exemplos de ambientes 4. taxonomia proposta
I. COPPE/UFRJ II. Título (série)

Ao Victor

AGRADECIMENTOS

À professora Ana Regina, por sua orientação segura na grande conquista que foi esse trabalho.

Ao meu filho Victor, por tanto amor.

À Maita, grande amiga, companheira, minha mãe.

Ao meu pai, por estar sempre ao meu lado.

À querida Tagu, pelo seu carinho, compreensão e grande ajuda.

À Aninha, amiga querida, pela confiança.

Ao amigo Jano, sempre presente com seu incentivo.

À minha querida avó, tio Roberto, tia Ignez, tia Lygia, tio Rassi, Léo, Titina, Flavinha e Leonardo pelo apoio constante.

Ao Beto e Edyr, meus padrinhos, meus amigos.

À toda minha familia, pelo estímulo.

À Marimar pela participação amiga, por sua ajuda valiosa.

À Regina Célia pelas observações seguras e todo apoio.

À Maria Angélica e a todos os amigos do Colégio Teresiano, pelo carinho, incentivo e constante colaboração.

Ao Xexéo, Cristina, Marta, Vera, Guilherme, Luis Carlos, Adriana, Emília, Eva, Teresa, Maria Paula e tantos outros amigos que participaram das dificuldades e conquistas desse período.

À Marilza por toda dedicação, ajuda e compreensão nos momentos mais difíceis.

A todos os professores do Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ.

À Denise, Claudia e Ana Paula por terem sido tão amigas e, a todos os funcionários do Programa pela constante colaboração.

À acolhida fraterna de todos os amigos feitos na COPPE. Foram tantos, e tão queridos, que seria difícil citá-los sem correr o risco de esquecer algum.

À CAPES e CNPq que proporcionaram a ajuda financeira para a realização desse trabalho.

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

TAXONOMIA DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

Lygia Maria Ventura Moura

Maio de 1992

Orientadora: Ana Regina Cavalcanti da Rocha

Programa: Engenharia de Sistemas e Computação

Esta tese apresenta o estado da arte em Ambientes de Desenvolvimento de Software.

Após o estudo de alguns ambientes significativos, este trabalho propõe uma Taxonomia para a classificação de Ambientes de Desenvolvimento de Software.

Finalmente, os ambientes estudados e a Estação TABA são classificados segundo a Taxonomia proposta.

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

TAXONOMY OF SOFTWARE DEVELOPMENT ENVIRONMENTS

Lygia Maria Ventura Moura

May, 1992

Thesis Supervisors: Ana Regina Cavalcanti da Rocha

Department: Systems and Computer Engineering

The State of Art on Software Development Environments is presented in this thesis.

Some particularly outstanding environments were studied and a Taxonomy for the classification of Software Development Environments was proposed.

Finally, the environments studied and the TABA Workstation are classified using the proposed Taxonomy.

SUMARIO

TAXONOMIA DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

Capitulo I - Introdução 1

I.1 Motivação 1

I.2 Objetivos da Tese 10

I.3 Divisão em Capitulos 14

Capitulo II - AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE: DEFINIÇÕES E TAXONOMIAS 15

II.1 O que são Ambientes de Desenvolvimento de Software. Definições. 15

II.2 Taxonomias 33

II.2.1 A Classificação Proposta por DART.......... 33

II.2.2 A Classificação Proposta por PERY e KAISER. 39

II.2.3 A Classificação Proposta por LUCENA 48

II.2.4 A Classificação Proposta por PENEDO........ 50

II.3 Meta-Ambientes 62

Capitulo III - ALGUNS EXEMPLOS DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE 67

III.1 INSCAPE 67

III.2 IDeA 80

III.3 ARCADIA 89

III.4 ADAGE 97

III.5 CENTAUR 105

III.6 PEGASE 108

III.7 GRASPIN 113

III.8 OASIS 117

III.9 PATHCAL 126

III.10 ASPIS 131

Capítulo IV - UMA TAXONOMIA PARA AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE 139

IV.1 Histórico e Tendências da Engenharia de Software 139

IV.2 Uma Taxonomia para Ambientes de Desenvolvimento de Software 145

IV.2.1 Classificações 145

IV.2.2 Classificação de Ambientes 146

IV.3 Classificação dos Ambientes no Contexto da Taxonomia Proposta 157

IV.4 Classificação da Estação TABA segundo a Taxonomia Proposta 164

IV.4.1 Descrição da Estação TABA 164

IV.4.2 Classificação da Estação TABA 172

Capitulo V - CONCLUSÃO 178

REFERÊNCIAS BIBLIOGRÁFICAS 180

CAPÍTULO I

INTRODUÇÃO

I.1 - Motivação

O processo de desenvolvimento de software vem evoluindo, rapidamente, ao longo dos anos. Inicialmente, o desenvolvimento de software consistia, quase que exclusivamente, de programas elaborados por programadores, de forma individual e, para si próprios. Tratava-se em sua maioria, de soluções para problemas matemáticos. Observando-se a evolução da computação, verifica-se a completa mudança deste cenário.

A partir da metade da década de 60, os problemas começaram a crescer em tamanho e complexidade, fazendo com que ao longo do processo de desenvolvimento de software, fossem observadas inúmeras distorções causadas pelo uso de práticas anteriores, que não conseguiram acompanhar o acentuado crescimento do poder dos computadores. Surgem, então, softwares ineficientes, onde pouca atenção é dada à especificação de requisitos, verificando-se ainda, a ausência de documentação, baixa confiabilidade e difícil manutenção.

A década seguinte foi marcada pela concentração de esforços em busca de uma solução para o que se convencionou chamar de crise de software, que envolvia segundo Lucena (LUCENA 1989) o alto custo do software ao longo do seu

processo de desenvolvimento, a diversificação da prática deste desenvolvimento e, a necessidade de uma maior produtividade para atender à demanda crescente em torno dos softwares produzidos.

Neste sentido, surgem novas técnicas, métodos e ferramentas para auxiliar o desenvolvimento de produtos de software confiáveis, uma vez que a adoção de um ambiente adequado a este desenvolvimento, envolve problemas relacionados com o tamanho dos sistemas produzidos e à sua crescente complexidade. Assim, como cita Blaschek (BLASCHEK 1987), "o desenvolvimento de um software é um processo extenso e complexo. Reveste-se de problemas ligados à natureza humana, à evolução rápida da ciência da computação e à dificuldade de elaboração de um modelo claro e concreto do que se vai produzir".

Justifica ainda que a não observância de um tratamento adequado a ambientes com tal nível de complexidade, certamente implicarão em produtos de software com "resultados catastróficos", em termos de especificação de requisitos, custos de manutenção e cronograma.

Um ambiente ideal deve fornecer suporte para construção do produto e controle da qualidade ao longo de todas as fases do ciclo de vida, tornando a atividade de

controle do produto inerente e simultânea ao processo de desenvolvimento.

Um outro fator importante a ser considerado durante o processo de desenvolvimento é o gerenciamento. Este deve ser tratado de forma criteriosa, uma vez que indivíduos estão envolvidos e se relacionam durante o desenvolvimento de um software.

Consideramos fundamental observar, que o potencial humano envolvido no processo, deve ser tratado com a mesma atenção dedicada ao potencial dos ambientes/máquinas disponíveis para o desenvolvimento de produtos de software.

Segundo Charette (CHARETTE 1986) "gerenciamento é uma atividade que concerne tanto ao processo de desenvolvimento de um produto, quanto ao próprio produto. O gerenciamento do processo é direcionado para dentro, controlando como os recursos estão sendo usados para desenvolver o produto e também direcionado para fora, no que concerne à integridade do produto e, como os usuários e clientes a percebem".

Cita ainda que o gerenciamento das pessoas envolvidas no processo é, de todas, a mais importante e difícil tarefa, uma vez que por mais intenso que seja o controle, o impacto do sucesso ou falha de um produto de software recairá sobre o fato de que, pela própria natureza das

coisas no mundo real, a probabilidade de que algo esteja errado é mais alta do que a probabilidade de que esteja certo.

Menciona como exemplo de impacto adverso no desenvolvimento de um produto, as seguintes decisões de gerenciamento:

- designar muito trabalho para um grupo e não o bastante para outro;
- designar a pessoa errada para uma determinada tarefa;
- deixar de ouvir o pessoal técnico quanto a resultados técnicos;
- escolher a estrutura organizacional inadequada para a tarefa em mãos;
- não entender completamente o produto a ser gerado.

A importância dada ao aspecto do gerenciamento deve-se à complexidade da tarefa na qual existe uma significativa e complexa rede de inter-relações e interações. O gerente ideal é aquele que pode entender e balancear bases tecnológicas, econômicas e sociais ao longo do desenvolvimento de software, tarefa relevante e fundamental.

Outro aspecto importante a ser considerado nesse cenário, diz respeito à Ergonomia.

Introduzida por Spier (SPIER et alli 1981), a expressão "*ergonomia da engenharia de dados*" visava descrever a disciplina relacionada com a qualidade das ferramentas disponíveis em um ambiente e, a possibilidade de se medir o efeito dessas, na produtividade dos programadores.

Wassermann (WASSERMAN 1988) justifica a relevância dos conceitos relacionados à ergonomia, observando que o termo ambiente, em seu aspecto geral, é definido como "um conjunto de coisas que circundam condições e influências que afetam a existência ou o desenvolvimento de alguém ou de alguma coisa". Assim, mudanças em um ambiente podem acarretar melhorias significativas ou efeitos drásticos.

Propõe, então, por analogia ao conceito de ecologia, que trata do relacionamento dos seres com seu ambiente, que sejam observados os fatores que afetam os programadores e engenheiros de software, ao serem efetuadas modificações em seu ambiente de desenvolvimento, dentre os quais:

- o próprio sistema de computação;
- a disponibilidade do sistema;
- o pessoal de suporte ao sistema;
- as linguagens disponíveis e suas facilidades;
- facilidades na preparação de textos e programas;
- práticas de gerenciamento de projeto;

- normas externas (uso benéfico do espaço físico e proibições de hábitos que agridam o equipamento);

Afirma que a produtividade, como uma medida do trabalho que pode ser desempenhado pelo programador, não deve ser um conceito rigido, uma vez que esta pode ser afetada na medida em que um ambiente cresce em eficiência. Isto ocorre através dos métodos utilizados, das ferramentas disponíveis e construídas para atender às necessidades identificadas.

A preocupação com a adequabilidade do espaço físico e com a escolha do equipamento, evidenciam, segundo o autor, aspectos ecológicos relacionados a ambientes de desenvolvimento de software.

Détienne (DETIENNE 1989) introduz um novo campo de pesquisa chamado "Psicologia da Programação" ou "Ergonomia Cognitiva da Programação".

A atividade de programação, conduzida inicialmente por informatas na década de 70 despertou, posteriormente, a atenção de psicólogos que "reconheceram aí um terreno para estudar as atividades de concepção, compreensão e resolução de problemas pelo especialista, bem como o meio para desenvolver e avaliar as ferramentas de auxílio a essas atividades".

Pesquisas precedentes tinham como objetivo principal a avaliação das ferramentas em termos de desempenho (SCHNEIDERMAN 1980).

Nesse novo campo de pesquisa, os psicólogos se fixaram na análise da atividade da programação e na construção de modelos dessa análise, ou seja, modelos das atividades cognitivas iniciadas ao longo da utilização de linguagens de programação e dos dispositivos informáticos. Os modelos dessa análise tendem a formalizar os conhecimentos do programador, assim como os mecanismos de utilização desses conhecimentos, ao longo da realização das tarefas de programação.

Pesquisas sobre compreensão de programas e da programação se referem à Teoria dos Esquemas. Esta é uma ferramenta de descrição do conhecimento dos especialistas em programação. É uma teoria sobre a organização do conhecimento na memória e sobre os processos de elaboração desses conhecimentos. Segundo essa teoria, os conhecimentos armazenados na memória são organizados sob a forma de estrutura de dados chamados Esquemas. Este conceito foi desenvolvido em Inteligência Artificial e nos estudos psicológicos sobre a compreensão de textos.

A atividade de Concepção de Programas apresenta como particularidades, a constatação de que os problemas a

resolver são mal definidos (inexistência de especificações), e a existência de várias soluções aceitáveis para o mesmo problema, as quais não podem ser julgadas satisfatórias segundo um critério único. Foi ainda observado que o programador dispõe de dois modelos mentais: um modelo do problema e da solução nos termos do domínio da aplicação e um modelo informático da solução.

A aprendizagem de uma linguagem de programação consiste em aprender além da sintaxe, também as regras de funcionamento do dispositivo subjacente à linguagem de programação. Assim, um iniciante costuma se valer dos mecanismos de aprendizagem por analogia à soluções já conhecidas. Utiliza uma estratégia ascendente, partindo de soluções detalhadas para construir um programa.

A atividade de compreensão envolve aspectos de acertos em programas, tanto por programadores que os conceberam em outras épocas, como por aqueles encarregados dos mesmos acertos, em programas escritos por outros. Está aí ressaltada a importância da atividade de manutenção e, segundo a autora, "compreender um programa consiste, em parte, em evocar conhecimentos na memória".

Afirma, ainda, que as estruturas são ativadas de forma ascendente ou descendente, de acordo com a familiaridade de quem as trata, com o problema em questão.

Em relação aos aspectos metodológicos, os estudos mais recentes situam-se ao longo da Psicologia Cognitiva sobre a resolução de problemas e o tratamento da informação. Foi efetuada uma análise detalhada da atividade de programação, procurando levar em conta os aspectos dinâmicos ligados a seu desenvolvimento.

Um aspecto importante observado foi a constatação de uma grande distância entre o que as pessoas dizem de suas atividades e aquilo que elas realmente fazem ao longo destas. Isto implica em dizer que, ao discorrerem sobre suas atividades em situações de lazer, os indivíduos tendem a expressar os aspectos normativos destas e, não, o que eles realmente fazem durante suas tarefas.

Através das técnicas de observação e de protocolos verbais, foi possível obter vários dados sobre as pessoas envolvidas e suas dificuldades, que auxiliaram na determinação de diversas direções para a pesquisa.

I.2 - Objetivos da tese

Partindo da constatação de que "as características de uma determinada aplicação devem determinar as características de seu ambiente de desenvolvimento", isto é, que não se pode ter um ambiente único adequado a qualquer projeto (ROCHA 87), iniciou-se na COPPE/UFRJ o desenvolvimento de um projeto de pesquisa que objetiva prover ambientes de desenvolvimento de software adequados a diferentes domínios de aplicação, através da construção de uma estação de trabalho configurável para desenvolvimento de software: o Projeto TABA.

A escolha do nome TABA, que em língua tupi significa aldeia indígena, foi feita por analogia entre uma taba indígena com várias ocas onde ocorre toda a vida da comunidade e a Estação TABA, composta por um conjunto de recursos que suportam todo o processo de desenvolvimento e execução do software.

A Estação TABA pretende:

a) auxiliar o engenheiro de software na especificação e criação do ambiente mais adequado ao desenvolvimento de um produto específico;

b) auxiliar o engenheiro de software a implementar as ferramentas necessárias ao ambiente definido em (a).

c) permitir aos desenvolvedores do produto (software) o uso da estação através do ambiente especificado em (a) e gerado em (b);

d) permitir a execução do software, dado que, eventualmente, o software produto poderá ser executado na própria estação para ele configurada (o que é sempre verdade pelo menos na fase de testes).

A partir destas funções, foram determinados os quatro ambientes que compõem a Estação TABA:

a) Ambiente Gerador de Ambientes (Meta-ambiente), cujo objetivo é fornecer o ambiente de desenvolvimento de software adequado para uma determinada aplicação. Esta função é realizada pelo ambiente gerador de ambientes (meta-ambiente), que deve especificar o ambiente de desenvolvimento de software mais adequado para o desenvolvimento de uma determinada aplicação e torná-lo operacional.

b) Ambiente Gerador de Ferramentas, que irá auxiliar na construção de ferramentas e que, uma vez implementadas, passarão a fazer parte da Estação.

c) **Ambiente de Desenvolvimento de Software**, é o ambiente específico a ser utilizado no desenvolvimento de uma determinada aplicação.

d) **Ambiente de Teste e/ou Execução da Aplicação**, que permitirá a execução do software desenvolvido na própria Estação.

Esta tese situa-se no contexto do projeto TABA. Seu objetivo é realizar um estudo avaliativo sobre os ambientes de desenvolvimento de software presentes na literatura técnica atual fornecendo, assim, subsídios para o projeto e outros trabalhos relacionados ao mesmo, especialmente:

- **UMBOÉ**[1], o ambiente de desenvolvimento de software educacional (STAHL 1991);

- **TABA-HEP**, o ambiente para desenvolvimento de software para Física de Altas Energias (SOUZA 1991);

- **TABA-OBJ**, o ambiente para desenvolvimento de software orientado a objetos em sua segunda versão (MATTOSO 1990);

- o ambiente para desenvolvimento de sistemas baseados em conhecimento;

- **XAMÃ**[2], o planejador de ambientes da Estação TABA (AGUIAR 92).

1 **UMBOÉ**, significa "ensinar" em língua Tupi
2 **XAMÃ**, significa "pajé", "especialista" em língua ianomani

I.3 - Divisão em Capítulos

Este trabalho está organizado da seguinte forma:

No capítulo II estão descritos os principais conceitos, enfoques e classificações sobre Ambientes de Desenvolvimento de Software presentes na literatura técnica atual.

O capítulo III apresenta exemplos de alguns ambientes propostos na literatura, visando obter subsídios que nos permita elaborar uma classificação, observando suas características mais marcantes.

O capítulo IV apresenta os resultados da experiência adquirida nos capítulos anteriores. Na seção IV.1 é apresentado um histórico e as tendências da Engenharia de Software. Em IV.2 propomos uma Taxonomia de Ambientes de Desenvolvimento de Software considerando os aspectos de arquitetura, do suporte oferecido ao usuário, da escala e das características gerais de Ambientes de Desenvolvimento de Software. A seguir classificamos os ambientes descritos no capítulo III, dentro desse enfoque e, por fim, descrevemos a Estação TABA, situando-a no contexto da taxonomia proposta.

O capítulo V apresenta as conclusões e sugere novas direções de pesquisa.

CAPÍTULO II

AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE : DEFINIÇÕES E TAXONOMIAS

Este capítulo tem como objetivo descrever os principais conceitos, classificações e enfoques sobre Ambientes de Desenvolvimento de Software, presentes na literatura técnica atual.

A seção II.1 apresenta definições e conceitos básicos relativos a Ambientes de Desenvolvimento de Software .

A seção II.2 apresenta as principais classificações e enfoques de forma a permitir uma visão das tendências tecnológicas atuais.

II.1 - O que são Ambientes de Desenvolvimento de Software. Definições.

Nesta seção apresentamos algumas definições e conceitos descritos na literatura técnica atual, procurando chegar a uma noção de consenso em torno do assunto.

Ambiente de Desenvolvimento de Software, como definido em (ILLINGWORTH 1983), é um sistema de computação que provê suporte para o desenvolvimento, reparo e melhorias de programas, e para o gerenciamento e controle destas

atividades. Um sistema típico contém uma base de dados central e um conjunto de ferramentas de software. A base de dados central atua como um repositório para todas as informações relacionadas ao projeto ao longo de seu ciclo de vida. As ferramentas de software oferecem suporte para as várias atividades técnicas e gerenciais, que devem ser realizadas no projeto.

Diferentes ambientes variam quanto à natureza geral de suas bases de dados e à cobertura provida pelo conjunto de ferramentas. Em particular, alguns encorajam (ou ainda reforçam) um método específico de engenharia de software; enquanto outros provém somente suporte geral e permitem que seja adotado alguns dentre uma variedade de métodos.

Todos os ambientes, sobretudo as propostas mais recentes, refletem interesse em apoiar todo o ciclo de vida do software e em oferecer suporte para o gerenciamento de projetos. Estes dois aspectos são os que, normalmente, diferenciam um ambiente de desenvolvimento de software de um ambiente de desenvolvimento de programas.

Segundo Hoffnagle e Beregi (HOFFNAGLE 1985) os ambientes de desenvolvimento de software tem, muitas vezes se caracterizado pela proliferação de métodos e ferramentas desconexos. Isto vem apontar a necessidade da integração de ferramentas nos ambientes desenvolvidos e ainda, a

garantia de portatilidade, uma vez que "o custo de adaptação de uma ferramenta de um ambiente para outro é muitas vezes proibitivo, o que minimiza seu uso e efetividade".

Estes autores reconhecem como principal desafio do desenvolvimento de software, o aumento da produtividade sem o comprometimento da qualidade dos produtos gerados.

Identificando qualidade como a ausência de defeitos nos produtos de software, após ter havido a transformação dos requisitos dos usuários em produtos confiáveis e efetivamente adequados e, produtividade, onde se atinge a qualidade a um custo mínimo, os esforços devem estar direcionados desta forma:

- definição de um modelo de ciclo de vida, objetivando o controle da complexidade e dos avanços alcançados;

- adoção de métodos para cada fase, onde orienta-se quais os procedimentos adequados para atingir os objetivos;

- ferramentas de suporte ao desenvolvimento, e,

- construção de um ambiente de suporte ao desenvolvimento de software, que permita monitorar e controlar o ciclo de vida.

Charette (CHARETTE 1986) discute que o termo "ambiente" não é muito bem definido, quando usado em conjunto com engenharia de software. Cita alguns conceitos encontrados na literatura, tais como:

- conjunto de técnicas, automatizadas ou não, que auxiliam na produção de software;

- conjunto de métodos completos, rigorosos e robustos e sua total automação.

Na concepção do autor, ambientes de desenvolvimento de software são formados por "um processo, métodos e a automação necessária para produzir um sistema de software", como ilustra a figura 1, onde:

- o **processo** é a base do ambiente. Sua função consiste, basicamente, em descrever as cadeias de eventos necessários para criar um determinado produto de software. É, tipicamente, um submodelo de algo mais amplo, que descreverá todas as atividades necessárias para sua produção total;

- os **métodos** incluem todos aqueles necessários para definir, descrever, abstrair, modificar, refinar e documentar o produto de software, e que são definidos pelo **processo** de desenvolvimento;

- o uso do computador para implementar os **métodos** necessários ao desenvolvimento de um produto de software, é chamado de **automação**.

Assim, um ambiente de desenvolvimento de software é composto de cada um desses elementos. O grau pelo qual cada camada está integrada com aquela acima ou abaixo, irá determinar a classe ou o tipo do ambiente.

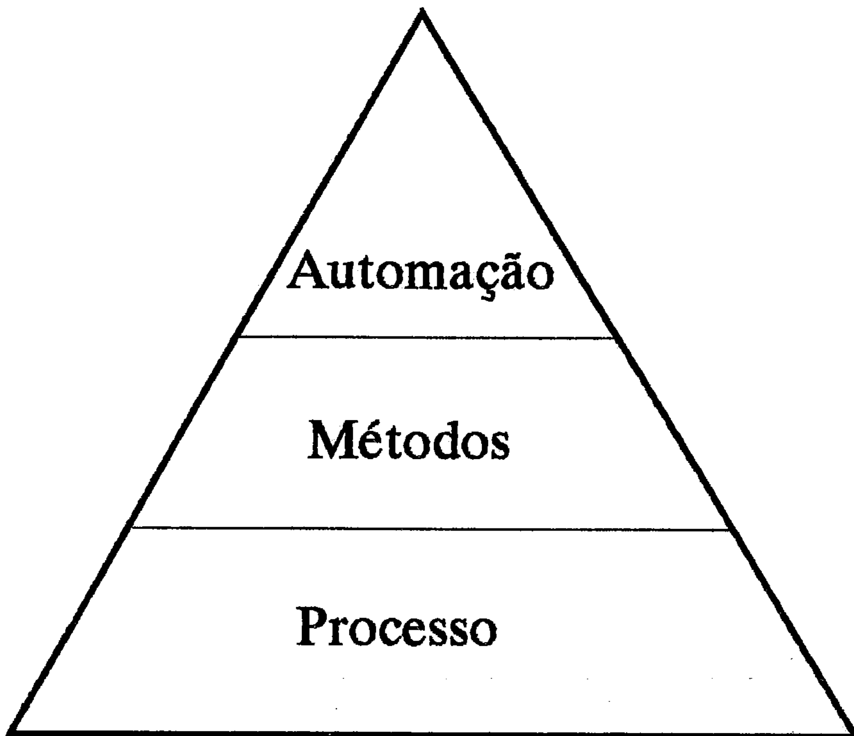


Figura 1: Modelo de Ambiente de Engenharia de Software
Fonte: Charette 86

Lucena (LUCENA 1987) define ambiente de desenvolvimento de software como "uma coleção coordenada de ferramentas de software organizada para dar apoio a algum enfoque para o desenvolvimento de software em conformidade com algum modelo do processo de software".

Em Dart (DART 1987) encontramos que " ambiente de desenvolvimento de software significa um ambiente que automatiza todas as atividades compreendidas no ciclo de desenvolvimento de software, incluindo tarefas de programação em larga escala, tais como gerenciamento de configuração e tarefas de programação em pequena escala tais como gerenciamento do projeto e da equipe participante. Oferece, ainda, suporte para a atividade de manutenção.

Giavitto et alli (GIAVITTO 1989) entendem ambientes de desenvolvimento de software como "o resultado da integração das diferentes ferramentas disponiveis para cada uma das fases do ciclo de vida, permitindo manipular de forma coerente, dados heterogêneos em um mesmo ambiente".

Para Saito (SAITO 1989) um ambiente de engenharia de software é um sistema que fornece suporte a qualquer tipo de atividade executada por engenheiros de software. Afirma que a seqüência destas atividades é chamada de processo de software . Sugere uma primeira abordagem para ambientes de

engenharia de software, como um conjunto de ferramentas que são usadas em diferentes situações do processo de software, indicando, entretanto, que ambientes deste tipo devem ser construídos com alto grau de elaboração e refinamento.

Takahashi (TAKAHASHI 1989), coloca que " Ambientes de Desenvolvimento de Software denotam sistemas de suporte ao desenvolvimento de software em todas as suas etapas, desde a aquisição de requisitos até a manutenção de um produto final, onde o "software-produto" pode chegar a dimensões de complexidade tais que são imprescindíveis mecanismos para apoiar o desenvolvimento distribuído, envolvendo o concurso de vários projetistas/grupos de forma coordenada e segura. Tal ambiente deve, conseqüentemente, apoiar "programming-in-the-large" e "programming-in-the-many" ", enfatizando o autor, "que a distinção entre esses conceitos, persiste por razões puramente históricas".

Prossegue, citando que, "um ambiente ideal" deve prover facilidades para:

- aplicação sistemática de metodologias de desenvolvimento, suportando técnicas e notações dessas metodologias e liberando o projetista de detalhes clericais de suas tarefas;

- gerência de projeto e controle de configuração de produtos;

- reutilização de conhecimento anterior (de projeto, componentes de código, estruturas de projeto, etc...).).

A nível de implementação, como cita o autor, algumas características hoje perseguidas em vários esforços de pesquisa se resumem na busca de uma arquitetura de ambiente que proveja:

- uma interface generalizada usuário/sistema, baseada em conhecimento;

- possibilidade de partes de programas em várias linguagens se comunicarem de forma natural e transparente;

- separação clara de tarefas (ex.: interação com o usuário vs. processamento interno à aplicação);

- mecanismos de extensão e inclusão de novas ferramentas;

- "inteligência" localizada (em ferramentas) e global (a nível de coordenação geral da operação do ambiente). "

Segundo Lubars (LUBARS 1989), ambientes para projeto de software devem apoiar o usuário de diversas formas,

liberando-o de tarefas tediosas e mecânicas para que se dedique mais às tarefas que exigem criatividade e inteligência. Assim sendo, este autor identifica e dá exemplos de sete aspectos onde este suporte ao projetista é necessário:

1. Suporte para atividades clericais de registro de projetos, com facilidades para criação, modificação e "browsing" de representações gráficas;

2. Suporte para interface, de forma a facilitar a comunicação entre o usuário e o ambiente tais como gráficos, menus;

3. Suporte para análise, para avaliar a qualidade do projeto, verificação e manutenção da consistência e realização de validações;

4. Suporte para teste, para construção de casos de teste, construção rápida de protótipos executáveis e avaliação dos resultados de teste;

5. Suporte organizacional, para acompanhar metas, objetivos e dependências de projeto;

6. Suporte baseado em conhecimento, para fornecer conhecimento especialista em *projeto* para projetistas inexperientes, incluindo suporte para reutilização;

7. Suporte inteligente, para assistir na exploração de novas possibilidades e outras atividades criativas.

Para Stinson (STINSON 1989), o desenvolvimento de software consiste de atividades sintéticas e criativas, que envolvem arte e engenharia. Desta forma, a automação deste processo necessita, obrigatoriamente, da disciplina da engenharia. Afirma que " programas são representações precisas da solução de problemas em um dado domínio; quando o *domínio* é a própria aplicação do software, torna-se necessário definir precisamente os aspectos do desenvolvimento".

Introduz um modelo abstrato de desenvolvimento de software, o **modelo agente - objeto** e, apresenta suas implicações em ambientes de desenvolvimento de software.

Através deste modelo, define: "desenvolvimento de software consiste de <u>operações</u> desempenhadas por <u>agentes</u> nos <u>objetos</u>, de forma a permitir a criação de objetos no estado final, os <u>produtos</u>, onde:

operações - constituem-se de operações de engenharia através de transformação, validação e acesso aos objetos;

agentes - entendido como "algo que age", um agente de desenvolvimento pode ser uma pessoa, várias pessoas, pessoas assistidas por sistemas baseados em software ou sistemas autônomos baseados em software, traduzidos em agentes de engenharia e agentes de gerenciamento.

objetos - são objetos da engenharia de software, tais como ferramentas, dados, especificação de requisitos, código, etc., que combinados com outros tipos de objetos, atingem o estado de objeto - final ou _produto_.

Define a automação da engenharia como " a aplicação de sistemas de hardware e software para operações do agente de engenharia.

Cita que sistemas automatizados tem sido diferenciados em termos de suporte, grau de inteligência e outros critérios. Para entender a automação, é útil diferenciar sistemas automatizados em termos de sua aplicação e combinação de serviços genéricos.

Introduz cinco paradigmas de automação que visam sugerir propostas para o uso de tecnologias aplicadas à engenharia de software:

- *"serve paradigm"* provê recuperação de objetos e associação. Está baseado no acesso a serviços genéricos;

- *"predict"* que provê validação antes da transformação;

- *"constrain"* provê validação durante a transformação;

-*"test"*, provê validação acompanhando a transformação;

- *"compile"* que provê transformação automática baseada em linguagem; além disso, compiladores executam a verificação da validação antes da transformação.

Descreve, então, a engenharia da automação em termos de como os componentes individuais desempenham os serviços descritos e como estes podem ser combinados dentro de um ambiente coeso.

Afirma que requisitos distintos devem ser observados em conformidade com o conceito de ambientes de desenvolvimento de software, para que forneçam suporte ao

desenvolvimento de produtos de alta qualidade. Tem-se, então:

- requisitos organizacionais, como por exemplo, extensibilidade do ambiente;

- requisitos do processo, como por exemplo, integração;

- requisitos de fatores humanos, como por exemplo, facilidade no uso;

- requisitos econômicos, como por exemplo, custo/efetividade.

Isto leva a dois princípios de arquitetura de ambientes, que devem ser **integrados** e **flexíveis**.

A **integração** conduz a um ambiente coeso, onde ferramentas integradas suportam uma estratégia de desenvolvimento comum, através da operação no mesmo ambiente, com particionamento dos dados e consistente interface com o usuário.

De acordo com o conceito de agentes de desenvolvimento, sugerido pelo autor em seu modelo agente-objeto, "ambientes integrados são desejáveis pelas mesmas

razões que é desejável se ter um bom engenheiro: ambos contribuem para a produção de produtos de qualidade".

Assim sendo:

- permitem a inclusão de novos requisitos dos usuários;

- provêm auxílio aos aspectos de produtividade, focalizando a atenção dos agentes de desenvolvimento em problemas reais;

- possibilitam o controle e monitoramento das atividades de engenharia pelos agentes de gerenciamento;

- conduzem para a integração das fases do ciclo de vida e prevêem que transformações associadas a fases adjacentes do ciclo de vida, devem partilhar sintaxe e semântica comuns, possibilitando testá-las.

A **flexibilidade** conduz a um ambiente eficiente, onde não haverá necessidade de construção de novo ambiente a cada mudança ocorrida em função das necessidades do desenvolvimento ou da introdução de novas tecnologias. Possibilita a introdução e/ou retirada de ferramentas, substituições, customização, extensão, etc. O autor cita

o paradigma da fábrica flexível para ambientes de desenvolvimento de software, que prevê um programa base segundo o qual "novas ferramentas podem ser rapidamente introduzidas caso haja necessidade; o ambiente pode ser *referramentado*, para atender às necessidades específicas de novos projetos e incorporar tecnologias emergentes".

Entretanto, estes dois princípios são conflitantes. O autor sugere a adoção de princípios mais rudimentares que possibilitem a integração e flexibilidade. São eles:

- **modularidade**, onde o uso de módulos com funcionalidade bem definida, auxilia a gerenciar a complexidade;

- **arquitetura aberta**, isto é, descreve um sistema cuja estrutura é pública, com interfaces providas pelo sistema e, os objetos manipulados são documentados e acessáveis por outros sistemas;

- **"standards"**, quando as interfaces do sistema são, além de públicas, partilhadas pela comunidade de usuários;

- **flexibilidade**, atingida quando estão presentes os princípios de modularidade e "standards", conduzindo à **integração**.

Sugere ainda, alguns temas que devem ser discutidos em termos de ambientes de desenvolvimento de software:

a) **métodos formais x não-formais** onde:

- os partidários dos métodos formais, indicam a necessidade do uso de um estilo de desenvolvimento rigoroso e matemático.

- os que defendem o uso dos métodos informais, afirmam que as implementações geradas a partir de métodos formais serão corretas, desde que se tenha partido de uma especificação de requisitos também correta. Assim, sugerem enfoques menos formais, tais como prototipagem, que consideram necessários para entender os requisitos dos usuários e acomodar rapidamente as mudanças do mundo real.

A importância das duas abordagens é inquestionável e, a disponibilidade de ferramentas adequadas para ambos os métodos bem como as aplicações a que são destinados, serão determinantes no momento da escolha.

b) **especificação x implementação,** ou seja, a importância da divisão do espaço de desenvolvimento, em especificação de requisitos (o que desenvolver) e a

implementação (como desenvolver) e que ferramentas permitiriam verificar a correção das especificações.

c) **Normas**, ou seja, esforços de padronização que permitissem ambientes de desenvolvimento de software integrados e flexíveis.

No contexto do Projeto TABA (ROCHA 1987) (CRISPIM 1988), ambiente de desenvolvimento de software é um conjunto integrado de métodos e ferramentas que suportam o desenvolvimento de um software nas diferentes etapas de seu ciclo de vida. Assim sendo, um ambiente de desenvolvimento de software é composto de:

- um **ciclo de vida**, que define as fases do processo de desenvolvimento e as atividades a serem realizadas em cada fase.

- um conjunto de **métodos**, usados para organizar o pensamento e o trabalho do usuário ao longo do processo de desenvolvimento. Métodos são compostos de **técnicas** e **instrumentos**. Refletem o conjunto de diretivas para a seleção e aplicação sistemática destas técnicas e instrumentos.

  - **técnicas**, podem ser definidas como um conjunto de principios para a execução de uma tarefa especifica

do processo de desenvolvimento de software. São exemplos de técnicas (FREEMAN 1980): "top-down", "bottom-up", "most-critical-element-first".

- **instrumentos** tornam possivel o uso de métodos e técnicas. Como exemplo de instrumentos temos as linguagens de especificação e projeto.

- um conjunto **ferramentas**, que automatizam os instrumentos.

II.2 Taxonomias

II.2.1 A Classificação Proposta por DART

Dart (DART 1987) afirma que os avanços tecnológicos e a crescente expectativa dos usuários altera, de forma significativa, o enfoque dado aos ambientes de desenvolvimento de software. Em termos de funcionalidade, atingida através das ferramentas disponíveis e de fatores que influenciam perspectivas distintas, cita como fundamental a perspectiva do usuário, considerando sua percepção e interação com um ambiente, determinada pela abrangência do suporte oferecido a todas as fases do ciclo de vida e de uma interface amigável.

Assegura que devido às diferenças nos níveis de conhecimento dos usuários, nos tipos de aplicação e no hardware utilizado, um único ambiente pode não satisfazer às necessidades de todos os usuários. Assim sendo, pretende mostrar, através de uma taxonomia de ambientes, as tendências que causam impactos significativos na área (ferramentas, interface com o usuário e arquiteturas). A taxonomia de DART está distribuída em quatro categorias:

a) **Ambientes centrados em linguagens** são altamente interativos e construídos em torno de uma linguagem de programação, provendo um conjunto de ferramentas adequadas a esta linguagem.

Ambientes centrados em linguagens preocupam-se em cobrir todos os aspectos do processo de desenvolvimento de software usando uma linguagem particular. São, especialmente, adequados para a fase de codificação do ciclo de vida, pois:

- provêm compilação incremental ou técnicas de interpretação para atenuar o impacto de pequenas mudanças no código durante a fase de manutenção;

- provêm reutilizabilidade do código;

- provêm facilidades no desenvolvimento de protótipos.

b) **Ambientes orientados a estruturas** fornecem ao usuário uma ferramenta interativa - um editor dirigido por sintaxe- através da qual os usuários interagem, manipulando as estruturas por ele providas. Este editor caracteriza-se como o componente central do ambiente orientado a estruturas e provê suporte somente à fase de codificação.

Os ambientes orientados a estruturas permitem:

- manipulação direta da estrutura do programa, evitando a necessidade de memorização de detalhes de sintaxe;

- multiplas visões de diferentes representações textuais do programa que podem ser geradas a partir de uma mesma estrutura, possibilitando visualizar estes programas em diferentes níveis de abstração;

- controle incremental das informações semânticas acessíveis aos usuários.

São ambientes onde instâncias do editor de estruturas para diferentes linguagens podem ser geradas e, são bem aceitos como auxiliares no ensino de cursos introdutórios de programação.

c) **Ambientes de conjunto de ferramentas ("toolkits")** são constituídos de um conjunto de ferramentas direcionadas a suportar a fase de codificação do ciclo de vida, possibilitando o controle e gerenciamento do ambiente através do uso destas ferramentas.

A abordagem inicial é feita pela utilização do sistema operacional que, através de suas facilidades, permite a introdução de novas ferramentas ao conjunto já existente, possibilitando suporte ao ambiente através de:

- ferramentas para a fase de codificação, tais como compiladores, editores e depuradores;

- ferramentas para tarefas de desenvolvimento de software em larga-escala, tais como controle de versões e gerenciamento de configuração;

- extensibilidade e portatilidade através do uso de um modelo de dados simples.

A motivação inicial dos ambientes incluídos nesta categoria foi a de prover um ambiente independente de linguagem, que suporte múltiplas linguagens, com ferramentas apropriadas.

Neste tipo de ambiente os usuários devem estabelecer diretrizes de gerenciamento visando assegurar que as ferramentas serão usadas corretamente, uma vez que são pequenas as diretrizes por ele providas.

d) Ambientes baseados em métodos, são ambientes que suportam um método particular para desenvolvimento de software e o gerenciamento do processo de desenvolvimento.

Os métodos podem recair na seguinte classificação:

- métodos de desenvolvimento para fases específicas do ciclo de vida, que incluem métodos para análise de requisitos, especificação do sistema, projeto,

validação e verificação, manutenção, com diferentes graus de formalidade.

- métodos para gerência do processo de desenvolvimento que são aqueles que dão suporte ordenado ao desenvolvimento do software, através de procedimentos para o gerenciamento do produto, visando uma evolução consistente do produto por seus desenvolvedores e através de modelos que organizam e gerenciam pessoas e atividades.

Diferentes graus de formalidade estão previstos:

- métodos semiformais, com descrições textuais e gráficas;

- métodos formais, com um modelo teórico básico, através do qual uma descrição pode ser verificada.

As ferramentas de apoio aos métodos semiformais vem sendo utilizadas maciçamente. Isto foi possivel pela disponibilidade de um grande número de desenvolvedores e das facilidades gráficas dos computadores pessoais e estações de trabalho, no desenvolvimento de ferramentas comerciais.

Estas ferramentas:

- suportam usuários individuais especificando e atualizando projetos interativamente;

- organizam ajudas para o projeto dentro de uma hierarquia e níveis de abstração;

- executam algum controle de consistência.

II.2.2 A Classificação Proposta por PERRY e KAISER

Perry e Kaiser (PERRY 1988) (PERRY 1991) apontam o problema de escala como um dos principais fatores a serem considerados em ambientes de desenvolvimento de software. O número de pessoas envolvidas e, ainda, o tamanho e a complexidade do sistema são fatores determinantes na escolha do ambiente adequado.

Introduzem um modelo geral de ambientes de desenvolvimento de software, constituído de três componentes inter-relacionados - diretrizes, mecanismos e estruturas - assim definidos:

Diretrizes são as regras, linhas gerais e estratégias impostas ao usuário do ambiente durante o processo de desenvolvimento de software;

Mecanismos são as linguagens e ferramentas visíveis e subjacentes, que operam nas estruturas providas pelo ambiente e que implementam, junto com as estruturas, as diretrizes impostas pelo ambiente;

Estruturas são aqueles objetos subjacentes e agregados nos quais os mecanismos operam.

Estes três elementos estão relacionados de tal forma que a escolha de um componente acarreta sérias conseqüências nos outros dois, implicando ainda em rígidas limitações nestes.

O enfoque principal apresentado é uma classificação dos ambientes de desenvolvimento de software do ponto de vista de escala. Apontam o tamanho do projeto, o número de desenvolvedores e a influência destas características, como determinantes das mudanças nos requisitos de um ambiente que suporte o desenvolvimento de sistemas de diferentes tamanhos.

A classificação proposta gira em torno de uma metáfora sociológica que sugere distinções quanto aos problemas de escala "na qual uma família é um conjunto de indivíduos, uma cidade é um conjunto de famílias e um estado é um conjunto de cidades".

A metáfora indica que cada classe incorpora aquelas classes que estão a sua esquerda (figura 2) e, que novas distinções podem ser feitas, por exemplo, para a direita do modelo família, como vizinhança, vilas, etc...

Figura 2: Classes de Ambientes de Desenvolvimento de Software.
Fonte: (PERRY 1988)

As quatro classes de ambientes identificadas são:

a) O Modelo Indivíduo

Este modelo abrange os ambientes que oferecem um conjunto mínimo de ferramentas necessárias para a construção de software, habitualmente chamados de ambientes de programação. São providos por uma única estrutura dividida entre os mecanismos, que são ferramentas para a construção de programas, com diretrizes livres sobre questões metodológicas e rígidas para questões de gerenciamento.

A grande maioria das pesquisas em ambientes e de ambientes comerciais estão incluídos nesta categoria. Como instâncias do modelo indivíduo são discutidos:

- **ambientes de conjunto de ferramentas,** considerado pelos autores como o protótipo original do modelo indivíduo onde os mecanismos se comunicam através de

uma estrutura simples. o sistema de arquivos e as diretrizes organizam as estruturas visando a ordenação da seqüência de desenvolvimento e construção.

- **ambientes interpretadores**, que consistem de um conjunto integrado de ferramentas centradas em torno de um interpretador para uma única linguagem tal como LISP ou SMALLTALK. A linguagem e o ambiente não são separáveis, isto é, a linguagem é a interface com o usuário e o interpretador, a ferramenta que o usuário utiliza para interagir com o ambiente. São ambientes cuja característica principal é a flexibilidade, onde os programadores podem construir seus sistemas da forma que lhes pareça mais agradável.

- **ambientes orientados a linguagens** são uma combinação dos ambientes de conjunto de ferramentas e dos ambientes interpretadores. Por este motivo, herdam características dos ambientes citados acima.

- **ambientes transformacionais** suportam uma linguagem de grande espectro que possibilita uma série de objetos e estruturas de controle do abstrato ao concreto. Os programas são, inicialmente, escritos de forma abstrata e, em seguida, modificados por transformações seqüenciais, de forma eficiente e concreta. Uma única diretriz define o estilo do

ambiente: a abordagem transformacional para a construção de programas.

b) O Modelo Família

Representa os ambientes que, além do conjunto de ferramentas para construção de programas disponíveis no Modelo Indivíduo, suprem facilidades que dêem suporte às interações de um pequeno grupo de programadores, utilizando para este fim, regras ou diretrizes que são necessárias para regular determinadas relações entre programadores.

Por analogia a uma familia, onde apesar de seus membros trabalharem de forma autônoma, preocupam-se em saber que todos os outros o fazem da mesma forma, existe neste modelo uma necessidade em manter interações cordiais entre os programadores, obtidas através de regras que irão regular e evitar as interações críticas entre eles. Há ainda, uma preocupação em não duplicar nem perder esforços, o que certamente ocorreria se não houvesse uma coordenação das atividades simultâneas dos programadores.

Neste modelo a ênfase é dada às estruturas que se tornam mais complexas para coordenar as atividades simultâneas, com diretrizes e mecanismos a elas adaptadas.

São instâncias do modelo Família:

- **ambientes estendidos de conjunto de ferramentas** que são um prolongamento do modelo indivíduo de conjunto de ferramentas com a inclusão da estrutura de controle de versão e mecanismos de controle de configuração.

- **ambientes integrados** extensão do ambiente orientado a linguagens do modelo indivíduo, provê consistência entre os módulos componentes, incrementalmente, através dos mecanismos e, ainda, para dentro dos módulos.

- **ambiente distribuído**, que expande o modelo integrado através do número de máquinas conectadas por uma rede local. Para prevenir falhas na rede são necessárias estruturas adicionais que garantam utilizabilidade e confiabilidade.

- **ambiente de gerenciamento de projetos** que provêm suporte adicional para a coordenação de mudanças através da designação de tarefas para programadores individuais, permitindo subtarefas e atividades.

c) O Modelo Cidade

Quando ocorre um crescimento no projeto envolvendo mais de vinte desenvolvedores, as interações entre estas pessoas crescem em número e complexidade. Isso se dá analogamente ao que ocorre em uma cidade pois, se nela forem aplicadas as mesmas regras que gerenciam as atitudes de uma família, o resultado seria algo próximo do caos. Os autores afirmam que " a liberdade apropriada a pequenos grupos produz anarquia quando permitida no mesmo grau em grupos grandes. É precisamente este problema de escala e complexidade de interações que justifica a introdução do modelo cidade".

O modelo cidade introduz um conjunto de métodos e técnicas de gerenciamento que tentam evitar a anarquia causada pelo uso de um modelo não apropriado para a tarefa que se tem em mãos, quando há um aumento na escala. A ênfase do modelo é dada pela obrigação de cooperação entre os desenvolvedores de grandes sistemas.

Ambientes que implementam o modelo cidade ainda são em número inexpressivo.

d) O Modelo Estado

Girando em torno da metáfora sociológica proposta pelos autores, a analogia aqui considerada é a do estado como uma coleção de cidades sugerindo uma companhia com uma coleção de projetos.

A padronização de uma única linguagem, o estabelecimento de um ambiente uniforme que faça uso de um método comum e um conjunto de modelos para todos os projetos, direciona para o racional e acarreta aumento de produtividade, redução nos custos, incluindo o de aprendizagem, uma vez que os desenvolvedores podem deslocar-se de um projeto para outro sem incorrer em um novo processo de aprendizagem de outro ambiente e ainda, a reutilizabilidade das ferramentas, do código, de projetos e especificações. Trata-se de um modelo onde o interesse pelo conjunto, pelo padrão é dominante.

Uma vez que a pesquisa neste campo ainda é pequena, os autores sugerem a existência de instâncias do modelo, apresentando a seguinte descrição geral:

- prover um modelo genérico com ferramentas auxiliares e estruturas de suporte para o desenvolvimento de software para uso geral da companhia;

- instanciar o modelo para cada projeto, adequando cada instância dinamicamente para as necessidades particulares dos projetos individuais;

- gerenciar as diferenças entre as várias instâncias para suportar o movimento de pessoal entre projetos.

II.2.3 A Classificação Proposta por LUCENA

Lucena (LUCENA 1987) discute a diversificação dos ambientes de desenvolvimento de software atuais, em termos de sua adequação às aplicações a que são destinados.

Questiona o aspecto harmônico entre métodos e ferramentas utilizados, sua integração e a eficiência de seu suporte às diferentes fases do ciclo de vida.

Discute que futuras gerações de ambientes tenderão a "incorporar modelos explícitos e adaptáveis dos processos que eles são capazes de representar". Isto ocorrerá a partir da observância de aspectos gerenciais, suporte tecnológico e incorporação de conhecimento, pontos fundamentais para a almejada automatização total de ambientes de desenvolvimento de software.

Apresenta duas classificações baseadas nos trabalhos de Dart (DART 1987) e Perry (PERRY 1988).

A primeira (a), trata os ambientes de desenvolvimento de software de acordo com as funções proporcionadas ao usuário ; a segunda (b) baseia-se no critério da análise do tipo de projeto ou arquitetura dos ambientes.

a) Classificação baseada nas funções proporcionadas ao usuário:

- ambientes básicos de programação;

- caixas de ferramentas ("toolboxes");

- repositórios de informações;

- ambientes integrados;

b) Classificação baseada na arquitetura dos ambientes:

- ambientes centrados em Sistemas Operacionais;

- ambientes centrados em linguagens;

- ambientes centrados em metodologias;

II.2.4 - A Classificação Proposta por PENEDO

Penedo (PENEDO 1988), através de uma análise de ambientes de desenvolvimento de software, identifica que a tendência inicial destes ambientes, voltada para as fases de projeto detalhado e programação, tem hoje novo enfoque. O escopo dos ambientes atuais, em termos de abrangência do ciclo de vida, bem como o seu desenvolvimento, vem evoluindo de forma significativa, prevendo:

- distinções entre ambientes de propósito geral e aqueles para aplicações específicas;

- suporte à utilização de diferentes métodos;

- suporte a diferentes niveis de competência dos usuários;

- uma interface consistente com o usuário;

- a utilização de um modelo de desenvolvimento de software unificado;

- a possibilidade do uso de múltiplas linguagens de programação, e,

- a utilização de arquiteturas abertas.

Em função destas características e, ainda, considerando a grande variedade dos ambientes desenvolvidos, Penedo propõe que estes sejam avaliados segundo quatro enfoques distintos:

III.2.4.1 – Tecnologia subjacente

Várias tecnologias, incluindo sistemas operacionais, sistemas de gerenciamento da base de dados, sistemas especialistas e orientados a objetos, são usadas como base para a grande maioria dos ambientes existentes.

Nos ambientes mais simples (os **ambientes baseados em sistemas operacionais**) ferramentas são chamadas como programas, dados são organizados e acessados através de um sistema de arquivos e o controle do usuário é efetuado através de uma linguagem de comando. Estes sistemas tem como extensões os ambientes orientados a estruturas e os ambientes centrados em linguagens;

Os ambientes voltados para **sistemas de gerenciamento da base de dados** estão, em sua maioria, baseados em modelos relacionais.

Os ambientes baseados em **tecnologia de sistemas especialistas** usam bases de conhecimento para obter informações sobre o produto em construção e o processo

usado para produzi-lo. Utilizam técnicas de Inteligência Artificial.

Os ambientes desenvolvidos através da **tecnologia de objetos** são, também, uma tendência nas pesquisas em ambientes. Utilizam tecnologias dos sistemas de gerenciamento de base de dados e de sistemas especialistas. Informações são modeladas como relações entre objetos e são mantidas em um sistema global baseado em objetos. A integridade da base de objetos é garantida através de regras declarativas, que expressam a manutenção da consistência, de forma automática.

III.2.4.2 - Características Desejáveis

Um conjunto de características devem estar presentes nos ambientes de desenvolvimento de software, dentre as quais são consideradas pela autora:

Utilidade - provendo o auxílio necessário para que os usuários executem suas tarefas, através da realização de todas as atividades previstas no ciclo de vida;

Usabilidade - permitindo um uso amigável, através de interfaces consistentes, facilidade de uso e de aprendizagem, fornecendo suporte à equipe participante e, prevendo a inclusão de novas práticas;

Adaptabilidade - possibilitando a customização, de forma a identificar as características do projeto, procedimentos e/ou preferências individuais dos usuários, adequando-o conforme ocorram mudanças nos requisitos;

Grau de automação - demonstrando a assistência automatizada fornecida aos usuários, que possibilite cumprir, por exemplo, os prazos de projetos e, ainda, auxílio nas atividades inerentes ao desenvolvimento do produto;

Grau de Integração - partindo da perspectiva do usuário, esta característica aponta para o grau de integração das facilidades providas pelo ambiente.

Valor - trata da relação custo/benefício. Custo está relacionado com fatores tais como, investimentos, aquisição e tempo necessário para adquirir determinados níveis de proficiência. Benefício, avalia o aumento de produtividade decorrente da qualidade crescente do produto, com a relação tempo/custo decrescente, no desenvolvimento da aplicação final.

III.2.4.3 - Suporte para o Desenvolvimento do Produto Final

Compara os ambientes considerando seu escopo visível, isto é, o que é percebido como suporte ao desenvolvimento do produto final. Isto pode demonstrar, ou não, a presença de sua característica de utilidade. Devem estar previstos nestes ambientes:

Domínios da Aplicação Final, ou seja, que tipos de aplicações podem ser criadas ou desenvolvidas pelo ambiente.

Propriedades Analisáveis, isto é, as propriedades que estão dirigidas à aplicação final (ex. funcionalidade, desempenho, manutenibilidade, etc.)

Funções, ou seja, as funções dos usuários que são suportadas pelo ambiente (testar, projetar, gerenciar, treinar, etc.)

Atividades do Ciclo de Vida, ou seja, o suporte às tarefas relacionadas às diferentes fases e às tarefas que perpassam os limites das fases, por exemplo, validação e monitoramento do progresso;

Atividades Técnicas, ou seja, as atividades de manipulação e representação do sistema, suportadas

pelo ambiente (ex. descrição, acesso, composição, decomposição, armazenamento/recuperação).

III.2.4.4 - Estrutura básica da Arquitetura

A arquitetura de um ambiente permite a identificação de seus componentes, suas inter-relações estáticas e interações dinâmicas. Existem várias arquiteturas possíveis para ambientes, quase tantas quantos ambientes disponíveis. Alguns atributos fundamentais, entretanto, emergem e fornecem a base inicial para uma taxonomia de arquiteturas.

Arquiteturas da Máquina Virtual, onde os componentes de um ambiente são organizados em camadas de suporte à implementação, com camadas mais baixas apoiando a implementação das mais altas;

Arquiteturas em Rede, onde os componentes são organizados como uma rede de processos que interagem através de comunicação de mensagens.

Arquiteturas Centradas em Dados, colocam o repositório de informações dos ambientes no núcleo do mesmo e organizam os componentes em termos do fluxo de dados dentro e fora da base de dados implementada (a maior

parte dos ambientes baseados em tecnologia de Banco de Dados tem este tipo de arquitetura).

Arquiteturas Centradas em Controle, colocam o subsistema supervisor interno do ambiente no núcleo e organizam os componentes em termos do fluxo de controle entre eles (a maior parte dos ambientes baseados em tecnologia de Sistemas Especialistas exibe esse tipo de arquitetura).

Ambientes de Desenvolvimento de Software podem exibir apenas um desses atributos, entretanto a realidade mostra que, em geral, exibem uma combinação destes atributos de arquiteturas.

Penedo introduz ainda, uma **arquitetura conceitual de ambientes**. Trata-se de uma arquitetura de máquina virtual com quatro camadas, que provêm facilidades para três funções a serem executadas por diferentes usuários do ambiente: o construtor de ambientes, o adaptador de ambientes e o desenvolvedor do projeto (figura 3). Nesta arquitetura conceitual, após construídos pelos construtores do ambiente, serão gerados os **ambientes específicos para projeto** pelos adaptadores de ambientes, que serão utilizados pelos desenvolvedores do projeto para desenvolver suas aplicações finais.

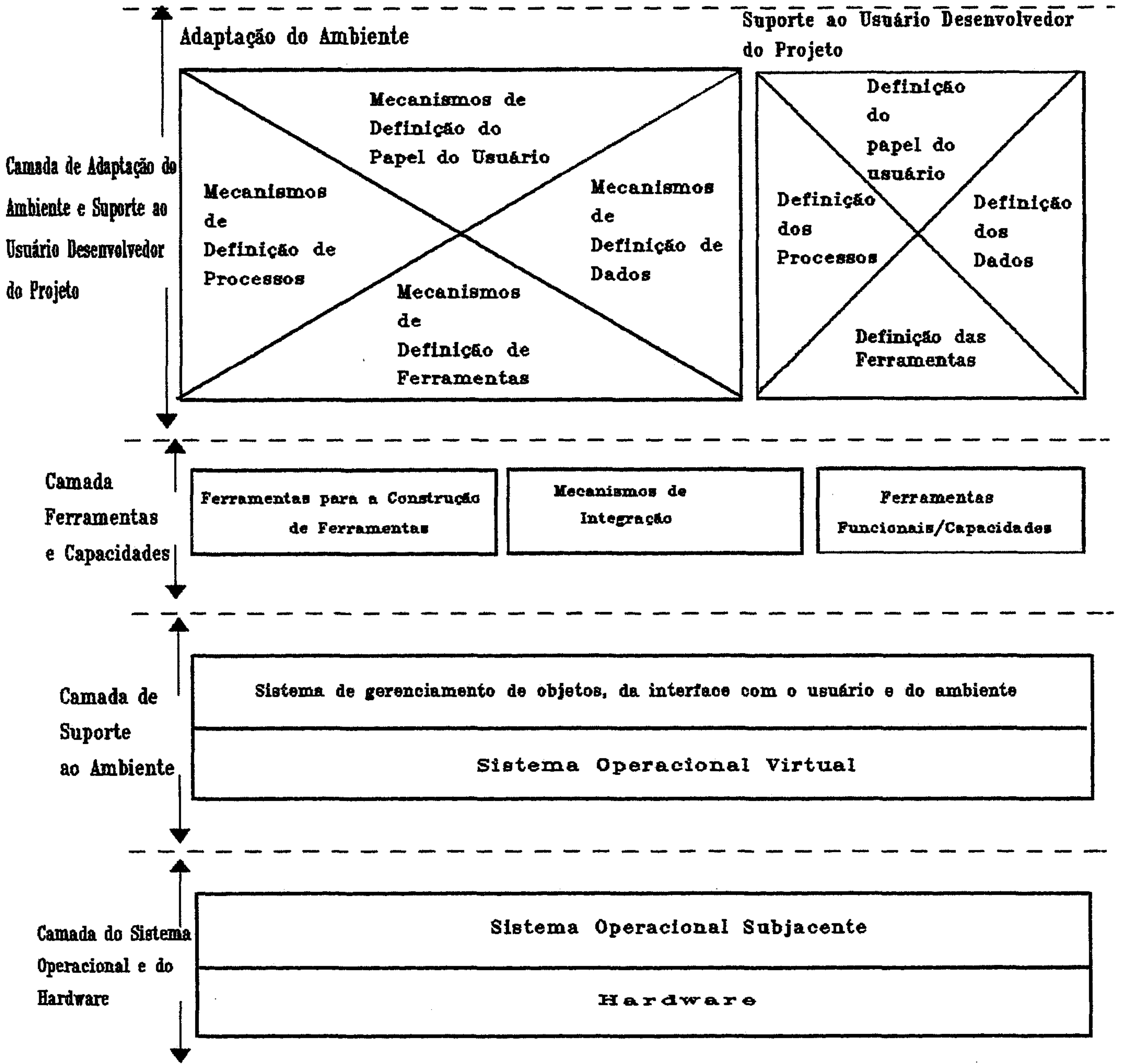


Figura 3: Arquitetura Conceitual de Ambientes
Fonte: Penedo 88

Esta arquitetura apresenta as seguintes camadas:

Camada do Sistema Operacional e do Hardware: consiste do hardware e do sistema operacional subjacentes. Pode compreender uma coleção heterogênea de processadores, estações de trabalho e periféricos e pode estar baseada em um sistema operacional em rede. Esta camada faz parte de qualquer ambiente.

Camada de Suporte ao Ambiente: Esta camada também faz parte de qualquer ambiente específico para projeto. Provê a infra-estrutura sobre a qual o ambiente é construído. Aqui, o principal objetivo é simplificar a construção dos componentes na camada de ferramentas fornecendo um conjunto de facilidades. Um segundo objetivo, também muito importante, é o transporte através de várias configurações de hardware e sistemas operacionais. Seus principais elementos são:

- **um sistema operacional virtual**, provendo a portatilidade, ou seja, o uso do ambiente em diferentes configurações de hardware;

- **um sistema de gerenciamento de objetos**, provendo um modelo de objetos e a definição de objetos, organização e mecanismos de acesso.

Este sistema apoia o armazenamento e recuperação de objetos (dados, processos e ferramentas):

- **um sistema de gerenciamento da interface com o usuário**, provendo mecanismos básicos para definir as interfaces com o usuário (janelas, gráficos, menus, texto, etc.) e associando objetos presentes na interface com objetos do ambiente;

- **um sistema de gerenciamento do ambiente**, provendo facilidades de controle necessárias para mapear requisitos dos usuários para atividades internas.

Camada Ferramentas e Capacidades: Contém os componentes que provêm a funcionalidade do ambiente. É, tipicamente, composta de várias ferramentas ou fragmentos de ferramentas necessárias a qualquer ambiente particular. A tarefa de adaptação de um ambiente consiste, em parte, da seleção de ferramentas ou capacidades desta camada que, então, farão parte de um ambiente especifico para projeto. São componentes desta camada:

- **ferramentas funcionais/capacidades**, que automatizam os vários métodos e técnicas que fornecem suporte ao processo de desenvolvimento

de software e podem fornecer assistência automatizada para as atividades do processo de desenvolvimento de software.

- **ferramentas para construção de ferramentas**, que são utilizadas principalmente pelos construtores de ambientes na preparação de seus componentes para esta camada e suporte à adaptação. Entretanto, adaptadores de ambientes e usuários desenvolvedores de projetos, podem fazer uso dessa camada para construir ferramentas para áreas de aplicações especificas ou processos de software. Usuários desenvolvedores de projetos podem ainda considerá-las úteis para a criação ou evolução de aplicações.

- **mecanismos de integração**, que suportam a combinação das ferramentas, de forma que estas cooperem de maneira harmônica.

Camada de Adaptação do Ambiente e Suporte ao Usuário Desenvolvedor do Projeto: Contém dois conjuntos de capacidades especificas para suporte às necessidades dos adaptadores de ambientes e aos usuários desenvolvedores de projetos:

- **mecanismos de suporte à adaptação.** que também podem ser chamados de ferramentas ou ***capacidades***, e que facilitam a definição e adaptação de ambientes.

- **capacidades de suporte ao usuário desenvolvedor de projeto:** suportam vários tipos de usuários num projeto de desenvolvimento de software (especificador, projetista, gerente do projeto, etc.). Estas capacidades compreendem, o sistema de gerência do ambiente e os mecanismos e ferramentas de integração para adaptar o ambiente às necessidades específicas de um projeto e de seus desenvolvedores. Por exemplo, a informação deve definir as visões que os usuários tem do processo baseadas em dados, experiência, procedimentos baseados em normas de projeto e padrões da organização.

II.2.5 - Meta-Ambientes

Tom de Marco (DE MARCO 1984) ao analisar a adoção de métodos na modelagem de projetos, cita como ideal a definição de um método para cada área de aplicação ou até mesmo para cada projeto. Justifica essa colocação afirmando que diferentes áreas de aplicações tem suas particularidades e que estas devem ser consideradas ao se pretender compreender os requisitos dos usuários.

Sugere, então, a adoção de meta-métodos que auxiliariam na tarefa de elaboração de projetos para diferentes domínios de aplicações, garantindo o aumento da qualidade e produtividade dos produtos a serem construídos.

Diferentes autores abordam este aspecto, de forma análoga, referindo-se a ambientes de desenvolvimento de software.

Sorenson (SORENSON 1988), atribui à diversificação dos sistemas produzidos e às diferentes áreas de aplicação, o aparecimento dos meta-ambientes. Estes, segundo o autor, visam a redução do esforço de implementação, produzindo um ambiente de suporte para este fim, cuja proposta principal é a de gerar, de forma automatizada, as principais partes de um ambiente particular de desenvolvimento de software .

Penedo (PENEDO 1988), ao identificar a necessidade de suporte abrangente ao desenvolvimento do produto final, aproxima-se do conceito de meta-ambientes, uma vez que considera para esse fim diversos fatores, dentre os quais os tipos de aplicações que podem ser criadas ou desenvolvidas pelo ambiente. Para tanto, observa diferentes aspectos de desenvolvimento e introduz uma arquitetura conceitual de ambientes. Esta suporta três diferentes usuários em um ambiente: o construtor de ambientes, o adaptador de ambientes e o desenvolvedor do projeto.

O Modelo Estado de Perry (PERRY 1988), apresenta como principal característica a padronização. Esta, segundo os autores, objetiva, entre outros, a redução do esforço de aprendizagem e o aumento da produtividade. Para tanto é sugerido um ambiente uniforme que atenda as necessidades dos projetos individuais de uma companhia. Identifica-se, então, o conceito de meta-ambientes ao serem considerados aspectos genéricos de desenvolvimento para atender a estas especificações.

Em seu estudo sobre meta-ambientes, Lucena (LUCENA 1987) mostra a existência de inúmeras pesquisas e tentativas de projeto e implementação de meta-ambientes. Isto permitiu ao autor traçar as tendências tecnológicas da

área, propondo uma classificação dos meta-ambientes, em termos de suas arquiteturas:

Meta-ambientes com estrutura puramente gerativa, baseados em um editor dirigido por sintaxe, um compilador incremental e um "linker" para a linguagem do ambiente alvo.

Configuradores de ambientes, baseados no modelo do processo de software e das aplicações onde, por analogia a uma configuração de software, utilizam uma arquitetura que fornece suporte a um modelo genérico do processo de desenvolvimento de software e suas ferramentas.

Caracterizam-se por:

- independência do modelo do processo e das ferramentas, visando flexibilidade e evolução do ambiente gerado;

- portatilidade;

- integração das ferramentas e integração com o usuário, através de um modelo de dados comum e de uma consistente interface com o usuário:

- definição formal do modelo do processo e a automação do seu controle, através de mecanismo adequado.

Meta-ambientes exploratórios geram ambientes no qual o processo de desenvolvimento adquire conotações de aprendizado especializado, para determinada área de aplicação. Para esse fim, o engenheiro de software utiliza métodos selecionados para projeto e programação, dentre aqueles mantidos pelo ambiente. A característica subjacente a esse tipo de desenvolvimento é o cooperativismo, onde especialistas trabalham, em conjunto, no mesmo problema. São meta- ambientes recentes e, encontram-se em fase de pesquisa.

No Projeto TABA (COPPE 1987), meta-ambiente é definido como um ambiente que abriga um conjunto de programas que interage com o usuário específico (meta-usuário) para definir tipos de objeto que comporão um ambiente de desenvolvimento específico.

A motivação para o Projeto TABA, foi a constatação, em diversos projetos de que as características de uma determinada aplicação devem determinar as características de seu ambiente de desenvolvimento e que a definição deste ambiente, vai depender do estado da arte em métodos e ferramentas e do conhecimento de especialistas no uso dos mesmos. Assim sendo, na Estação TABA, partindo das

características de um determinado produto que se pretende desenvolver, é possível determinar o ambiente adequado para seu desenvolvimento, através de XAMÃ, o planejador de ambientes.

As funções da Estação TABA, bem como os ambientes determinados por essas funções, já descritos na seção I.2, serão analisados, em detalhe, no capítulo IV.

CAPÍTULO III

ALGUNS EXEMPLOS DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE.

Este capítulo descreve alguns ambientes de desenvolvimento de software propostos na literatura atual.

Com este estudo pretendemos reunir maiores subsídios para propor, no próximo capítulo, uma classificação para ambientes de desenvolvimento de software, salientando suas características mais importantes.

III.1 - INSCAPE (PERRY 1989)

INSCAPE é um ambiente de desenvolvimento de software, integrado, para construção de grandes sistemas, por grandes grupos de desenvolvedores.

A integração dos componentes do ambiente se dá em dois níveis:

- no nível mais baixo estes partilham uma representação interna comum e uma interface com o usuário, também comum;

- no nível mais alto os componentes são integrados através do uso construtivo de especificações formais da interface entre os módulos.

INSCAPE está voltado para dois problemas fundamentais da construção de sistemas de software: evolução e escala.

Considera que o sistema evolui em todas as suas partes do início ao fim do desenvolvimento e que a tentativa de dividir o processo de desenvolvimento em fases, distintas e independentes, só serve para tornar obscura a realidade do processo de desenvolvimento de software.

A consideração dos problemas de escala, traz consigo, três subproblemas importantes: complexidade, "programming-in-the-large" e "programming-in-the-many". Estes subproblemas estão relacionados às seguintes questões:

- Como auxiliar na gerência dos problemas de complexidade, especialmente quando esta parece crescer exponencialmente ao crescer o tamanho?

- Que facilidades são necessárias para a construção de sistemas cujos componentes são desenvolvidos separadamente?

- O que deve ser acrescentado no ambiente para apoiar a interação entre pessoas e grupos realizando diferentes atividades em um projeto?

III.1.1 - Enfoques Básicos do Projeto

O Projeto INSCAPE adotou três enfoques básicos: a formulação de modelos das várias partes do sistema, a formalização das partes do processo (especificamente os aspectos de construção e evolução) e a exploração de alternativas para tornar os problemas mais manipuláveis.

Para desenvolvimento do projeto:

- foram considerados modelos para interconexão de software e chegou-se à proposta de um novo modelo, com interconexões semânticas, que é a base de INSCAPE;

- desenvolveu-se um modelo de Ambientes de Desenvolvimento de Software (PERRY 1988) identificando os aspectos fundamentais de Ambientes;

- formalizou-se o processo de desenvolvimento de software;

- fez-se uso da tecnologia de especificação e verificação, usando especificações formais da interface de forma construtiva;

- incorporou-se ao Ambiente de Desenvolvimento de Software o entendimento de especificações, processo de desenvolvimento e linguagens de programação.

III.1.2 - Estratégias de Pesquisa:

- trabalhar nas implicações do uso de especificações formais da interface como o ingrediente integrador de um Ambiente de Desenvolvimento de Software que dê suporte ao desenvolvimento de grandes sistemas por muitos desenvolvedores.

- tornar prático o uso de especificações, restringindo seu poder, fazendo verificações de consistência e automatizando ao máximo;

- usar técnicas incrementais, para distribuir o custo de análise.

Assim sendo, o objetivo do projeto é fornecer uma forma prática de utilizar métodos formais no desenvolvimento de projetos grandes, fortemente evolutivos e com grande número de pessoas envolvidas.

III.1.3 - INSTRESS: a linguagem de especificação formal da interface de módulos

O uso de especificações formais para descrever a interface de módulos permite apresentar, de forma clara e precisa, todas as informações sobre um módulo, necessárias

ao programador. Para isso INSTRESS fornece facilidades para:

- especificação de predicados, que são meios de criar o vocabulário necessário para descrever a semântica dos objetos de dados e das operações no módulo, isto é, para expressar a abstração fornecida pelo módulo;

- especificação de objetos de dados e suas propriedades (tipo, constantes e variáveis);

- especificação das operações, que descrevem o comportamento da interface abstrata (externa) de funções e "procedures". Em Instress, o comportamento desta interface é descrito por um conjunto de pré-condições (que podem ser de um dos seguintes tipos: assumida, validada ou dependente) e um conjunto de resultados;

- inclusão de informações pragmáticas, tais como recuperações de exceções;

- ver a especificação sobre diferentes aspectos, por exemplo: uma visão informal dos objetos (mostrando apenas os comentários informais e sinopses) ou uma visão formal (mostrando apenas as definições formais dos objetos);

- análise da especificação, quanto à consistência.

A figura 4 mostra um exemplo de especificação em Instress.

```
PutRecord(<in> filename FN; <in> int R;
          <in> int L; <inout> buffer B);
  {
  fileptr FP;
  if FileExists(FN)
     OpenFile(FN, FP);
        <exception IllegalFileName pruned>
        <exception NonExistentFile precluded>
  else
     CreateFile(FN, FP);
        <exception IllegalFileName pruned>
        <exception FileAlreadyExists precluded>
  WriteRecord(FP, R, L, B);
     exception IllegalRecordNr       <coalesced>
     exception UnallocatedBuffer     <coalesced>
     exception BufferTooSmall        <coalesced>
        CloseFile(FP);
        return ParameterError;       <propagated>
     exception WriteError
        CloseFile(FP);
        return IOerror;              <propagated>
  CloseFile(FP);
  }
Propagated Preconditions:
  LegalFileName(FN)          Allocated(B)
  LegalRecordNr(R)           BufferSizeSufficient(B, L)
  RecordIn(B)
Propagated Results:
  <normal exit>
     Propagated Postconditions:
     LegalFileName(FN)       BufferSizeSufficient(B, L)
     FileExists(FN)          Deallocated(B)
     LegalRecordNr(R)        RecordExists(R)
     Propagated Obligations: <none>
  <exception ParameterError>
     Propagated Postconditions:
     FileExists(FN)
     not IllegalRecordNr(R) or not Allocated(B)
        or not BufferSizeSufficient(B, L)
     Propagated Obligations: <none>
  <exception IOerror>
     Propagated Postconditions:
     LegalFileName(FN)       Deallocated(B)
     FileExists(FN)          BufferSizeSufficient(B, L)
     LegalRecordNr(R)        not RecordWriteable(R)
     Propagated Obligations: <none>
```

Figura 4: Exemplo de Implementação em INSTRESS
Fonte: Perry 89

Assim sendo, através da especificação da semântica do módulo e da inclusão de informações pragmáticas, o projetista pode definir precisamente o significado de um módulo e indicar formas nas quais ele pode ser usado adequadamente.

A figura 5 mostra um exemplo de implementação.

```
***** Module FileManagement *****

Predicates
    LegalFileName(filename F)
        Definition:  NonNullString(F) and
                     each ( i in 1..length(F) ) { Alphabetic(F[i]) }
        Informally:  A legal file name is a non-empty string of
                     alphabetic characters only.

    FileExists(filename F) . . .
    ValidFilePtr(fileptr FP) . . .

    FileOpen(fileptr FP)
        Definition:  primitive
        Informally:  The file is opened for reading and writing.

    FileClosed(fileptr FP)
        Definition:  not FileOpen(FP)
        Informally:  The file is closed for I/O.

    LegalRecordNr(int R) . . .
    RecordExists(int R) . . .
    RecordReadable(int R) . . .
    RecordConsistent(int R) . . .
    RecordWriteable(int R) . . .
    RecordIn(buffer B) . . .
    BufferSizeSufficient(buffer B; int R) . . .

Data Objects
    Type: filename
        Representation: string
        Properties:  each ( filename F ) { LegalFileName(F) }
        Synopsis:    A filename is a non-empty string that is limited to
                     alphabetic characters.

    Type: fileptr . . .

Operations
    int CreateFile(<in> filename FN; <out> fileptr FP)
        Synopsis:  CreateFile creates a file named by FN,
                   automatically opens it, and returns a handle to be
                   used in all subsequent file operations until the file
                   is closed.
        Preconditions:
            LegalFileName(FN)         <validated>
            not FileExists(FN)        <validated>
        Results:
            <Successful result: CreateFile == 0>
                Synopsis: The file named by FN has been created and
                    opened, and the output parameter FP is the
                    handle for subsequent file operations.
                Postconditions:
                    LegalFileName(FN)   ValidFilePtr(FP)
                    FileExists(FN)      FileOpen(FP)
                Obligations: FileClosed(FP)
            <Exception IllegalFilename: CreateFile == 1>
                Synopsis: The file was not created because the file
                    name in FN was invalid.
                Failed: LegalFileName(FN)
                Postconditions: not LegalFileName(FN)
                Obligations: <none>
                Recovery: Ensure that FN has an appropriate string.
            <Exception FileAlreadyExists: CreateFile == 2> . . .

    int OpenFile(<in> filename FN; <out> fileptr FP) ...

    void CloseFile(<inout> fileptr FP)
        Synopsis:  CloseFile closes the file and trashes the file
                   pointer FP.
        Preconditions:
            ValidFilePtr(FP)          <assumed>
            FileOpen(FP)              <assumed>
        Results:
            <Successful result: assumed>
                Synopsis: Assumes the file is open and FP is a valid
                    file pointer, the file is closed and FP is no
                    longer valid.
                Postconditions:
                    FileClosed(FP)      not ValidFilePtr(FP')
                Obligations: <none>

    int ReadRecord(<in> fileptr FP; <in> int R;
                   <out> int L; <out> buffer B) ...

    int WriteRecord(<in> fileptr FP; <in> int R;
                    <in> int L; <inout> buffer B)
        Synopsis:  . . .
        Preconditions:
            ValidFilePtr(FP)          <assumed>
            FileOpen(FP)              <assumed>
            LegalRecordNr(R)          <validated>
            RecordIn(B)               <assumed>
            Allocated(B)              <validated>
            BufferSizeSufficient(B, L) <validated>
            RecordWriteable(R)        <dependent>
        Results:
            <Successful result: WriteRecord == 0>
                Synopsis: The record R has been written in the file
                    denoted by FP.
                Postconditions:
                    LegalRecordNr(R)    BufferSizeSufficient(B,L)
                    Deallocated(B)      RecordExists(R)
                Obligations: <none>
            <Exception IllegalRecordNr: WriteRecord == 1>
                Synopsis: An invalid record number.
                Failed: LegalRecordNumber
                Postconditions: not LegalRecordNr
                Obligations: <none>
                Recovery: . . .
            <Exception UnallocatedBuffer: WriteRecord == 2>
                . . . Postconditions: not Allocated(B) . . .
            <Exception InsufficientBufferSize: WriteRecord == 3> . . .
            <Exception WriteError: WriteRecord == >
                Synopsis: The record has not been written due to an
                    I/O error.
                Failed: RecordWriteable(R)
                Postconditions:
                    LegalRecordNr(R)    BufferSizeSufficient(B, L)
                    Allocated(B)        not RecordWriteable(R)
                Obligations: <none>
                Recovery: RecoverFileSystem

    boolean FileExists(<in> filename FN) ...
        Synopsis:  FileExists determines whether the file named
                   by FN exists.
        Preconditions:
            LegalFileName(FN)         <assumed>
        Results:
            <Successful result: FileExists == TRUE>
                Synopsis: The file exists.
                Postconditions: FileExists(FN)
                Obligations: <none>
            <Successful result: FileExists == FALSE>
                Synopsis: The file does not exist.
                Postconditions: not FileExists(FN)
                Obligations: <none>

***** End FileManagement *****
```

Example 1: A sample Specification of FileManagement

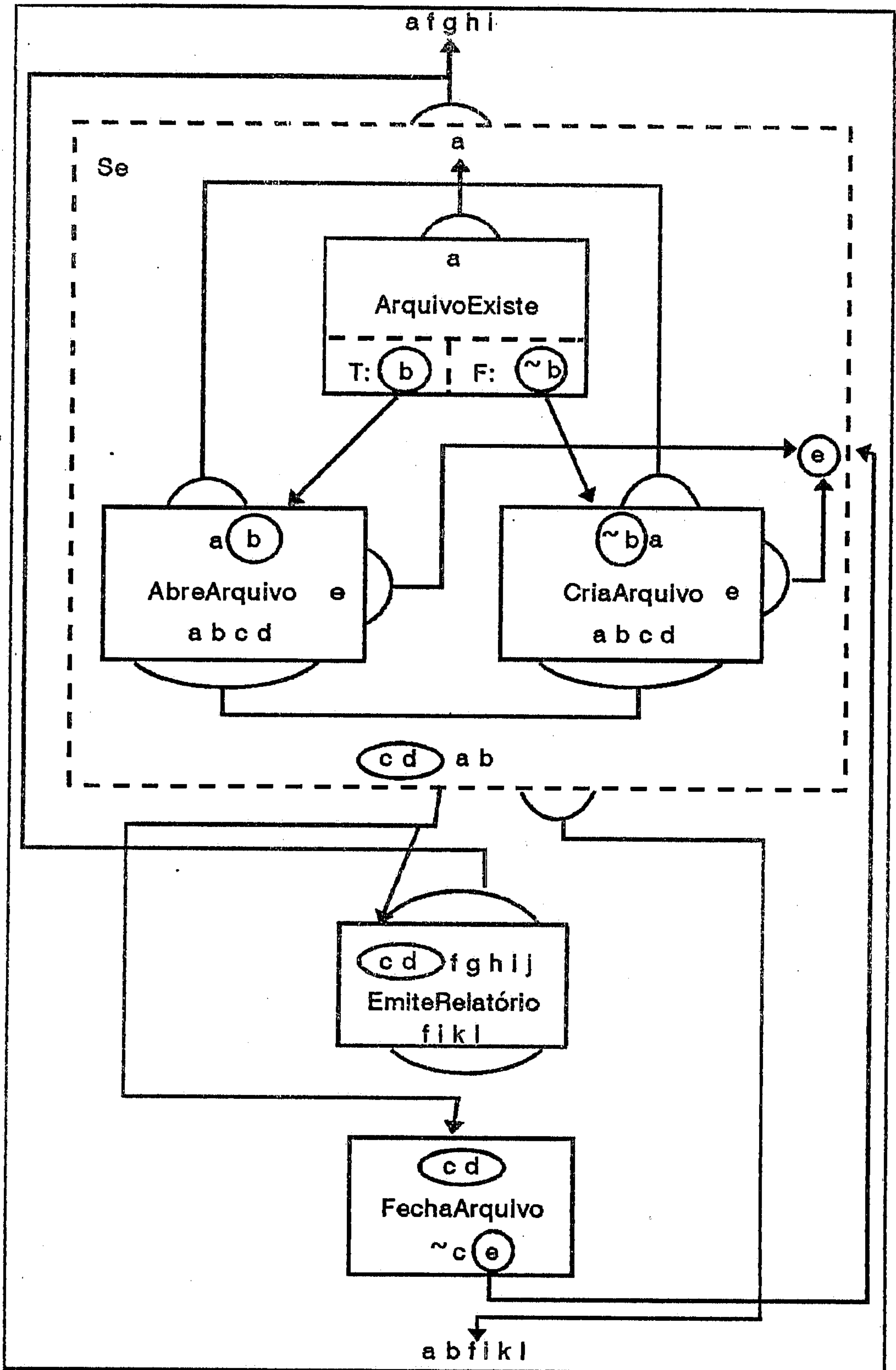
Figura 5: Exemplo de Implementação
Fonte: Peryy 89

III.1.4 - INFORM: Editor para Construção de Sistemas

INFORM é um editor para construção interativa e cooperativa de programas, que incorpora o conhecimento de especificações INSTRESS, a linguagem de implementação e as regras de INSCAPE para a construção de programas.

O ambiente maximiza o uso deste conhecimento para minimizar o esforço do programador na construção de sistemas manuteníveis. Como a implementação é construída de forma interativa, o ambiente determina quando a implementação está completa e, então, cria automaticamente a especificação da interface a partir da implementação. Caso uma especificação da interface já exista, esta especificação pode ser comparada com a especificação da interface derivada da implementação, para determinar a correção da implementação.

Existem dois tipos de visões disponíveis em INFORM: várias visões contextuais e visões sintetizadas. Neste último tipo estão os exemplos das figuras 6 e 7. A figura 6 mostra graficamente uma visão da interconexão semântica (dependências e fontes) e a figura 7 mostra graficamente uma visão do fluxo de controle.



a:NomeVálidoArquivo(FN)
b:ArquivoExiste(FN)
c:ArquivoVálidoPtr(FP)
d:ArquivoAberto(FP)
e:ArquivoFechado(FP)
f:RelatórioVálidoNr(R)
g:Relatório(B)
h:Alocado(B)
i:SuficienteTamanhoBuffer(B,L)
j:RelatórioEscrito(R)
k:NãoAlocado(B)
l:RlatórioExiste(R)

Figura 6: Visão das Interconexões Semânticas
Fonte: PERRY 89

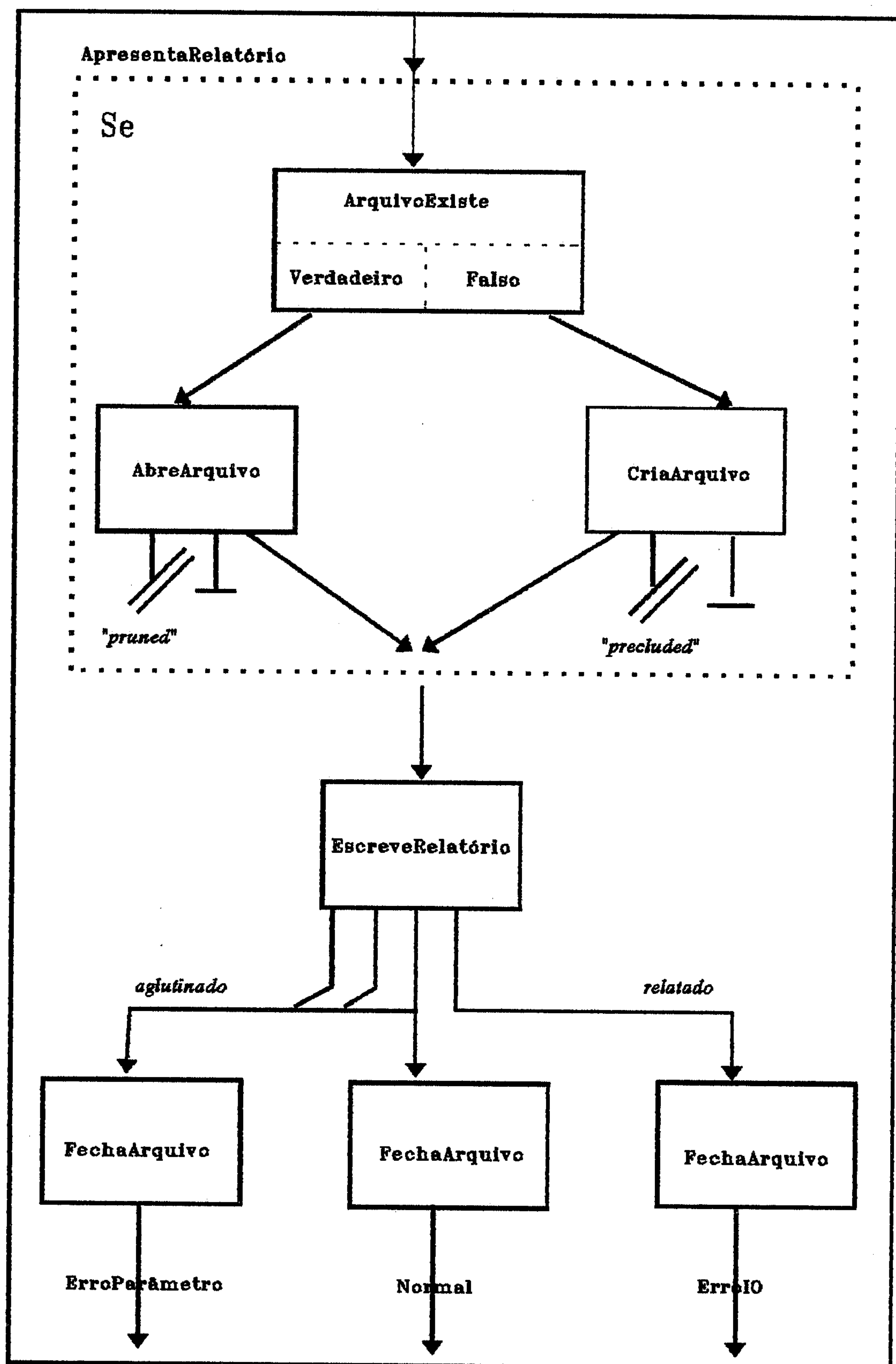


Figura 7: Visão do Fluxo de Controle
Fonte: PERRY 89

III.1.5 - INFUSE: Evolução do Sistema

O subsistema INFUSE gerencia e coordena as mudanças feitas por vários programadores e as propaga interativamente, para garantir sua completude e consistência. Os benefícios situam-se na evolução apoiada pelo ambiente que permite, entre outros, a compreensão dos efeitos de mudanças num nível mais profundo do que seria possível com métodos informais, e, o retorno do investimento nas especificações dado o apoio que oferece na parte evolutiva do Ciclo de Vida do Software. Isto torna-se, ainda, mais significativo na medida em que nos damos conta que a evolução de um sistema começa muito cedo no processo de desenvolvimento ocasionada por erros nos requisitos, projeto, especificação da interface e implementação.

III.1.6 - INQUIRE E INVARIANT: Utilização e Reutilização

Existem dois aspectos importantes a serem considerados ao se reutilizar componentes na construção de sistemas: encontrar os componentes para usar e substituir um componente por outro no modelo do sistema. INQUIRE resolve o primeiro problema e INVARIANT o segundo.

INQUIRE permite a navegação pela base do sistema seguindo interconexões unitárias, sintáticas ou semânticas, o que é importante no processo de desenvolver a compreensão de um sistema, examiná-lo e encontrar componentes para a utilização.

Dado que as ferramentas de INSCAPE são baseadas na semântica, é o comportamento das operações e as propriedades dos dados que são importantes na determinação da utilização e reutilização. Uma escolha entre possíveis objetos requer, ainda, uma análise do custo associado a cada possibilidade, o que também é fornecido por INQUIRE.

INVARIANT fornece definições precisas de equivalência e compatibilidade, e um útil conceito de "plug-compatibility" na composição de versões do sistema (duas versões são equivalentes se as especificações de suas interfaces são idênticas).

III.1.7 - VALIDAÇÃO

INSCAPE é um ambiente mais formal mas também prático. Isto fez com que fosse sacrificada uma parte da verificação

semântica tendo-se uma forma de verificação da consistência mais simples, embora útil. Esta simplificação faz com que se detecte um volume razoável de erros lógicos, mas não todos. Por esta razão são considerados modos de integrar testes no ambiente.

III.1.8 - Conclusão

INSCAPE é um ambiente integrado que provê ferramentas com conhecimento sobre o processo de construção e evolução do sistema e que trabalha em simbiose com os desenvolvedores e encarregados da evolução de sistemas.

Se considerarmos a própria classificação de Ambientes de Desenvolvimento de Software feita por Perry (PERRY 1988), INSCAPE é um ambiente do tipo cidade dado que foi desenvolvido para apoiar grandes projetos e tem procedimentos para apoio à interação de grupos.

III.2 - IDeA (Lubars 1989)

O sistema IDeA é um protótipo de ambiente de projeto, que pretende demonstrar que um único ambiente pode integrar o suporte de vários aspectos, liberando o projetista dos aspectos mecânicos do processo de desenvolvimento do projeto tornando possível que este se concentre nos aspectos mais complexos do projeto. Para isso, IDeA fornece suporte baseado em conhecimento para reutilização de software e outros aspectos inteligentes de suporte ao projeto de software.

A justificativa para a construção de tal ambiente, reside no fato de que o desenvolvimento de software é complexo envolve tarefas tediosas.

Estas tarefas envolvem a exploração e análise de alternativas para projeto, a consideração e reutilização de soluções anteriores, a gerência dos objetivos, dependências e soluções parciais de projeto e o registro das decisões de projeto.

IDeA tem como objetivo dar suporte a sete aspectos que serão descritos a seguir: atividades clericais, interface usuário-ambiente, análise, teste, organização, conhecimento e inteligência.

III.2.1 - Visão Geral do Ambiente

IDea está baseado no paradigma de desenvolvimento orientado a fluxo de dados e como conseqüência apoia os métodos de análise e projeto estruturados. A figura 8 mostra a estrutura do ambiente.

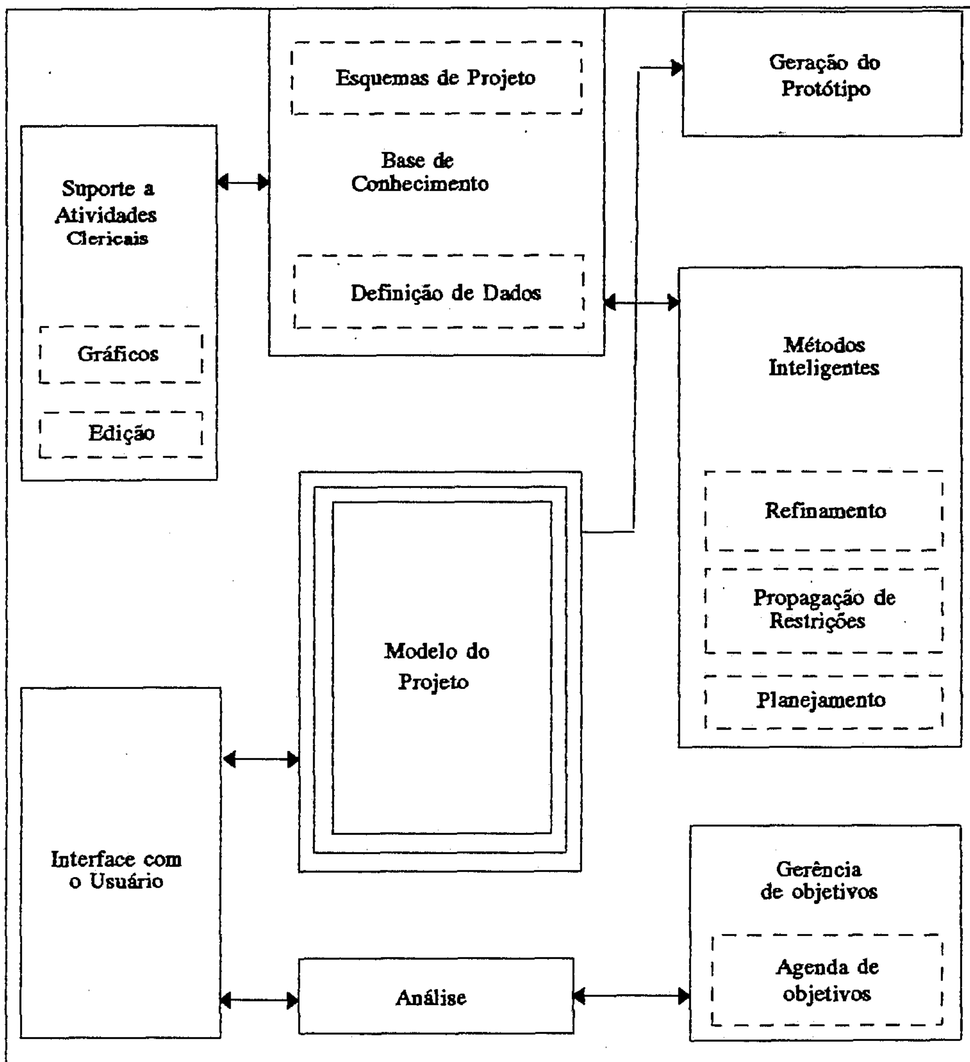


Figura 8: Estrutura do Ambiente IDea
Fonte: LUBARS 89

A seguir descreveremos como IDeA fornece facilidades de suporte aos sete aspectos identificados.

III.2.1.1 - Suporte a atividades clericais

IDeA fornece suporte para construção de projeto. baseado em fluxo de dados. através de um editor gráfico. facilidades de layout gráfico automático. acesso ao dicionário de dados e à biblioteca de esquemas de projeto.

O editor permite ao usuário:

- criar. combinar. modificar ou apagar diferentes componentes dos fluxos de dados;

- mostrar todos ou. um só nível do fluxo de dados e focalizar objetos específicos;

- consultar e acessar de forma geral o dicionário de dados. a biblioteca de esquemas e outros componentes baseados em conhecimento;

- ajustar. automaticamente. o projeto para atender às exigências de certos tipos de requisitos de consistência. e.

- acessar a facilidades inteligentes, tal como planejamento.

III.2.1.2 - Suporte à Interface

IDeA apresenta as informações do projeto em três tipos separados de janelas (figura 9):

Figura 9: Um exemplo do Projeto IDeA
Fonte: LUBARS 89

- Janela de diagrama de fluxo de dados, localizada na área superior direita, que apresenta o diagrama de

fluxo de dados do nível corrente, o número do nível e o nome da transformação correspondente;

- Janela de Definição de Transformações, localizada na área central, que apresenta informações sobre cada transformação no fluxo de dados no nível corrente: o número, o nome da transformação e um nome interno, que é único para cada transformação;

- Janela de Definição do Fluxo de Dados, localizada na área esquerda, contém a informação correspondente a cada fluxo de dados no diagrama de fluxo de dados corrente: a identificação, a descrição do tipo de fluxo de dados e um nome interno.

- A janela no nível mais baixo à direita é a janela de interação, na qual o usuário entra com os comandos e recebe "feedback" do sistema.

Uma linguagem natural também foi desenvolvida. Seu objetivo é interpretar o fluxo de dados e as especificações funcionais, aceitar diretivas informais de projeto e solicitar informações adicionais ao usuário, quando necessárias. Consiste de um "parser" estático e semântico baseado na gramática ATN ("Augmented Transaction Network").

III.2.1.3 - Suporte à Análise

IDeA mantém de forma automática. o balanceamento do projetos. Isso significa que todas as transformações no projeto são mantidas consistentes com seus refinamentos em termos dos números e tipos de seus diagramas de entrada e saída.

No caso de adição de uma entrada num determinado nível de projeto, isto requer que o nível superior considere a criação deste fluxo de dados. IDeA adiciona uma cópia deste fluxo de dados no nível superior. Entretanto, como o sistema não sabe onde colocá-lo, o fluxo é deixado desconectado e uma meta para determinar esta informação é incluída na agenda de metas de IDeA, de forma a garantir que o projetista resolva o problema posteriormente.

Este apoio fornecido por IDeA é um exemplo de suporte que dispensa o projetista de realizar um trabalho desnecessário de edição. Como conseqüência, para cada modificação lógica, o projetista tem que editar apenas uma modificação e não várias ao longo dos diferentes níveis de projeto. IDea também aponta para as ramificações das alterações nos demais níveis do projeto evitando que o projetista deixe de considerá-las.

III.2.1.4 - Suporte à Prototipagem

Os projetos gerados podem ser diretamente executados, numa arquitetura de suporte apropriada, provendo código para processos individuais e tratando os fluxos de dados como caminhos de comunicação entre processos.

A linguagem para interconexão de módulos e o sistema de suporte à execução Polylith fornecem esta arquitetura. IDeA gera código Polylith com um protótipo executável.

III.2.1.5 - Suporte Organizacional

IDeA mantém uma agenda com metas do projeto, como uma lista de prioridades. Novos componentes são automaticamente colocados na lista para refinamento, da mesma forma que os fluxos de dados adicionados ao diagrama aguardam sua união às transformações fonte e resultado. Quando dirigido pelo usuário, IDeA seleciona as metas pendentes e tenta alcançá-las, usando suas características inteligentes, ou apresentando-as como tarefas ao usuário.

Se determinados objetivos não podem ser alcançados em uma situação particular, o projeto deve retornar a estados prévios, e novos refinamentos devem ser explorados.

Quando já não existem metas pendentes e o usuário está satisfeito com o resultado do projeto, o projeto é considerado completo com relação à base de conhecimentos de IDeA.

III.2.1.6 - Suporte Baseado em Conhecimento

A base de conhecimento de IDeA consiste de vários componentes, sendo os principais, o catálogo de esquemas e o dicionário de dados. Estes estão organizados numa hierarquia abstrata para permitir herança e partilha de componentes. Os objetos nas hierarquias estão organizados de acordo com os vários domínios de aplicação e suas características, de forma a relacionar grupos de projetos semelhantes, que podem ser descritos por abstrações comuns.

As referências cruzadas entre esquemas e o dicionário de dados, facilita a seleção de esquemas. IDeA utiliza os componentes do sistema especificado pelo usuário e os fluxos de dados como padrões para serem localizados na base de conhecimentos. Se os componentes são encontrados, são analisados para garantir que se encaixam no fluxo de dados parcialmente construído e, então, são integrados aos outros componentes.

III.2.1.7 - Suporte Inteligente

Os métodos inteligentes e baseados em conhecimento de IDeA, suportam um método de projeto semi-automático, através de um diálogo construtivo com o projetista, que fornece as especificações. IDeA usa-as para construir e refinar o projeto do sistema desejado.

III.2.2 - Conclusão

IDeA foi implementado em ambiente SUN e tem sido usado, principalmente, na experimentação da abordagem baseada no conhecimento, para reutilização de projetos.

Segundo seus desenvolvedores a experiência de integração de técnicas inteligentes e baseadas em conhecimento, em um ambiente de projeto, é um enfoque adequado para fornecer um suporte completo ao desenvolvimento.

Por sua natureza baseada em fluxos de dados, IDeA tem sua aplicação limitada a certos tipos de projetos.

III.3 - ARCADIA (TAYLOR et alli 1989)

O projeto ARCADIA, é um esforço conjunto de pesquisa envolvendo empresas, centros de pesquisa e universidades: Universidade da California, Irvine; Universidade do Colorado, Boulder; Universidade de Massachussets, Amherst; TRW e o Laboratório AT&Bell . O objetivo do projeto é pesquisar sobre a construção de ambientes de desenvolvimento de software integrados e ao mesmo tempo flexíveis e extensíveis.

Para atingir este objetivo o projeto ARCADIA está direcionado a pesquisar e desenvolver protótipos que sirvam de demonstração para:

- arquiteturas de ambientes para a organização de grandes conjuntos de ferramentas, facilitando suas interações com usuários bem como com outras ferramentas;

- ferramentas para facilitar teste e análise de software concorrente e seqüencial;

- contextos para avaliação de ambientes e ferramentas.

III.3.1 - Arquitetura do Ambiente ARCADIA

Pesquisas em ambientes de desenvolvimento de software vem demonstrando que estes devem ser altamente flexíveis, extensíveis e integrados. No entanto, essas características são, algumas vezes, conflitantes. O objetivo de se ter um ambiente integrado é facilmente alcançado se este for estático e tiver limitado seu escopo. Isto, entretanto, os torna rapidamente obsoletos. Por outro lado, um ambiente amplo e dinâmico, tende a ser pobremente integrado, o que implica em dificuldades para seus usuários.

A arquitetura de ARCADIA foi definida com o objetivo de maximizar a flexibilidade e a extensibilidade, compreender as tensões existentes entre estes dois princípios conflitantes e, buscar a integração do ambiente. Foram motivações para este enfoque, constatar que:

- usuários são diferentes e percebem suas necessidades de formas diversas;

- diferenças nos projetos implicam em diferentes requisitos de suporte;

- ocorrem mudanças na percepção dos desenvolvedores,na medida em que as necessidades são alteradas ao longo do avanço dos projetos.

Estas questões conduzem, portanto, para a busca de um ambiente cuja flexibilidade permita mudanças na natureza do suporte a seus usuários e, ainda, a incorporação de novas tecnologias e ferramentas, o que evitará sua obsolescência.

A questão flexibilidade/extensibilidade no projeto ARCADIA está suportada pela noção de processos de programação. A idéia básica é que os processos de programas, escritos em PPL (Process Programming Language), irão descrever os diversos processos de software que os usuários do ambiente desejam utilizar no desenvolvimento e manutenção de software. Com este modelo será alcançada a flexibilidade através do suporte a alterações nos processos. Extensibilidade é obtida pela possibilidade de se escrever novos processos ou de realizar modificações nos já existentes para incorporar novas ferramentas, subprocessos, tipos ou objetos.

O requisito de uma firme integração tem uma série de manifestações:

- os usuários podem interagir com o ambiente de maneira uniforme em vez de terem de se adaptar a diferentes interfaces de ferramentas;

- as ferramentas do ambiente devem partilhar informações entre si, evitando que os usuários sejam molestados para fornecer a mesma informação várias vezes;

- componentes do ambiente devem ser partilhados sempre que possível.

Após a observação dos aspectos acima, a questão da integração no ambiente ARCADIA ficou dividida em questões de integração interna e externa. A integração interna preocupa-se com aspectos de gerenciamento de objetos no ambiente. A integração externa, trata as questões relativas ao gerenciamento da interface com o usuário, de forma que esta seja o mais uniforme e confortável possível.

A figura 10 mostra a arquitetura de alto nível do protótipo do ambiente ARCADIA.

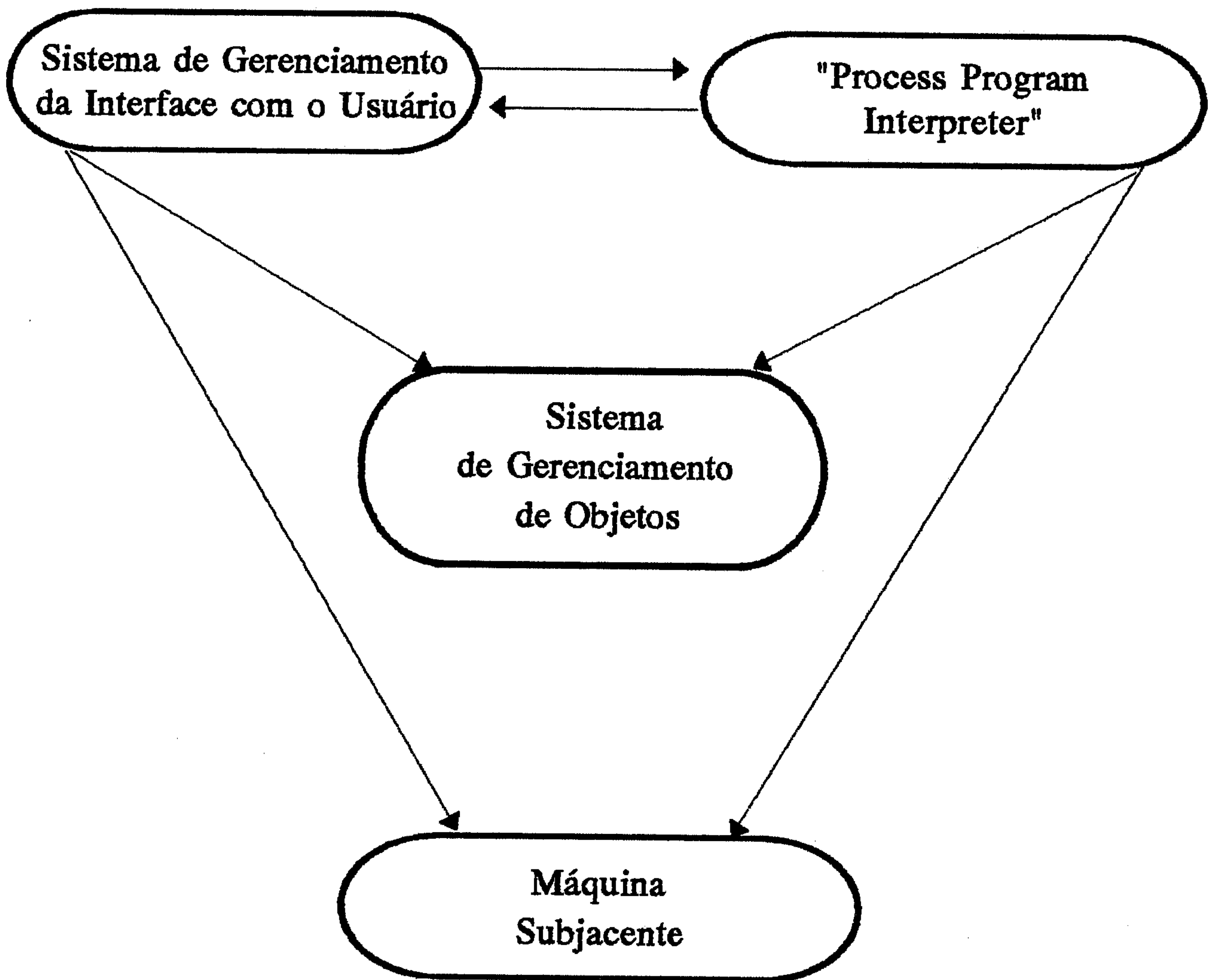


Figura 10: Arquitetura do Ambiente Arcadia
Fonte: Taylor et alli 89

Nesta arquitetura o "Process Program Interpreter" (PPI) é responsável por levar as instruções dos programas de processos. "PPI" comunica-se com os usuários através do Sistema de Gerenciamento da Interface com o Usuário (UIMS) e acessa os objetos de software, via o Gerenciador de Objetos. Estes três componentes devem interagir com uma máquina virtual para suporte básico ao sistema operacional. Nesta versão está o sistema operacional Unix de Berkeley, versão 4.3, rodando numa rede de estações SUN.

A figura 10 mostra a infra-estrutura, ou parte fixa do ambiente e, portanto, não mostra os processos de programas, ferramentas e objetos de software que povoam o ambiente. Estes componentes mudam com frequência e por isso são chamados de parte variável da arquitetura. A arquitetura de ARCADIA encoraja mudanças nesta parte variável e desencoraja na parte fixa. Esta é uma das contribuições do projeto ARCADIA, ao reconhecer a necessidade de se separar a parte fixa e a variável numa arquitetura do ambiente.

É desejável que um ambiente de desenvolvimento de software tenha um sistema de gerenciamento de objetos poderoso. Este sistema poderá, assim, propiciar e suportar mudanças, integração, reutilização de software e, ainda, o trabalho cooperativo entre múltiplos desenvolvedores.

Como está evidenciado na figura, o sistema de gerenciamento de objetos é o principal componente da infra-estrutura do ambiente ARCADIA. Ele é responsável pelo gerenciamento de duas categorias diferentes de objetos: os componentes dos produtos de software que são produzidos pelos usuários do ambiente e, as ferramentas e estruturas de informação que constituem o próprio ambiente.

O terceiro componente da infra-estrutura do ambiente é o Gerente da Interface com o Usuário.

Este componente irá prover uma interface com o usuário agradável e um acesso eficiente às funções suportadas pelas partes fixas e variáveis de um ambiente. Esta interface tem como requisitos:

- uniformidade (ou consistência), de forma a reduzir os detalhes que o usuário deve memorizar;

- manipulação direta aumentando a comunicação entre ferramentas e usuários através de mecanismos gráficos;

- permissividade, de forma a permitir que seja o próprio usuário quem escolha a seqüência de ações a realizar.

III.3.2 - Conclusão

O projeto ARCADIA é um projeto ambicioso de pesquisa e desenvolvimento cujo objetivo é investigar aspectos inovadores de ambientes. O objetivo básico é estudar ambientes integrados e ao mesmo tempo flexíveis e extensíveis. Estas características permitirão à ARCADIA incorporar ferramentas desenvolvidas por uma comunidade variada.

III. 4 - ADAGE (GIAVITTO 1989)

ADAGE, um ambiente de ferramentas CASE genérico objetiva garantir qualidade, confiabilidade e a conseqüente redução nos custos de desenvolvimento, fatores preponderantes nos projetos de pesquisa atuais.

Inexiste, nos ambientes atuais, a idéia do "todo", onde é pobre a representação dos dados, suportando apenas parcialmente o ciclo de vida. Os ambientes são fechados, habitualmente ligados a uma metodologia, o que impede sua evolução e ainda, a construção e integração de novas ferramentas. As interfaces com o usuário são, normalmente, pouco amigáveis, as verificações de coerência entre uma ou mais fases são insuficientes, acarretando dificuldades na comunicação e ausência de suporte ao trabalho em equipe.

Ferramentas CASE deveriam propiciar este tipo de utilização, integrando e manipulando de forma coerente os dados em um mesmo ambiente, mas existem algumas falhas nos produtos atuais disponíveis.

ADAGE pretende, através de uma estrutura voltada para todas as fases do desenvolvimento, a integração das ferramentas necessárias para cobrir todo o ciclo de vida, qualquer que seja o modelo utilizado.

III.4.1 - O aspecto genérico em ADAGE

III.4.1.1 - O Meta-Modelo

Foi desenvolvido um meta-modelo de dados, possante e expressivo, que possibilita gerar ambientes, independente do método utilizado. Este, deverá ser capaz de representar as noções de agregação, decomposição e hierarquia, necessárias à fase de programação global, aqui entendida como a gestão de versões e configuração, a documentação, acoplamento dos módulos, etc., e que podem estar representados por grafos, uma vez que trata-se de uma ferramenta de amplo espectro, permitindo fácil visualização dos objetos e de suas relações.

ADAGE suporta, ainda, para as fases de análise e concepção, a utilização de linguagens gráficas, fortemente calcadas no conceito de grafos estruturados, o que justifica a construção do meta-modelo ter sido baseada em GDL (Graph Description Language).

III.4.1.2 - Propriedades

ADAGE, em seu aspecto genérico, caracteriza-se por:

- neutralidade, isto é, não está ligado a nenhum método particular, tornando possível, inclusive, a coexistência de diferentes formalismos;

- configurabilidade, que consiste na possibilidade de instanciar o meta - modelo ADAGE, permitindo obter um ambiente que se adapte a diferentes domínios de aplicação:

- extensibilidade, que prevê a evolução do sistema ao longo do tempo, dada a evolução dos métodos, ferramentas e linguagens

- flexibilidade e abertura, permitindo a integração de estruturas e organizações existentes

III.4.2 - A linguagem GDL

O objetivo de uma definição na linguagem GDL é modelar os dados gerados pelo ambiente, permitindo representar entidades abstratas construídas a partir de recursos básicos oferecidos pelo sistema.

O componente básico de GDL é o nó. Este possui um grafo que representa suas interações com o ambiente e os atributos, que representam suas qualidades intrínsecas.

A parte grafo de um nó representa suas interações com os outros nós. Um grafo é composto de outros nós e de flechas. Uma flecha liga dois nós e, desta forma, estabelece uma relação binária entre duas entidades.

III.4.3 - A Arquitetura de ADAGE

Os procedimentos usados para gerar um ambiente são os seguintes:

- instanciação do meta-modelo pela descrição em GDL de um método particular;

- parametrização das ferramentas genéricas: criação da base de dados pela compilação incremental das descrições em GDL dos tipos de nós e de suas instâncias, compilação do código mantendo as restrições sobre a base de dados, descrição dos estilos de representação para a interface com o usuário;

- inserção de novas ferramentas (e de ferramentas autônomas já existentes) na estrutura de ADAGE, utilizando a linguagem GRaal.

O resultado de uma instanciação ADAGE é o núcleo de um assistente desenvolvedor capaz de levar em conta:

- gerenciamento lógico de dados;
- armazenamento/recuperação da informação;
- verificação sintática;
- a pesquisa e a navegação nos dados;
- a execução das atividades automatizadas.

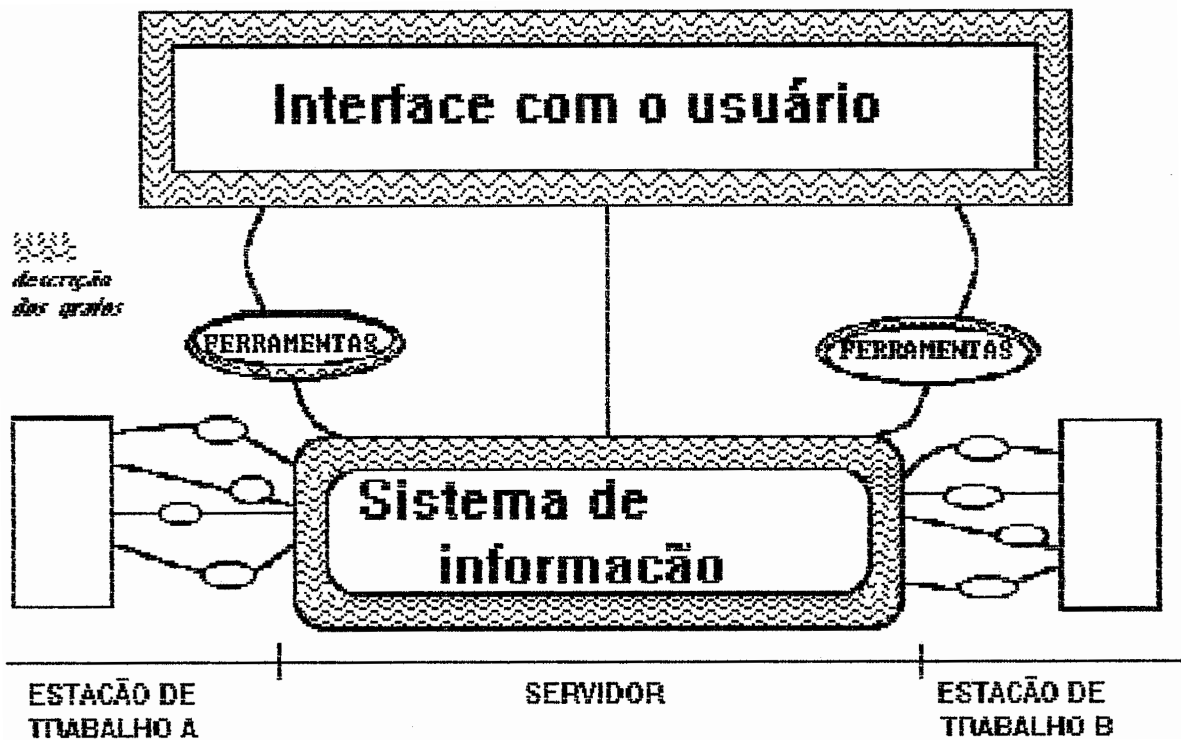


Figura 11: Arquitetura de ADAGE
Fonte: GIAVITTO 89

O assistente está implementado por dois subsistemas:

- o sistema de pilotagem (ou a interface com o usuário);
- o sistema de informação (ou servidor na terminologia ADAGE).

III.4.3.1 - A interface com o usuário

- **Navegação:** A navegação entre os dados de ADAGE é feita a partir da interface com o usuário. É uma ferramenta genérica capaz de manipular qualquer dado definido em GDL.

A navegação em ADAGE é feita num estilo aplicativo. através da linguagem GRaal.

Os tipos de requisitos necessários em um ambiente para uma navegação eficaz são mais complexos que aquelas fornecidas pelas linguagens usuais.

A interface com o usuário em ADAGE integra à linguagem GRaal, uma interface gráfica, baseada em múltiplas janelas, acessadas por "mouse". Ela é parametrizável pelo usuário que pode , por exemplo, decidir a representação de um nó em função de seu tipo.

- **GRaal:** Os dados gerados em ADAGE podem ser manipulados através de uma linguagem de requisitos interpretados, chamada GRaal (Graph Request Language). A criação de um nó não é de responsabilidade de Graal. Esta se dará a qualquer momento, fora dos programas que os poderiam manipular. E é por esta razão que a declaração dos dados é feita por uma ferramenta autônoma, o compilador GDL. Esta ferramenta é também explicitamente chamada a partir de um

programa em GRaal. Desta forma, GDL pode ser vista como uma sub-linguagem da declaração de dados da linguagem GRaal.

III.4.3.2 - O Servidor

O servidor é o repositório comum dos dados do ambiente armazenados em uma base de dados. O esquema aqui utilizado provém de uma descrição GDL.

As principais funções do servidor são:

- gerência dos dados manipulados pelo ambiente: criação/destruição/atualização, armazenamento/recuperação, propagação de restrições...;

- gerência dos esquemas que estruturam e parametrizam o ambiente;

- gerência da concorrência de acesso.

III.4.4 - Conclusão

Dois objetivos principais estão subjacentes ao projeto ADAGE:

– O Domínio da Tecnologia

Fortemente interativas, concebidas para auxiliar o programador, as ferramentas do ambiente devem oferecer interfaces avançadas e partilhadas. Uma interface partilhada permite uma navegação mais eficaz e diminui o tempo de adaptação a um sistema por usuários pouco familiarizados com ferramentas sofisticadas.

A eficácia é necessária visando ferramentas verdadeiramente utilizáveis.

– A Capacidade de Evoluir

O aspecto genérico, a instanciação, a extensibilidade, são as chaves para preparar o futuro: elas permitem levar em conta as próximas gerações de métodos e ferramentas, a reutilização e aperfeiçoam a adequação das ferramentas às aplicações.

Um primeiro protótipo foi realizado em 1988 para validar os conceitos de base do meta-modelo e as decisões de arquitetura. A validação das idéias expostas neste artigo levam à construção do segundo protótipo.

III.5 - CENTAUR (BORRAS 1988)

CENTAUR é um ambiente genérico e interativo. Quando lhe é fornecida a especificação formal de uma determinada linguagem de programação, incluindo sua sintaxe e semântica, produz um ambiente específico para esta linguagem, que compreende, um editor de estrutura, um interpretador/depurador e outras ferramentas, todas elas possuindo interface gráfica.

A construção de tal ambiente é justificada pela expectativa do aparecimento de novas linguagens, como uma atividade bastante comum no futuro. No projeto CENTAUR, foram identificados três desafios:

a) É possível utilizar como descrição de uma linguagem de programação uma especificação formal no sentido usado pelos pesquisadores de semântica de linguagens de programação?

b) Pode-se obter, nestas bases, um sistema eficiente, tanto em termos de velocidade de processamento quanto de espaço na memória?

c) É possível criar uma interface homem-máquina comparável em conveniência com o que pode ser encontrado nos sistemas comerciais, por exemplo, em um MacIntosh?

I.5.1 Arquitetura de CENTAUR

A arquitetura do gerador de ambientes CENTAUR é composta de:

- Um **núcleo**, para a representação e manipulação de objetos estruturados. Ocupa um papel central na arquitetura, uma vez que está voltado para ser usado de maneira direta ou indireta, por todos os outros componentes do sistema. Seu objetivo é suportar manipulação simbólica (aqui entendida como manipulação destrutiva, tipicamente usada na edição de documentos, e avaliação, mais comum em processamento semântico) de documentos estruturados. O núcleo de CENTAUR é composto de dois componentes:

    - uma máquina abstrata que trata de aspectos sintáticos, chamada "Virtual Tree Processor" (VTP);

    - uma máquina lógica que trata de aspectos semânticos, isto é, avaliação de todos os tipos. Neste nível a máquina lógica é um interpretador Prolog.

- um **nível de especificação** para suportar os aspectos da sintaxe e da semântica das linguagens. Este nível é essencial para se ter a geração do ambiente de programação específico.

A especificação da sintaxe, contém os componentes necessários para derivar um editor orientado a estrutura. Para derivar um ambiente de programação específico para uma determinada linguagem é necessário ir além da especificação da sintaxe. Isto é feito através das especificações semânticas.

- uma **interface com o usuário** que gerencia as interações entre o sistema e seus usuários. As funções de CENTAUR são acessadas de duas formas não exclusivas: através de chamadas diretas das funções, e através da interface homem-máquina. O projeto da interface prevê que os usuários interagem com CENTAUR através de uma estação de trabalho moderna, com todas as facilidades que estas oferecem.

CENTAUR foi implementado em LISP e Prolog. Pode ser executado em Estações de Trabalho ou em equipamentos de grande porte.

Seu objetivo vem responder à necessidade de se terem ambientes configuráveis dada a grande evolução da área.

Procura, desta forma atender às necessidades de fornecer ambientes de programação específicos para quaisquer linguagens desde que estas sejam descritas utilizando formalismos adequados.

III.6 - PEGASE (BERNAS 1989)

PEGASE é um ambiente de desenvolvimento gráfico que a partir de especificações algébricas deriva programas em ADA e sua respectiva documentação.

O ambiente consiste de um conjunto de ferramentas de desenvolvimento organizadas em torno de um servidor, o banco de dados do projeto. Este servidor interage, graficamente, com o usuário através de menus, mouse e grafos.

O ambiente é orientado para a gerência de versões destas especificações, para os aspectos de interface gráfica e gerência de múltiplos usuários.

Assim sendo, o ambiente PEGASE é formado por um conjunto de ferramentas integradas que reúne:

- o sistema de gerência da base de projetos que controla todas as informações sobre um conjunto de projetos que foram desenvolvidos no contexto de PEGASE;

- uma interface homogênea que permite conduzir e observar o desenvolvimento;

- um conjunto de ferramentas de desenvolvimento de especificações e programas e suas interfaces com o sistema de gerência da base do projeto.

Várias ferramentas podem ser chamadas simultaneamente, bem como uma mesma ferramenta diversas vezes. Vários usuários podem trabalhar ao mesmo tempo no mesmo projeto, cada um dispondo de um servidor específico.

III.6.1 - Características do servidor em PEGASE

- sessões de trabalho são abertas tão logo o ambiente é chamado, e estas podem, a qualquer instante, ser armazenadas ou anuladas. Sessões paralelas de utilizadores diversos podem ser integradas, embora sejam completamente independentes. A coerência deste mecanismo está assegurada pelo modelo de desenvolvimento de software utilizado em PEGASE;

- são permitidas chamadas simultâneas das ferramentas de desenvolvimento e, suas interrupções a qualquer momento;

- panes estão previstas e, o ambiente é robusto o suficiente de forma a permitir a melhor reconstituição possível;

- um bom desempenho está assegurado através de dois níveis de armazenamento de dados, que permitem, ainda, que um projeto passe de um nível para outro:

- nível usual, correspondendo ao que se vê, visualisando os projetos, e,

- nível de armazenamento, onde estão armazenados os projetos que não são utilizados freqüentemente.

- são permitidos o intercâmbio de projetos completos, entre diferentes bases de projeto PEGASE. Isto leva a um ambiente evolutivo e adaptável a outras estruturas de objetos a desenvolver.

- parte do servidor é gerada automaticamente a partir da descrição dos tipos dos objetos e de suas relações.

III.6.2 - Modelo de Desenvolvimento

PEGASE baseia seu desenvolvimento no modelo derivação-arquivamento, onde todo o projeto desenvolvido num ambiente passa por diferentes estados, estes formados por um conjunto de especificações, programas e documentação, suas relações e informações associadas.

III.6.3 - Desenvolvimento e Evolução no ambiente PEGASE: Versões e Especificações.

Tratando-se de um ambiente integrado, PEGASE permite a recuperação de toda a história do desenvolvimento, desde seu início até o fim de sua manutenção.

Para construção das especificações, é utilizada a linguagem PLUSS, desenvolvida por Gaudel. As ferramentas

de desenvolvimento de especificações algébricas provém do projeto ASSPRO. do ambiente Asspégaz. e do sistema REVE. Descrições detalhadas destes ambientes e ferramentas podem. segundo o autor, ser encontradas em (BIDOIT 1985), (BIDOIT 1987). (BERNAS 1988) e (FOORGARD 1984).

Um editor sintático permite escrever especificações. definindo seu nome. o nome dos módulos que elas utilizam, o nome dos tipos de dados e operações nelas definidas e os axiomas definindo suas operações.

A integridade é assegurada pois quando, durante uma edição. é feita uma modificação em um módulo. modificações subseqüentes serão efetuadas na estrutura da especificação.

A coerência sintática de uma especificação está assegurada pela validação através de um compilador. Após sua compilação, a especificação pode ser explorada por outras ferramentas. Assim sendo. pode-se, a partir da compilação. utilizar as seguintes ferramentas:

- ferramenta de análise estática, que permite obter um texto em francês ou inglês que caracteriza as propriedades da especificação e que serve de base para a geração automática da documentação

- ferramenta de geração da estrutura de programas ADA, que permite a geração das declarações de funções ADA

correspondentes às operações da especificação. Um editor sintático, a seguir da geração, permite definir as instruções destas funções.

- ferramentas de avaliação da especificação.

PEGASE permite, ainda, a visualização do histórico de desenvolvimento de uma especificação, através de suas diferentes edições, cópias, etc.

III.6.4 - Conclusão

O ambiente Asspégaz, antecessor de PEGASE, foi fartamente utilizado em diferentes situações, como por exemplo, o sistema de controle da velocidade do métro de Lyon, que provam, através de experiências, a qualidade que conferem a um ambiente. Por outro lado justificam o interesse por PEGASE, mesmo que este ainda não tenha um histórico similar de utilização

III.7 - GRASPIN (MANNUCCI 1989)

GRASPIN, Projeto ESPRIT nº 125 foi executado em conjunto por "Gesellschaft fur Mathematik und Datenverarbeitung", Siemens da Alemanha e pela Olivetti e Tecsiel da Itália. GRASPIN buscou fornecer os benefícios de um ambiente de desenvolvimento integrado, para as fases de análise e projeto.

O projeto foi iniciado em setembro de 1983 e concluído em setembro de 1989, tendo como resultado um protótipo de ambiente de desenvolvimento, baseado em uma estação de trabalho, para especificação incremental gráfica e implementação formal de sistemas não-seqüenciais. Suporta uma metodologia para análise do problema, especificação formal e desenvolvimento incremental de programas.

O ambiente GRASPIN oferece um conjunto completo de métodos e ferramentas CASE para linguagens gráficas. Como não existe um método amplamente aceito como o melhor e independente do domínio da aplicação, o ambiente GRASPIN pode ser configurado por seus usuários para o ciclo de vida, formalismos e ferramentas que este julgar conveniente.

GRASPIN possui um conjunto de ferramentas, que inclui:

- um editor dirigido por sintaxe que permite a edição incremental de documentos de forma que o usuário pode se

fixar na lógica do documento sem se preocupar com restrições sintáticas tediosas. Além disso, o editor trata aspectos semânticos e permite que o usuário defina verificações adicionais, fornecendo suporte dirigido à definição da linguagem.

- ASDL (Abstract-Sintax Definition Language) a qual permite que a incorporação de novas linguagens ao ambiente seja realizada facilmente. Além da definição sintática e semântica da linguagem, ASDL permite que sejam definidas mais de uma representação para a linguagem. Pode-se, ainda, construir facilmente uma nova ferramenta, com o conjunto de ferramentas fornecidas pelo núcleo de GRASPIN.

- ferramentas para "lay-out" automático que facilita a tarefa de produzir diagramas de alta qualidade e evita o trabalho tedioso de organizar o "lay-out" de diagramas. Esta ferramenta é configurável para novas linguagens definidas para o ambiente.

III.7.1 - Arquitetura

A figura 12 mostra a arquitetura de GRASPIN como um conjunto de módulos funcionais. Um conjunto desses módulos, o núcleo do sistema que contém a máquina abstrata, o processador de comandos e ferramentas customizáveis, permitem a configuração do ambiente para métodos e linguagens específicas, bem como a inclusão de ferramentas

para estas linguagens, as ferramentas para linguagens específicas.

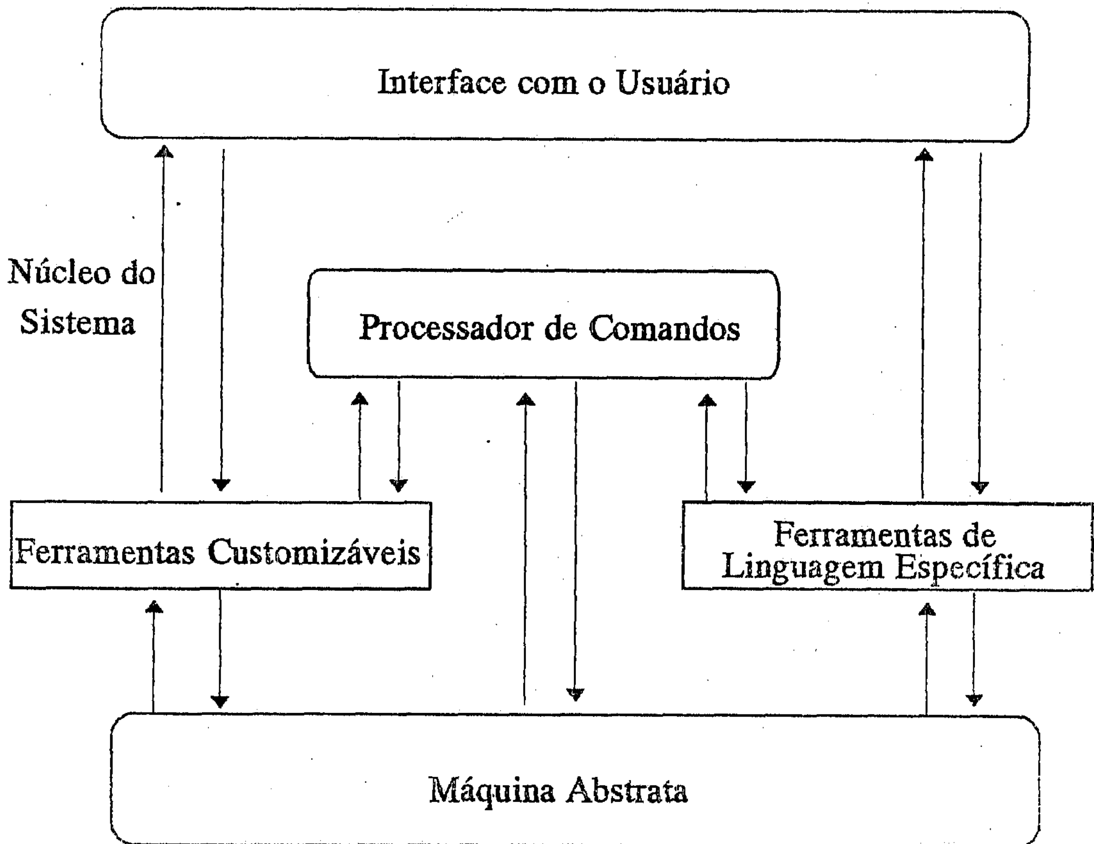


Figura 12: A Arquitetura Graspin
Fonte: MANNUCCI 89

O módulo interface com o usuário integra as ferramentas do ambiente sob o ponto de vista do usuário garantindo uniformidade e consistência em sua apresentação.

O protótipo de GRASPIN suporta, além de linguagens textuais, duas linguagens gráficas já bastante tradicionais: a linguagem para diagrama SADT e a linguagem para modelagem de entidades e relacionamentos.

São caracteristicas importantes de GRASPIN a possibilidade de configuração do ambiente para atender às necessidades de usuários especificos e as facilidades oferecidas para construção de novas ferramentas para o ambiente.

III.8 - OASIS (POSTEL 1989)

OASIS foi desenvolvido pela Aerospatiale em conjunto com o IRIT da Universidade Paul Sebastier e o IGL Technology na França. Tem como objetivo atender às necessidades da Aerospatiale no desenvolvimento e manutenção de software cientifico (simulações numéricas) com alto grau de evolução e complexidade.

OASIS é, portanto, um ambiente de desenvolvimento e manutenção de programas, oferecendo um conjunto de ferramentas para programação, documentação, validação e teste. Além disso OASIS oferece, também, apoio à reutilização.

O ambiente OASIS tem as seguintes características:

- implementação do ambiente em torno do produto IDE: o **"Software through Pictures (StP)** e sua arquitetura do tipo ambiente CASE. Isto foi possível dadas as facilidades de StP, no que se refere à sua arquitetura aberta que permite adaptação e extensão de funcionalidades.

- exploração dos recursos do StP e das capacidades de suas "conexões visíveis": base de projetos de múltiplos usuários, gerência aberta de objetos, ferramentas dotadas de interfaces públicas e modificáveis, ferramentas de geração de código e de documentação, parametrizáveis e extensíveis, etc.;

- primeira integração do modelo OASIS pela reutilização de um dos modelos formais já suportado, acompanhado por certos mecanismos complementares à edição, gerência e extensão do esquema de dados nativo;

- incorporação de funções especificas da gestão de objetos OASIS, do tratamento e controle de seus atributos, principalmente pela combinação de analisadores estáticos e dinâmicos com as ferramentas padrão e a gerência de objetos do StP.

III.8.1 - O Modelo de Desenvolvimento.

A obediência às regras é responsabilidade do gerente da simulação, que é encarregado da organização, validação e integração das modificações feitas pelos usuários, através de um ciclo de evolução que leva à criação de uma nova versão de simulação.

O modelo é composto de:

- os objetos do modelo, organizados dentro de uma estrutura hierárquica, separando os objetos gerais daqueles gerados pelo fonte, e,

- a gerência dessas modificações nos objetos, onde está definida a hierarquia entre os diferentes níveis de modificações dos objetos, através de:

- nível **"trabalho"** que correspondem ao espaço de modificação do usuário, nos quais são realizadas as modificações.

- nível **"espera de validação"**, que reúne as modificações introduzidas pelos usuários e submetidas ao gerente, para validação. Este pode transferir as modificações recebidas a este nível ao nível "espera de integração", transferindo os objetos que passaram os testes elementares previstos para a validação individual dos objetos modificados (teste individual e controle da qualidade dos programas). Este nível pode ser considerado como uma extensão do nível "trabalho": as modificações ficam relacionadas ao autor e só são visíveis para ele e para o gerente.

- nível **"espera de integração"** que reúne as diferentes modificações validadas individualmente pelo gerente. Neste nivel são, portanto, associadas as diferentes modificações feitas por diferentes usuários. Após testes de validação, cujo objetivo é verificar se juntas formam uma modificação globalmente válida, que pode ser transmitida ao nível "espera de oficialização";

- nível **"espera de oficialização"** que reúne as diferentes modificações, globalmente validadas pelo gerente. A aceitação de uma modificação se traduz

na transferência automática dos objetos modificados ao nível "referência".

- nível **"referência"** que contém todos os elementos de uma simulação totalmente validados correspondendo, portanto, ao último nível de revisão.

Em todos os níveis descritos acima, o autor ou gerente podem ver todas as modificações transmitidas, salvo no nível "espera de validação", onde somente serão vistas pelo usuário, suas próprias modificações.

III.8.2 - OASIS no contexto do Sistema StP

Facilidades oferecidas pelo sistema hospedeiro StP, principalmente por sua arquitetura aberta, permitem algumas facilidades, dentre as quais:

- a comunicação usuário/ferramentas é feita através da janela principal do StP, enquanto um conjunto de janelas, mostram determinadas funções das ferramentas;

- o conforto na utilização está garantido pelo paralelismo de execução das ferramentas com divisão de projeto, multi-utilização, múltiplas janelas, etc.;

- em StP, as ferramentas são homogêneas e complementares.

- abertura dos modelos convencionais suportados, permitindo a implementação do modelo OASIS sobre esses modelos e a

utilização direta das ferramentas existentes para a edição, geração do dicionário, manipulação e controle dos objetos;

- extensão prevista do esquema de dados inicial, considerando as especificações dos objetos do modelo OASIS;

- o aspecto genérico das interfaces com o usuário e das funções de edição e documentação dos objetos;

- são colocadas à disposição dos usuários, um conjunto de ferramentas básicas para a programação e integração de ferramentas complementares.

A seguir destacamos os principais objetivos de OASIS e como estes são atendidos através de StP:

- Coerência entre os objetos de uma simulação

A coerência entre o conjunto de objetos de uma simulação (objetos interdependentes tais como código-fonte, lista de compilação, código objeto executável, documentação, testes, resultados de execução, etc.), será assegurada por OASIS.

- Gerência do ciclo de evolução

A importância do tempo de vida de um software de simulação deve-se ao fato de que, a cada revisão, há uma mudança no número da versão para todos os objetos modificados após a última revisão. Através desse

procedimento, está garantida a atualização do software, completamente validado.

A partir de funções acessíveis em edições gráficas e diagramas, a gerência, o controle das modificações e a migração dos objetos sobre os diferentes níveis em OASIS, são executados pelo gerente encarregado da integração progressiva.

- Segurança no acesso aos elementos da simulação

Somente determinados objetos podem ser modificados, a partir de uma autorização fornecida a certos usuários.

- Integração na organização atual dos desenvolvimentos

Garantindo a coerência nas modificações, onde vários usuários podem ter acesso e, modificar uma simulação, OASIS facilita a integração final das modificações, a partir de:

- uma ferramenta de análise da estrutura dos objetos e de seu código fonte, integrada em StP, realiza os controles de coerência e completude das modificações. Os resultados da análise são registrados no dicionário de dados do nível correspondente à modificação;

- através do "browser" de StP, o gerente pode inspecionar os objetos modificados, verificando e validando as informações.

- Produção da documentação das simulações

É particularmente necessária a qualidade da documentação para a compreensão e manutenção das simulações. Esta é obtida através do sistema de preparação de documentos (DPS) de StP.

- Gerência de dados da simulação

A importância da gerência de dados de uma simulação é essencial para a compreensão dos modelos simulados, onde deverão estar presentes uma descrição detalhada a cada dado importante.

- Reutilização e integração de softwares externos

Existe a necessidade de transferência, em uma simulação, de módulos de códigos-fonte externos, feita através da ferramenta de análise, que importa esses módulos e os incorporam aos objetos OASIS que eles implementam na simulação corrente.

As seguintes ferramentas de StP são utilizadas, em OASIS:

- o editor de diagrama de dados: DSE ("Data Structure Editor");

- o editor de anotações de objetos: OAE ("Object Annotation Editor");

- o dicionário de dados: "Data Dictionary" e seu "browser";

- o sistema de preparação de documentos: DPS ("Document Preparation System") e seu "browser";

- o utilitário para gerência de versões: LOCK;

- o gerente do projeto e os repertórios de ferramentas: o Banco de Dados do projeto;

- a ferramenta VCC ("Version and Configuration Control").

III.8.3 - Aspectos específicos de OASIS

As funções específicas de OASIS referem-se aos modos de utilização do mesmo, definidos segundo a qualidade dos utilizadores e são relativas ao desenvolvimento dos programas FORTRAN e à gerência do ciclo de evolução.

O ambiente é adaptável e extensível o que permite sua evolução, já prevista, em várias direções. Em primeiro lugar prevê-se, para novas versões, a exploração de outras ferramentas de StP.

Estão previstas incorporações de aspectos que permitirão reutilização de software e engenharia reversa, que permitirão gerar diagramas a partir do código fonte.

III.8.4 - Conclusão

O uso das capacidades de adaptação de funções e das facilidades de integração de ferramentas de StP são largamente utilizadas no ambiente OASIS.

Estão previstas ferramentas e novas funções que poderão ser necessárias após a utilização pelos usuários, incluindo aspectos de reutilização e engenharia reversa.

III.9 - PATHCAL (WILANDER 1986)

Desenvolvido na Universidade de Linkoping, Suécia, o projeto PATHCAL teve como objetivo criar um ambiente de programação incremental para PASCAL.

Um ambiente de programação é composto de vários subsistemas, dentre os quais, um editor, um sistema depurador e uma "prettyprinter". Em PATHCAL, esses subsistemas estão acoplados para permitir a chamada de um sistema para outro. Todos os subsistemas são "procedures" em Pascal, embora alguns possam ter privilégios não permitidos na linguagem Pascal standard.

Existe a possibilidade do uso do sistema PATHCAL de forma similar à programação convencional, mas há ainda um conjunto de facilidades que não estão normalmente disponíveis nos sistemas convencionais. Estas são:

- <u>execução incremental</u>, que permite o uso de declarações da linguagem, fazendo-as executar diretamente. Isto propicia facilidade no aprendizado do sistema. Pode-se experimentar facilidades do sistema e da linguagem sem ter que escrever o programa completo.

- <u>sistema de uma linguagem</u>, ou seja, PATHCAL tem somente Pascal como linguagem de comando para os subsistemas;

- desenvolvimento incremental de programas, onde "procedures" podem ser desenvolvidas uma por vez, para formar programas, assim como construir e testar módulos de programas, separadamente, para mais tarde, combiná-los dentro do sistema. O programa pode ser desenvolvido segundo a preferência do programador, ou seja, "top-down", "bottom-up" ou "most critical portion first".

- modelo da secão terminal, ou seja, PATCHCAL mantém um modelo da sessão terminal e armazena informações sobre interações anteriores e seus resultados. Possibilita a comparação de resultados de um exemplo de teste, antes e depois da edição de uma "procedure".

- continuação após erros, dado que, quando um erro ocorre durante a execução do teste, o programa não será abortado. Em vez disso, ele é interrompido e, é permitido o uso de uma declaração. Se a "procedure" é editada durante uma interrupção, a nova versão será usada no futuro. Isso é especialmente adequado, por possibilitar a continuação após um erro, especialmente, em caso de programas interativos.

- editor de estrutura, permitindo que a edição do programa seja realizada em termos da linguagem de programação. Uma vantagem desse editor de estrutura é que somente aquelas partes que são realmente afetadas pela edição, são trocadas. Ocorre, então, a progamabilidade do editor.

III.9.1 - Descrição do sistema

PATHCAL, funciona como se o programa fosse executado, instrução por instrução e declaração por declaração, embora com a diferença de que não é necessário obedecer uma ordenação rígida entre instruções e declarações. Instruções são guardadas em um bloco especial, denominado "bloco do sistema", que armazena, ainda, diferentes declarações, incluindo todas as "procedures", tipos e variáveis fornecidas pelos usuários.

Contém ainda todos os subsistemas necessários para a utilização do sistema.

As instruções são imediatamente executadas e, inexiste uma diferença entre instruções e declarações, pois ambas são executadas.

Se comparado com os ambientes convencionais de programação, PATHCAL difere nos seguintes aspectos:

- enquanto nos ambientes convencionais o ciclo de trabalho é composto de edição, compilação e execução, utilizando um editor de textos de caráter geral, em PATHCAL, o programa é testado, por partes, com recursos para a construção da base de dados da "procedure", para combiná-la nos programas.

- a edição tradicional é, normalmente efetuada por um editor de textos de caráter geral. Em PATHCAL, o editor utilizado reconhece símbolos em Pascal, suas declarações e diversas estruturas de programas.

- de uma forma geral, um compilador tradicional assemelha-se a uma "caixa preta". Em PATHCAL, a compilação é interrompida a cada vez que um erro sintático é encontrado, possibilitando seu acerto pelo programador, que pode editar o fonte. Isto significa que não haverá uma cascata de mensagens de erro, quando ocorrer, um único erro. Na grande maioria dos sistemas de depuração interativos, o valor das variáveis podem ser lidos e alterados. Isto, entretanto, não é normalmente possível para chamar "procedures" no programa ou para entrar com novas declarações. Em PATHCAL, todas as operações são permitidas num possível ponto de ruptura ou interrupção por erro, incluindo um resumo da execução do programa. É possível chamar qualquer "procedure", criar novas declarações e editar a execução do programa.

- o suporte a testes de programas raramente está presente nos sistemas tradicionais. Testar em PATHCAL permite a comparação interativa ou programada dos resultados de teste. A verificação dos testes pode ser reforçada através de testes especiais das "procedures" que são acrescentados às declarações no código. Isto pode ser visto com uma

precondição programada. Esta abordagem encoraja a construção de bibliotecas de testes e verificadores para os programas.

III.9.2 - Desvantagens do enfoque usado em PATHCAL

1. O sistema PATHCAL assume que um programa, após seu desenvolvimento ou manutenção, é movido para um ambiente de produção. Isso levanta a questão da compatibilidade entre o sistema PATHCAL e seu compilador de produção.

2. Eficiência pode ser um fator crítico mesmo durante a fase de teste do programa. Isso pode ser parcialmente controlado através de compiladores especializados dentro da estrutura de PATHCAL.

III.9.3 - Conclusão

PATHCAL demonstra que é possível e razoável a construção de um sistema de programação incremental para a linguagem Pascal, que somente estavam disponíveis, anteriormente, para as linguagens Lisp e APL.

III.10 - ASPIS (PUNCELLO et alli, 1988)

ASPIS (Application Software Prototype Implementation System), projeto 401 ESPRIT, foi desenvolvido no Laboratório Tecsiel na Itália.

O projeto objetiva atingir, através da utilização de um modelo evolutivo, maior flexibilidade e efetividade no ciclo de vida, tornando mais suave a transição entre suas fases iniciais, (análise de requisitos e projeto).

Pretende, ainda, através de técnicas de Inteligência Artificial e de um conjunto de ferramentas especialmente dirigidas a essas fases, a obtenção de maior qualidade do produto e produtividade no desenvolvimento. Nesse ponto, é considerada a habilidade dos especialistas na área de aplicação, uma vez que são fases intensivas em conhecimento.

Foi adotado no ambiente, um modelo evolucionário de ciclo de vida, visando cobrir o "gap" existente entre as tarefas de análise e projeto.

Dentre os aspectos inovadores do ambiente ASPIS tem-se:

(a) ferramentas baseadas em conhecimento, denominadas <u>assistentes</u>:

(b) definição de um formalismo baseado em lógica para especificações.

III.10.1 - Ciclo de Vida de ASPIS

Visando atenuar o aspecto crítico da passagem da Análise para Projeto foram escolhidos dois métodos (um para análise e outro para projeto), que foram modificados e então definida a melhor forma de ser feita a passagem de uma fase para outra.

O Método para Análise selecionado compreende várias fases, utilizando SADT para analisar funções e esquemas de entidades-relacionamentos para representar dados. Dada a informalidade de SADT foi definido um formalismo, a Lógica de Suporte ao Raciocínio *(RSL - Reasoning Support Logic)*, que aumenta os diagramas SADT e permite verificar a correção e consistência das especificações. RSL permite especificar as propriedades do sistema através de um conjunto de axiomas numa linguagem baseada em lógica. Estes axiomas são transformados em código Prolog para serem executados.

O processo de Projeto foi dividido em duas fases, projeto do sistema e projeto de software. O projeto do sistema busca descrever o sistema em termos de sua arquitetura global, envolvendo hardware e software. O

projeto de software busca descrever as funções do software. em termos de processos e transições. Nos dois casos foi adotada uma estratégia top-down.

O ciclo de desenvolvimento de ASPIS é apresentado na Figura 13.

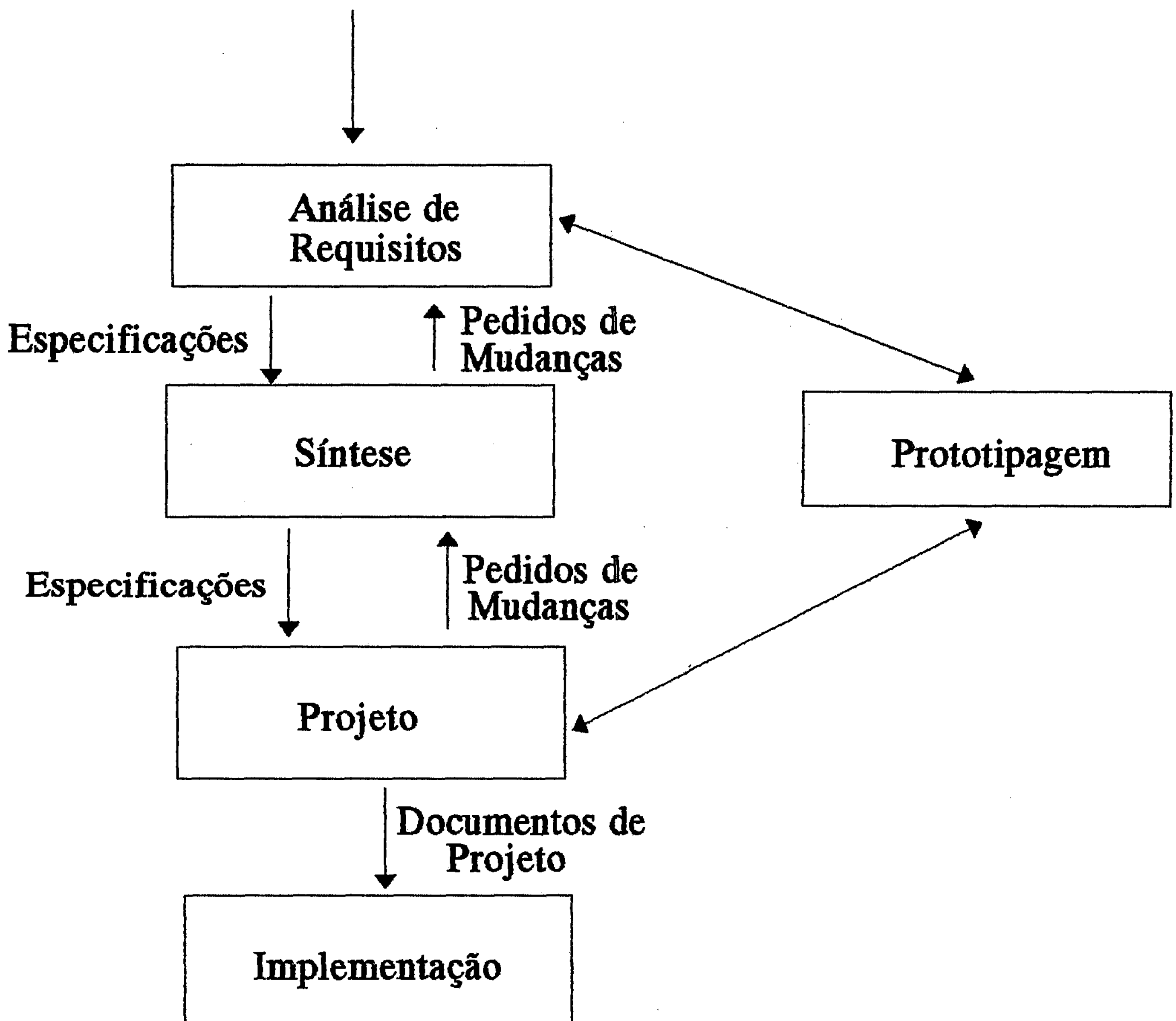


Figura 13: O ciclo de desenvolvimento de ASPIS
Fonte: PUNCELLO 88

Em função da possibilidade de ocorrência de mudanças, nas especificações, tanto por necessidades de aperfeiçoamento tanto por alterações nos requisitos dos usuários e, da conseqüente necessidade de alteração no projeto, o ciclo de vida ASPIS fornece suporte à interação entre projetistas e analistas, através de um processo denominado **síntese**. Neste processo a informação é obtida da fase de análise e tornada disponivel na forma adequada à fase de projeto, possibilitando então a comunicação entre o analista e o projetista. Trata-se de um processo contínuo que também inclui a verificação da consistência entre a análise e o projeto.

III.10.2 - Assistentes Baseados em Conhecimento

ASPIS tem dois assistentes baseados em conhecimento: o Assistente de Análise e o Assistente de Projeto. Estes assistentes contém conhecimento tanto do método utilizado na fase de análise e projeto (conhecimento metodológico) quanto da área de aplicação do sistema que está sendo desenvolvido (conhecimento do domínio).

O conhecimento metodológico é classificado como:

- conhecimento sobre a sintaxe do método, que permite ao usuário questionar sobre os "links" que conectam as fases do método;

- conhecimento sobre os critérios do método, que é incluído para informar ao usuário os procedimentos para realizar cada etapa de uma fase, e,

- conhecimento sobre heurísticas independentes do domínio de aplicação que é a experiência ganha utilizando o método e que permite que analistas não experientes ajam como especialistas na utilização do método.

As heurísticas de maior utilidade são entretanto as relacionadas ao domínio de aplicação. O conhecimento do domínio é uma especialização do conhecimento metodológico e permite verificar, em alguns casos, a adequação dos documentos gerados com relação a alguns conceitos gerais do domínio.

A interface com o usuário permite que este decida o nível de interação e o tipo de ajuda que necessita.

III.10.3 - Assistentes de Suporte

ASPIS possui, ainda, dois assistentes de suporte: o Assistente para Prototipagem e o Assistente para Reutilização.

O Assistente para Prototipagem utiliza informações sobre as propriedades formais, para executar a especificação. Para isto tem-se um tradutor, um executor e um módulo de interface. O tradutor toma os axiomas RSL como entrada e produz clausulas PROLOG que são interpretadas pelo executor. Como os axiomas RSL definem as propriedades semânticas dos componentes de um modelo SADT, no momento da execução é mostrado o comportamento do sistema. Desta forma pode-se validar as especificações com relação às necessidades dos usuários.

O Assistente para Reutilização tem como objetivo auxiliar os usuários ASPIS, na reutilização de especificações e projetos. O enfoque adotado em ASPIS está fortemente baseado nos trabalhos de Freeman e Prieto-Diaz.

A figura 14 dá uma visão geral do ambiente ASPIS.

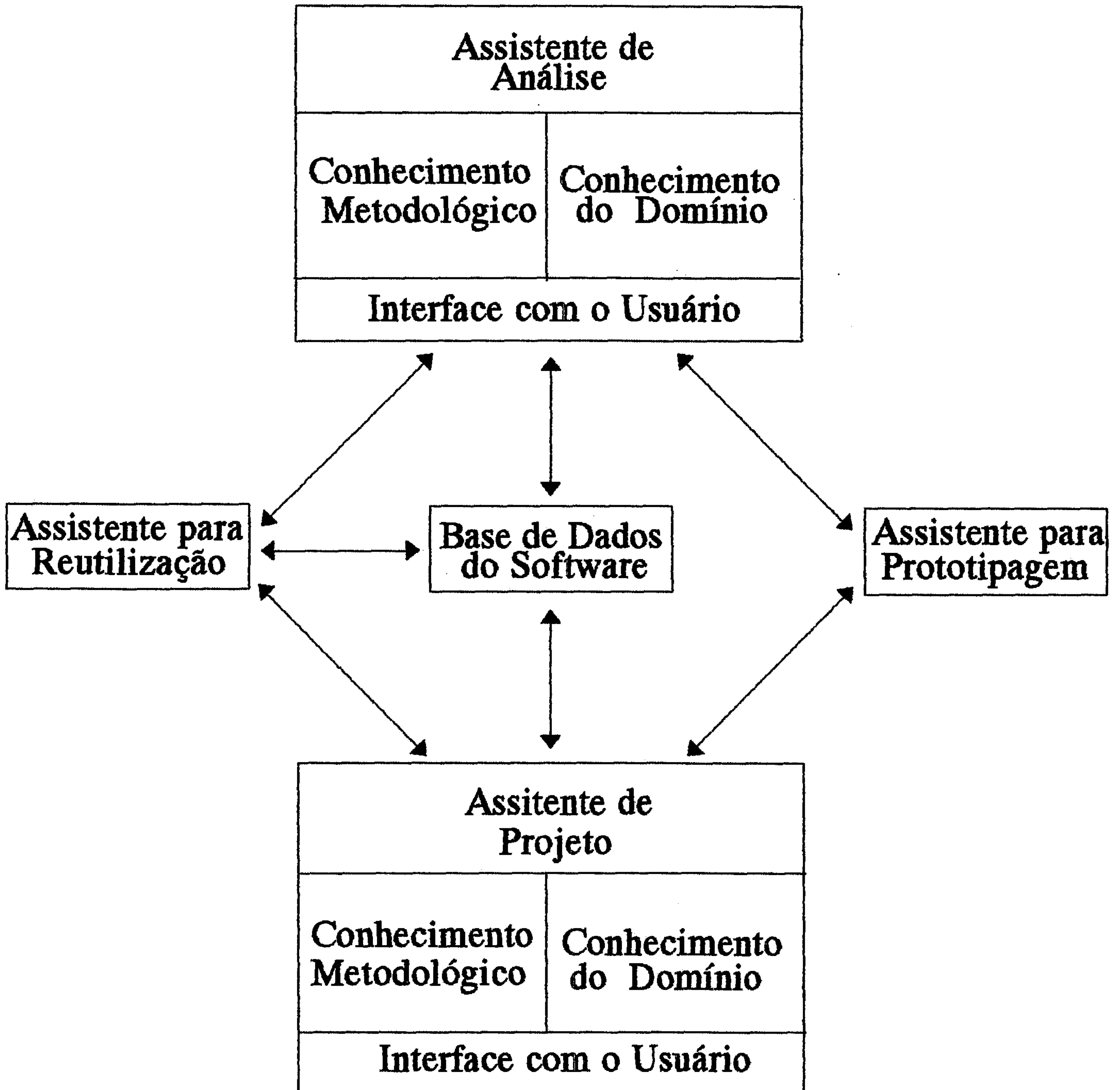


Figura 14: Visão Geral do Ambiente ASPIS
Fonte: PUNCELLO 88

III.10.3 - Conclusão

O projeto foi desenvolvido em duas fases, compreendendo um ano para pesquisas e três anos de desenvolvimento. Segundo os autores, os maiores esforços concentraram-se na definição do ciclo de vida ASPIS, na definição da linguagem de especificação formal, na pesquisa das características dos assistentes e no projeto dos assistentes baseados em conhecimento.

A implementação foi feita em estações SUN com UNIX 4.2 BSD.

CAPÍTULO IV

UMA TAXONOMIA PARA AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE

Este capítulo propõe uma taxonomia para ambientes de desenvolvimento de software que permite classificar os ambientes atualmente disponíveis considerando, também, as tendências na área para os próximos anos.

Esta taxonomia foi elaborada tendo como base o estudo da literatura descrito nos capítulos II e III, bem como o histórico da evolução da Engenharia de Software e as tendências futuras descritas a seguir.

IV.1 - Histórico e Tendências da Engenharia de Software

De acordo com o glossário de termos de Engenharia de Software do IEEE, Engenharia de Software é "o enfoque sistemático para o desenvolvimento, operação, manutenção e descontinuação de software".

A Engenharia de Software desde seu surgimento em 1968 teve uma longa história de evolução. O progresso obtido pode ser caracterizado de diferentes maneiras. Uma forma

bastante familiar é caracterizar este progresso a partir da evolução da programação de forma artesanal ("programming-any-which-way"), para a programação em pequena escala (programming-in-the-small") e desta para a programação em larga escala (programming-in-the-large).

Antes da metade dos anos 60 (SHAW 1990) a programação foi feita fundamentalmente de forma artesanal, de maneira empírica e dependente das características pessoais do desenvolvedor.

Ao redor dos anos 70 tem lugar as primeiras tentativas de transformar o processo de desenvolvimento de programas em ciência. São desta época os trabalhos sobre algoritmos e estrutura de dados de Knuth e o trabalho de Floyd sobre verificação de programas. A partir da metade dos anos 70 começa a evolução da programação em pequena escala para a programação em larga escala, onde as preocupações de pesquisa se dirigem a sistemas complexos, onde passam a ter importância aspectos de interface e gerência.

No que se refere a métodos e técnicas, estas acompanham a evolução da programação em pequena para a programação em larga escala. O início dos anos 70 (LEWIS 1990) é marcado pela definição do que seriam boas práticas de programação, pelo advento da programação estruturada e pelo reconhecimento das vantagens de projeto "top-down",

refinamentos sucessivos e modularidade. Os anos 1976/1977 marcam o aparecimento das primeiras técnicas de especificação e projeto. Os anos 80 são marcados pelas ferramentas CASE e Estações de Trabalho de Engenharia de Software "stand alone". Para os anos 90 espera-se que vejamos a era da aplicação de técnicas de Inteligência Artificial à Engenharia de Software e o uso de Estações de Trabalho que permitam apoio ao trabalho cooperativo.

Um workshop realizado nos Estados Unidos em 1990 pelo "Computer Science and Technology Board" (CSTB 1990) reuniu pesquisadores de universidades e do âmbito empresarial com o objetivo de estabelecer prioridades de pesquisa para esta década. Alguns aspectos foram especialmente destacados, por sua relevância:

- a necessidade de que pesquisadores tomem conhecimento do que acontece na prática, para desta forma encontrarem bons problemas para pesquisa e terem possibilidade de validar os resultados obtidos;

- a necessidade do uso mais rigoroso de técnicas matemáticas, através de métodos formais para desenvolvimento de software;

- a necessidade de se ter em conta, no projeto de sistemas de software, que um sistema completo inclui pessoas além de software e hardware;

- necessidade de reconhecer a existência de mudanças, como inerente ao software, e já considerar, no processo de desenvolvimento de software, a possibilidade de futuras manutenções;

- necessidade de analisar os sistemas atualmente existentes e o que provocou seu sucesso ou fracasso, pois o estudo sistemático destes sistemas possibilitará o entendimento de como se está desenvolvendo software, atualmente, e das características do software desenvolvido utilizando-se as técnicas atuais;

- necessidade de serem estabelecidos processos de avaliação da qualidade de software, de forma que estas medidas possibilitem a avaliação do impacto de novos métodos e técnicas na qualidade do produto;

- necessidade da construção de um modelo unificado do processo de desenvolvimento de software e de linguagens adequadas às diferentes etapas deste processo (diferentes níveis de abstração) e a diferentes domínios de aplicação, bem como dos mecanismos para transição entre os vários níveis de abstração;

- necessidade de entender o desenvolvimento de software como um exercício de colaboração e a necessidade de maximizar sua efetividade através do desenvolvimento de ferramentas para apoio ao trabalho cooperativo, essenciais para o aumento da produtividade no desenvolvimento;

- necessidade de desenvolvimento de métodos e ferramentas que facilitem o processo de aquisição e representação de requisitos, favorecendo a interação com os usuários;

- necessidade de reutilizar e codificar o conhecimento;

- necessidade de interação com outras áreas, tais como ciências do comportamento e administração que podem contribuir para o aprimoramento de práticas de Engenharia de Software;

- necessidade do desenvolvimento de ferramentas de apoio ao desenvolvimento de software, especialmente de ferramentas inteligentes.

Cabe destacarmos, aqui, o apoio ao trabalho cooperativo. Colaboração é um sinal de maturidade. O

desenvolvimento de sistemas complexos exige cada vez mais o trabalho de grandes equipes atuando em paralelo e cada vez se vê maiores perspectivas para o apoio computadorizado ao desenvolvimento de software nesse contexto.

O desenvolvimento de ambientes que apóiem a colaboração exige um maior entendimento do impacto de tecnologias sobre os grupos e organizações. Assim sendo, um elemento fundamental da pesquisa para se chegar a estes ambientes é sobre o comportamento de indivíduos e grupos e suas formas de interação.

Outro aspecto a ser considerado, no contexto de desenvolvimento cooperativo de software, é a criação de espaço adequado para propiciar uma cooperação ativa entre usuários e projetistas de software. Trata-se de um enfoque para desenvolvimento de software atualmente conhecido como projeto participativo.

Para enfatizar este enfoque de cooperação entre usuários e projetistas nas primeiras atividades do projeto, Kyng (KYNG 1991) propõe que o termo "análise" seja substituído por "aprendizado mútuo". Este termo implica que os projetistas aprendem sobre a área de aplicação e os

usuários sobre novas possibilidades técnicas. A idéia chave é mover o foco de discussões de descrições do sistema para ações cooperativas.

Outras formas de cooperação poderão estar presentes ao longo do processo de desenvolvimento e estas deverão ser adequadas à área de aplicação em questão.

IV.2 - Uma Taxonomia para Ambientes de Desenvolvimento de Software

IV.2.1 - Classificações

Classificar (Werneck 1990) é o ato de agrupar objetos semelhantes, sendo que todos os membros de um grupo ou classe produzidos por uma classificação têm pelo menos uma característica que outro membro de outra classe não tem.

Classificação mostra as relações entre objetos e classes de objetos, sendo uma ferramenta de grande importância na organização do conhecimento.

As relações de um esquema classificatório podem ser expressas de forma hierárquica ou sintética. Relações

hierárquicas se fundamentam no princípio de subordinação ou inclusão enquanto que o relacionamento sintético é criado sob dois ou mais conceitos pertencentes a diferentes hierarquias.

Classificações podem ser organizadas de forma enumerativa ou tendo por base algum aspecto particular. No esquema enumerativo o universo de conhecimentos é postulado com a divisão sucessiva, em classes descritivas que incluem todas as possibilidades de classes compostas. O método de classificação a partir da consideração de algum aspecto particular é usado, principalmente, na biblioteconomia, sintetizando relações de tópicos de documentos ou objetos específicos.

Esquemas que consideram algum aspecto específico são mais flexíveis e precisos podendo ser melhor ajustados para coleções grandes e de expansão contínua.

IV.2.2 - Classificação de Ambientes

A classificação que propomos para Ambientes de Desenvolvimento de Software está baseada em quatro enfoques:

a) Classificação Baseada na Arquitetura

b) Classificação Baseada no Suporte Oferecido ao Usuário

c) Classificação Baseada na Escala

d) Classificação Baseada em Características Gerais de Ambientes

A seguir descreveremos, com detalhe, as bases para cada um desses enfoques que adotamos para classificar Ambientes.

- Classificação baseada na Arquitetura dos Ambientes

A consideração da arquitetura na classificação de Ambientes de Desenvolvimento de Software está baseada nos trabalhos de Penedo (PENEDO 1988), Lucena (LUCENA 1987) e Dart (DART 1987). Esta classificação também está inspirada nos conceitos de programação em pequena escala e programação em larga escala, bem como nas tendências que observamos para os novos Ambientes.

Segundo este enfoque classificamos Ambientes de Desenvolvimento de Software em:

Ambientes Centrados em Sistemas Operacionais;

Ambientes Centrados em Linguagens de Programação;

Ambientes Centrados em Métodos;

Ambientes Centrados em Configuração, e,

Ambientes Centrados em Trabalho Cooperativo.

A classificação de Ambientes segundo sua arquitetura considerando a característica de serem centrados em Sistemas Operacionais, centrados em Linguagens de Programação e centrados em Métodos segue as propostas de Penedo (PENEDO 1988), Dart (DART 1987) e Lucena (LUCENA 1987).

A evolução que observamos no estado da arte nos leva a estender esta classificação acrescentando outras duas classes: os **ambientes centrados em configuração** e **ambientes centrados em trabalho cooperativo.**

Ambientes Centrados em Configuração. caracterizam-se pelo suporte oferecido à configuração do ambiente para permitir o desenvolvimento segundo diferentes modelos de ciclo de vida e a utilização dos métodos e ferramentas mais adequados às características dos diferentes domínios de aplicação, projetos e equipes desenvolvedoras. Assim, em detrimento aos ambientes de propósito geral, esses ambientes vêm como resposta à tendência atual de construção

de ambientes para aplicações específicas, que forneçam suporte tanto à utilização de diferentes métodos quanto aos diferentes níveis de competência dos usuários.

Ambientes Centrados em Trabalho Cooperativo, têm sua arquitetura proposta de forma a prover facilidades para o apoio computadorizado ao trabalho cooperativo onde se enfatiza a colaboração entre os diferentes grupos de desenvolvedores e entre desenvolvedores e usuários. Várias formas de cooperação podem estar presentes ao longo do processo de desenvolvimento e devem ter apoio do ambiente. Exemplos possíveis são o suporte a reuniões de inspeção através de salas de reunião adequadamente equipadas com computadores e software, ferramentas para edição colaborativa de documentos, etc. Para prover estas facilidades, ambientes centrados em trabalho cooperativo tem características específicas e complexas tanto do ponto de vista de hardware quanto de software.

Assim sendo, ao classificar ambientes segundo sua arquitetura consideramos cinco classes:

1. **Ambientes Baseados em Sistemas Operacionais**, são aqueles ambientes cuja arquitetura sofre influência marcante da tecnologia dos Sistemas Operacionais, onde o controle do usuário sobre o ambiente é efetuado através de uma linguagem de comando:

2. **Ambientes Centrados em Linguagens de Programação** são ambientes cuja arquitetura objetiva fornecer suporte a todas as fases do ciclo de vida através de uma linguagem de programação particular, com um conjunto de ferramentas adequadas a esta linguagem;

3. **Ambientes Centrados em Métodos** são os ambientes cuja arquitetura privilegia um ou mais métodos para o desenvolvimento de software e o gerenciamento do processo de desenvolvimento, objetivando, ainda, a compatibilidade entre os métodos utilizados;

4. **Ambientes Centrados em Configuração**, são os ambientes que se caracterizam pela possibilidade de serem configurados para atender a diferentes características de domínios de aplicação, projetos e usuários;

5. **Ambientes Centrados em Trabalho Cooperativo** são os ambientes cuja arquitetura suporta o apoio computadorizado ao desenvolvimento cooperativo de software.

- Classificação baseada no Suporte Oferecido ao Usuário

Nesta classificação consideramos o suporte oferecido ao usuário do Ambiente para realização das três atividades fundamentais presentes no processo de desenvolvimento de

software: construção do produto, controle da qualidade do produto e gerência do processo de desenvolvimento (ROCHA 1987).

Esta classificação está inspirada, ainda, nos trabalhos de Lubars (LUBARS 1989), Takahashi (TAKAHASHI 1989) e Penedo (PENEDO 1988).

Assim sendo, ao classificar ambientes consideramos o apoio oferecido à:

Construção do Produto

- Suporte a todo o ciclo de vida;
- Suporte para atividades clericais;
- Suporte para reutilização de componentes anteriores;
- Inteligência localizada, isto é, inclusão de inteligência nas ferramentas;
- Suporte inteligente para assistir na exploração de possibilidades e atividades criativas
- Suporte para re-engenharia.

Avaliação da Qualidade do Produto ao longo do desenvolvimento

- Suporte para estabelecimento da meta de qualidade;

- Suporte para avaliação da qualidade ao longo do desenvolvimento (avaliação de especificações de requisitos, avaliação de especificações de projeto e avaliação de programas);
- Suporte para testes;
- Suporte para estimação e medição da confiabilidade do software.

Gerência do Processo de Desenvolvimento

- Suporte para Planejamento;
- Suporte para Acompanhamento;
- Suporte para Gerência de configuração.

- Classificação Baseada na Escala

Esta classificação, que propomos, é uma extensão do trabalho de Perry e Kaiser (PERRY 1988), que ao definirem uma taxonomia para Ambientes de Desenvolvimento de Software fazem uma analogia sociológica e consideram quatro tipos de ambientes: o modelo Indivíduo, o modelo Família, o modelo Cidade e o modelo Estado, conforme descrevemos na seção II.2.2.

A evolução que observamos no estado da arte em ambientes, nos leva a estender esta classificação de forma

a incluir uma classe que contemple as particularidades dos ambientes capazes de integrar ferramentas desenvolvidas em outros contextos. Assim sendo, propomos uma nova classe - **o modelo Comunidade de Estados.**

Neste modelo a analogia que fazemos é com as comunidades de nações. Atualmente as nações (Estados) se unem em comunidades, onde cada vez as fronteiras são menos nítidas, buscando uma maior integração e cooperação. A exemplo das Comunidades de Estados (por exemplo, a Comunidade Econômica Européia ou o Mercosul) um Estado, para tornar vendáveis seus produtos em outro Estado, busca que estes tenham padrões de qualidade que tornem possível sua assimilação em outros contextos. Isso leva a uma busca sobretudo de padronização nos produtos. A exemplo do que se viu com a Comunidade Européia, esta integração tende a se tornar cada vez mais forte, com uma diminuição progressiva das fronteiras entre as nações.

Este é o contexto dos ambientes caracterizados dentro do modelo Comunidade de Estados que propomos. Aqui, de uma forma análoga ao que Tsichritzis (TSICHRITZIS 1990) se refere ao falar de "comunidades de software", tratando do tema da reutilização de componentes em sistemas de informação, busca-se reutilização de componentes (no caso ferramentas) em Ambientes de Desenvolvimento de Software.

Estes ambientes, permitem a integração de ferramentas desenvolvidas em diferentes contextos. ("off the shelf").

Nesta classificação consideramos, portanto, os seguintes modelos:

- o **Modelo Individuo**, onde o ambiente oferece um conjunto mínimo de ferramentas necessárias para construção do software. Estão incluídos nesta classe os, habitualmente chamados, ambientes de programação.

- o **Modelo Família**, inclui os ambientes que além das facilidades oferecidas no modelo individuo oferecem outras que dão suporte às interações de um pequeno grupo de programadores.

- o **Modelo Cidade**, onde as facilidades de gerenciamento do modelo família são estendidas de forma a poder apoiar um grupo maior de desenvolvedores.

- o **Modelo Estado**, onde se provê um ambiente genérico para o desenvolvimento de software numa empresa e que pode ser instanciado para atender às necessidades de projetos particulares:

- o **Modelo Comunidade de Estados**, que inclui os ambientes capazes de integrar ferramentas desenvolvidas "off the shelf".

- Classificação Baseada em Características Gerais de Ambientes

Ao classificar segundo as características gerais do ambiente, consideramos aqueles atributos não contemplados nas classificações anteriores e que nos parecem de importância ao analisar ambientes de desenvolvimento de software. Nesta classificação consideramos, portanto, os seguintes atributos:

- Facilidade de uso (interface amigável);
- Facilidade de aprendizado;
- Suporte a usuários inexperientes (ferramentas cognitivas);
- Grau de automação
- Valor (relação custo/benefício) do ambiente
- Inteligência global (controle geral das operações realizadas no Ambiente).

A figura 15 apresenta uma visão geral da Taxonomia de Ambientes de Desenvolvimento de Software proposta.

SEGUNDO	CLASSIFICAÇÃO DE AMBIENTES DE DESENVOLVIMENTO DE SOFTWARE	
ARQUITETURA	- Amb. Cent. em Sist. Operacionais - Amb. Cent. em Ling. de Programação - Amb. Cent. em Métodos - Amb. Cent. em Configuração - Amb. Cent. em Trabalho Cooperativo	
SUPORTE OFERECIDO AO USUÁRIO	**CONSTRUÇÃO**	- Sup. a todo Ciclo de Vida - Sup. p/Atividades Clericais - Sup. p/ Reutil. de Comp. Anteriores - Sup. Inteligente p/Assistir na Exploração de Novas Possibilidades e Atividades Criativas - Inteligência Localizada - Sup. p/Re-Engenharia
	AVALIAÇÃO	- Sup. p/Avaliação da Qualidade ao longo do desenvolvimento - Sup. p/Testes - Sup. p/Estimação e Medição da Confiabilidade
	GERÊNCIA	- Sup. p/Planejamento - Sup. p/Acompanhamento - Sup. p/Gerência da Configuração
ESCALA	- Modelo Indivíduo - Modelo Família - Modelo Cidade - Modelo Estado - Modelo Comunidade de Estados	
CARACTERÍSTICAS GERAIS	- Facilidade de uso - Facilidade de aprendizado - Suporte a usuários inexperientes - Grau de automação - Valor do ambiente - Inteligência global	

Figura 15: Taxonomia de Ambientes de Desenvolvimento de Software

IV.3 Classificação dos Ambientes no Contexto da Taxonomia Proposta

Nesta seção classificamos os ambientes de desenvolvimento de software descritos no capítulo III de acordo com a taxonomia proposta na seção anterior:

- Classificação quanto à Arquitetura do Ambiente

- Ambientes Centrados em Sistemas Operacionais:

Nenhum dos ambientes estudados pode ser caracterizado como centrado em Sistemas Operacionais

- Ambientes Centrados em Linguagens de Programação:

- PATHCAL

- Ambientes Centrados em Métodos:

- INSCAPE
- IDeA
- PEGASE
- OASIS
- ASPIS

- **Ambientes Centrados em Configuração:**
    - ARCADIA
    - ADAGE
    - CENTAUR
    - GRASPIN

- **Ambientes Centrados em Trabalho Cooperativo:**

Nenhum dos ambientes estudados pode ser caracterizado como centrado em Trabalho Cooperativo

- **Classificação quanto ao Suporte Oferecido ao Usuário**

Construção do Produto

- **Suporte a todo o ciclo de vida;**
    - INSCAPE
    - ARCADIA
    - ADAGE
    - CENTAUR
    - PEGASE
    - GRASPIN
    - ASPIS

- **Suporte para atividades clericais;**
  - INSCAPE
  - IDeA
  - ADAGE
  - CENTAUR
  - PEGASE
  - GRASPIN
  - OASIS
  - PATHCAL

- **Suporte para reutilização de componentes anteriores;**
  - INSCAPE
  - IDeA
  - ARCADIA
  - ADAGE
  - OASIS
  - PATHCAL
  - ASPIS

- **Inteligência localizada, isto é, inclusão de inteligência nas ferramentas;**
  - INSCAPE
  - ASPIS

- Suporte inteligente para assistir na exploração de possibilidades e atividades criativas
  - ARCADIA
  - GRASPIN

- Suporte para re-engenharia
  - ADAGE
  - OASIS

Avaliação da Qualidade do Produto ao longo do desenvolvimento

- Suporte para estabelecimento da meta de qualidade;
  - IDeA
  - ARCADIA

- Suporte para avaliação da qualidade ao longo do desenvolvimento (avaliação de especificações de requisitos, avaliação de especificações de projeto e avaliação de programas);
  - INSCAPE
  - CENTAUR
  - PEGASE
  - GRASPIN

- PATHCAL
- ASPIS

- **Suporte para testes;**
  - INSCAPE
  - ARCADIA
  - OASIS
  - PATHCAL

- **Suporte para estimação e medição da confiabilidade do software.**

Nenhum dos ambientes estudados possui esta característica.

Gerência do Processo de Desenvolvimento

- **Suporte para Planejamento;**
  - INSCAPE

- **Suporte para Acompanhamento;**
  - INSCAPE
  - ARCADIA

- **Suporte para Gerência de Configuração.**
    - ADAGE
    - PEGASE
    - OASIS

- **Classificação quanto à Escala**

- **Modelo Indivíduo:**
    - CENTAUR
    - PATHCAL

- **Modelo Família:**

Nenhum dos ambientes estudados pode ser caracterizado como pertencente ao Modelo Família.

- **Modelo Cidade:**
    - INSCAPE
    - PEGASE
    - OASIS

- **Modelo Estado:**

Nenhum dos ambientes estudados pode ser caracterizado como pertencente ao Modelo Estado.

- Modelo Comunidade de Estados:
    - ARCADIA

- Classificação quanto às Características Gerais:

- Facilidade de uso (interface amigável):
    - INSCAPE
    - IDeA
    - ARCADIA
    - ADAGE
    - CENTAUR
    - GRASPIN
    - OASIS
    - PATHCAL
    - ASPIS

- Facilidade de aprendizado:
    - PATHCAL

- Suporte a usuários inexperientes (ferramentas cognitivas):
    - ADAGE

- Grau de automação:
    - INSCAPE
    - IDeA

- ADAGE
- PATHCAL

- **Valor (relação custo/benefício) do ambiente:**
  - INSCAPE

- **Inteligência global (controle geral das operações realizadas no Ambiente):**
  - IDeA
  - CENTAUR

IV.4 - Classificação da Estação TABA segundo a Taxonomia Proposta

Nesta seção descrevemos a Estação TABA e a classificamos de acordo com a taxonomia proposta.

IV.4.1 Descrição da estação TABA

Uma solução para aumentar a qualidade e a produtividade no desenvolvimento de produtos de software está na utilização de um Ambiente de Desenvolvimento de Software construído a partir do conhecimento das características do software a ser desenvolvido. De acordo

com Tom de Marco (DE MARCO 1984) o ideal seria definir um ambiente para cada área de aplicação ou até mesmo para cada projeto.

No contexto do Projeto TABA, considera-se que um Ambiente de Desenvolvimento de Software é composto por métodos, instrumentos e ferramentas que são utilizados ao longo do ciclo de vida do software. O ciclo de vida define todas as fases do processo de desenvolvimento e as atividades que devem ser realizadas em cada fase. Métodos são utilizados de forma a organizar o modo como as pessoas pensam e a maneira como o usuário trabalha ao longo do processo de desenvolvimento. Os instrumentos tornam possível a utilização de métodos. E, por fim, as ferramentas são os instrumentos com recursos que permitem seu uso em computadores.

A experiência do grupo envolvido no Projeto TABA em desenvolvimento de software para diferentes áreas (Engenharia Civil, Geofísica, Engenharia Oceânica, Física de Altas Energias, Educação, Sistemas Baseados em Conhecimento e ferramentas CASE) foi a motivação para o Projeto TABA. O objetivo do projeto é a construção de uma estação de trabalho para desenvolvimento de software, configurável, que suporta desde a criação de Ambientes de Desenvolvimento de Software que considerem as

características dos sistemas a serem desenvolvidos até a própria execução destes sistemas.

O nome TABA - em língua tupi, aldeia indígena - foi escolhido por analogia. A taba indígena agrega várias ocas e nelas se desenrola toda a vida da comunidade. Da mesma forma a estação TABA é composta por um conjunto de recursos que suportam todo o processo de desenvolvimento e execução do software.

A Estação TABA tem quatro funções:

a) auxiliar o engenheiro de software no planejamento e criação do ambiente mais adequado ao desenvolvimento de um produto específico;
b) auxiliar o engenheiro de software a implementar as ferramentas necessárias ao ambiente definido em (a);
c) permitir aos desenvolvedores do produto (software) o uso da estação através do ambiente especificado em (a) e gerado em (b);
d) permitir a execução do software, dado que, eventualmente, o software produto poderá ser executado na própria estação para ele configurada (o que é sempre verdade pelo menos na fase de testes).

Estas funções determinam quatro ambientes na Estação TABA:

a) Ambiente Gerador de Ambientes (Meta-ambiente)

A primeira função da estação de trabalho é fornecer o Ambiente de Desenvolvimento de Software adequado para uma determinada aplicação. Esta função é realizada pelo ambiente gerador de ambientes (meta-ambiente).

b) Ambiente Gerador de Ferramentas

Este ambiente auxilia na construção de novas ferramentas sugeridas com o apoio de XAMÃ, o planejador de ambientes da Estação TABA. Após construídas, estas ferramentas passarão a fazer parte do repertório do meta-ambiente para uso em futuros ambientes.

Desenvolver ferramentas automatizadas é, na realidade, um caso particular de desenvolvimento de software. Este ambiente é uma das instâncias do Ambiente de Desenvolvimento de Software descrito em (c) com especificidades por sua interação com o meta-ambiente.

c) Ambiente de Desenvolvimento de Software (ADS)

A partir de (a) e (b) tem-se o ADS específico a ser utilizado no desenvolvimento de uma determinada aplicação.

d) Ambiente de Teste e/ou Execução da Aplicação

Este ambiente permite executar o software desenvolvido em (c), na própria estação.

O meta-ambiente tem cinco funções principais:

- alimentar a base de conhecimentos da Estação;
- alimentar a base de ferramentas da Estação;
- sugerir ferramentas para implementação;
- planejar o Ambiente de Desenvolvimento de Software mais adequado para o desenvolvimento de uma determinada aplicação;
- instanciar o Ambiente de Desenvolvimento de Software especificado.

A função "Alimentar Base de Conhecimentos" só pode ser realizada pelo meta-usuário Engenheiro de Software. Este meta-usuário é o único que pode incluir novos conhecimentos na base de conhecimentos da Estação TABA. O conhecimento incluído visa levar à indicação de elementos que juntos formem um ambiente.

A função "Alimentar Base de Ferramentas" só pode ser realizada pelo meta-usuário Engenheiro de Software. Esta função é realizada para se incluir na base de ferramentas TABA uma nova ferramenta que foi desenvolvida fora da Estação.

A função "Sugerir Ferramentas para Implementação", também, só pode ser realizada pelo meta-usuário Engenheiro de Software. Este meta-usuário da Estação TABA pode acrescentar ao depósito "Ferramentas para Implementar", a sugestão de novas ferramentas que a seu ver são candidatas a implementação pelo Ambiente Gerador de Ferramentas.

O trabalho de Aguiar (AGUIAR 1992) objetivou fornecer uma solução que permitisse a tomada de decisões por aquele que pretendesse o planejamento de um Ambiente de Desenvolvimento de Software na Estação TABA. Isto inclui a determinação dos seus componentes, ou seja, o ciclo de vida, os métodos, as ferramentas e o hardware adequados.

Para tanto foi especificado um sistema baseado em conhecimento, estando já disponível a primeira versão da ferramenta que o implementou: XAMÃ, o Planejador de Ambientes.

Os componentes sugeridos que formam um ambiente são dos seguintes tipos:

- Ciclo de vida

- Representação do sistema
  . métodos e ferramentas para construção do modelo do sistema.
  . documentação de apoio.

- Gerenciamento
    - métodos para planejamento do projeto
    - métodos para acompanhamento do projeto

- Gerência de configuração
    - ferramentas para estimação e medição da confiabilidade

- Controle da qualidade de software
    - ferramenta para apoio ao estabelecimento da meta de qualidade
    - ferramentas para avaliação da qualidade ao longo do desenvolvimento

Os componentes são sugeridos considerando-se cinco tipos de parâmetros:

- Parâmetros das características do sistema: estão relacionados diretamente às características do software a ser desenvolvido, como o tamanho do sistema, seu grau de inovação, o grau de confiabilidade desejado e a função da aplicação.

- Parâmetros do computador: estão relacionados à configuração de máquina existente para desenvolver o

software (os recursos de máquina disponíveis irão determinar o software que poderá ser utilizado).

- Parâmetros do pessoal: estão relacionados às questões que envolvem as pessoas que trabalharão com o software, como por exemplo, a experiência profissional da equipe de desenvolvimento com a aplicação.

- Parâmetros de desempenho e custo: estão relacionados ao que se espera do software em termos de desempenho e custo.

- Parâmetros relacionados a outros recursos que serão utilizados: por exemplo, a necessidade de se utilizar um determinado método irá envolver a escolha de ferramentas que apóiem este método.

- Parâmetros tecnológicos do ambiente: estão relacionados à tecnologia adotada pelo ambiente (preferências do usuário por uma tecnologia, como por exemplo, a de orientação a objetos, irão influenciar na escolha dos componentes do ambiente).

- Parâmetros relacionados a outros componentes de software: a determinação de um componente pode implicar que outros mais sejam escolhidos de forma a apoiar o uso do

primeiro. É o caso de um compilador, que determina o uso de um ou mais depuradores.

- Parâmetros relacionados diretamente ao componente: referem-se a questões relacionadas ao próprio componente, como por exemplo, o fato de um método de custo só ser recomendado se tiver sido calibrado adquirindo o perfil do ambiente onde será utilizado.

A partir do planejamento do ADS , o instanciador gera o Ambiente de Desenvolvimento de Software. A função "Instanciar ADS" tem como objetivo gerar, de forma operacional, o Ambiente de Desenvolvimento de Software planejado com o apoio de XAMÃ bem como gerar um Assistente Especialista de apoio ao uso do ambiente.

O Ambiente Gerador de Ferramentas tem como objetivo auxiliar na construção de novas ferramentas que serão incorporadas à Estação TABA.

IV.4.2 - Classificação da Estação TABA

De acordo com a taxonomia proposta, a Estação TABA pode ser classificada da seguinte forma:

- **Classificação baseada na Arquitetura**

Em termos de sua arquitetura, pela possibilidade que oferece de ser configurada gerando ambientes de desenvolvimento de software adequados a diferentes domínios de aplicação, a Estação TABA possui um meta-ambiente configurador situando-se, portanto, na classe dos **Ambientes Centrados em Configuração.**

- **Classificação baseada no Suporte Oferecido ao Usuário**

A Estação TABA considera que no desenvolvimento de um projeto são realizadas três atividades fundamentais: **construção do produto, avaliação da qualidade do produto e gerência do processo de desenvolvimento.** Assim sendo, estas três atividades são apoiadas através de ferramentas específicas.

No que se refere ao suporte para construção do produto, tem-se:

- **Suporte a todo o ciclo de vida:** esta característica está presente na especificação da Estação. O planejador de ambientes XAMÃ (AGUIAR 1992), parte da definição do ciclo de vida mais adequado ao projeto (em geral, uma composição de mais de um modelo) e sugere métodos e ferramentas

adequadas a cada uma das atividades presentes no ciclo de vida escolhido.

- **Suporte para atividades clericais:** o suporte a este tipo de atividades também está previsto e pode ser caracterizado como um subconjunto do item anterior.

- **Suporte para reutilização de componentes anteriores:** a reutilização de componentes se dará de várias formas. Estão previstas ferramentas que permitam reutilização de especificações e projeto. embora ainda não tenham sido iniciados sub-projetos que visem sua construção. A reutilização de módulos será feita conforme proposto para o sistema CAOS (WERNER 1992). Neste momento tem-se um protótipo deste sistema. Outro contexto onde se terá reutilização de componentes será no Ambiente Gerador de Ferramentas. dado que ferramentas serão construídas a partir de um repositório de componentes (TRAVASSOS 1992).

Inteligência localizada: várias das ferramentas previstas para a Estação são ferramentas inteligentes. XAMÃ. o planejador de ambientes é um sistema baseado em conhecimento.

No que se refere à avaliação da qualidade tem-se especificado e em fase de desenvolvimento. um ambiente completo e configurável para avaliação da qualidade (ROCHA

1992). Este ambiente fornece ferramentas para especificação da meta de qualidade de um produto, para predição de sua qualidade ao longo do desenvolvimento (**avaliação de especificações, avaliação de projeto e avaliação de programas**) e para estimação e predição da qualidade nas fases de teste e operação. Pela característica de ser configurável este ambiente suporta a avaliação de produtos construídos a partir de diferentes métodos e linguagens.

No que se refere à gerência do processo de desenvolvimento está especificado (SARDINHA 1992) um conjunto integrado de ferramentas que darão apoio ao **planejamento e acompanhamento de projetos**, bem como à **gerência de configuração.**

- Classificação baseada na Escala

A Estação TABA pode ser classificada como pertencente ao **Modelo Comunidade de Estados.** Além de ter as características dos modelos anteriores no que se refere ao apoio a um grupo considerável de desenvolvedores (Modelo Cidade) e a configuração do ambiente para um determinado projeto (Modelo Estado), a Estação TABA tem prevista através do integrador de ferramentas, capacidade para integrar ferramentas desenvolvidas fora da Estação.

- Classificação baseada em Características Gerais de Ambientes

O projeto da Estação TABA considera os aspectos de **facilidade de uso, facilidade de aprendizado e suporte a usuários inexperientes,** através das seguintes características:

- o planejador de ADS, XAMÃ, considera as características e conhecimentos dos usuários do ambiente ao planejá-lo;

- está prevista a existência de um assistente especialista de apoio ao uso dos ambientes gerados na Estação. Um primeiro protótipo nesta direção está descrito em (FALCÃO 1992)

Está previsto um elevado **grau de automação** com a possibilidade de desenvolvimento rápido de ferramentas automatizadas através do Ambiente Gerador de Ferramentas.

Questões relacionadas ao **valor** não foram consideradas, por ser este um projeto de pesquisa, sem perspectivas imediatas de comercialização.

CLASSIFICAÇÃO	AMBIENTES	INSCAPE	IDeA	ARCADIA	ADAGE	CENTAUR	PEGASE	GRASPIN	OASIS	PATHCAL	ASPIS	TABA
ARQUIT.	CENT. EM SIS. OPER.											
	CENT. EM LING. PROG.									X		
	CENT. EM MÉTODOS	X	X				X		X		X	
	CENT. EM CONFIGURAÇÃO			X	X	X		X				X
	CENT. EM TRAB. COOP.											
SUP. OFERECIDO AO USUÁRIO	P/ CONSTRUÇÃO											
	SUP. A TODO CICLO DE VIDA	X		X	X	X	X	X			X	X
	SUP. P/ ATIVIDADES CLERICAIS	X	X		X	X	X	X	X	X		X
	SUP. P/ REUT. DE COMP. ANTERIORES	X	X	X	X				X	X	X	X
	SUP. INTELIGENTE P/ ASS. NA EXP. DE NOVAS POSS. E ATIV. CRIAT.			X				X				
	INTELIGÊNCIA LOCALIZADA	X										X
	SUP. P/ RE-ENGENHARIA				X				X			
	P/ AVALIAÇÃO											
	SUP. P/ AV. DA QUALIDADE		X	X								X
	SUP. P/ TESTES	X		X					X	X		
	SUP. P/ EST. E MED. DA CONFIABILIDE											
	P/ GERÊNCIA											
	SUP. P/ PLANEJAMENTO	X										X
	SUP. P/ ACOMPANHAMENTO	X		X								X
	SUP. P/ GERÊNCIA DA CONFIG.				X		X		X			X
ESCALA	MODELO INDIVÍDUO					X				X		
	MODELO FAMÍLIA											
	MODELO CIDADE	X					X		X			
	MODELO ESTADO											
	MODELO COM. DE ESTADOS			X								X
CARAC. GERAIS	FACILIDADE DE USO	X	X	X	X	X		X	X	X	X	X
	FACIL. DE APRENDIZADO									X	X	X
	SUP. A USUÁRIOS INEXPER.				X							X
	GRAU DE AUTOMAÇÃO	X	X		X					X		X
	VALOR DO AMBIENTE	X										
	INTELIGÊNCIA GLOBAL		X			X						

Figura 16: Quadro Resumo da Taxonomia Proposta

CAPÍTULO V

CONCLUSÃO

Este trabalho apresenta os resultados de um estudo detalhado sobre Ambientes de Desenvolvimento de Software.

A análise dos principais conceitos, enfoques e classificações presentes na literatura técnica atual, tornaram possível a identificação das características mais marcantes da área.

Foi observado, ao longo desse estudo, que a questão da escala (PERRY 1988), ou seja, a crescente complexidade dos sistemas e do número de pessoas envolvidas, vem preocupando a comunidade científica, sugerindo a necessidade de ambientes mais voltados para o conjunto, envolvendo o trabalho cooperativo.

Com a experiência adquirida, foi proposta uma Taxonomia para a Classificação de Ambientes de Desenvolvimento de Software considerando-se como fundamentais os aspectos de arquitetura, do suporte oferecido ao usuário, de escala e das características gerais de ambientes.

A seguir, classificamos os ambientes descritos, no contexto da taxonomia proposta, verificando sua adequação ao objetivo desse trabalho.

A Estação TABA foi classificada, detalhadamente, segundo essa taxonomia.

Sugerimos como tema para trabalhos futuros, a continuação desse estudo, analisando outros Ambientes de Desenvolvimento de Software.

REFERÊNCIAS BIBLIOGRÁFICAS

(AGUIAR 1989) Aguiar. T.C. de "Protótipo do especificador de ambientes da Estação TABA"; Relatório Técnico - **ES-208/89**: Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. julho 1989.

(AGUIAR 1992) Aguiar. T.C. de "Um Sistema Especialista de Suporte à Decisão para Planejamento de Ambientes de Desenvolvimento de Software"; Tese de Doutorado; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. março 1992.

(BERNAS 1988) Bernas. P. "Manuel d'utilisation d'Asspégaz"; L.R.I.. Univ. de Paris XI; Orsay. 91400 ORSAY. France. octobre 1988.

(BERNAS 1989) Bernas. P. "L'Environnement PEGASE: Aspects Pratiques & Fondements Theoriques";Toulouse 89 - Second International Workshop on Software Engineering & its applications; Toulouse, France, december 1989.

(BIDOIT 1985) Bidoit, M.; Choppy, C.; Voisin, F. "The ASSPEGIQUE specification environmemt, motivations and design"; 3rd Workshop on Theory and Applications of abstract data types; Bremen, novembre 1985.

(BIDOIT 1987) Bidoit, M.; Capy, F.; Choppy, C. et alli "ASSPRO: un environment de programmation interactif et integré"; TSI vol. 6. No 1, 1987.

(BLASCHEK 1987) Blaschek. J.R.S. "Dicionário de Dados para Análise"; Anais do I Simpósio Brasileiro de Engenharia de Software; Petrópolis-RJ, outubro 1987.

(BORRAS 1988) Borras, P.; Clément, D. "CENTAUR: the System"; ACM SIGSOFT'88: Third Symposium on Software Development Environments (SDE3); Boston, Massachusetts, USA; november 1988.

(CHARETTE 1986) Charette, R. N. **Software Engineering Environments: Concepts and Technology**; McGraw-Hill, 1986

(COPPE 1987) Rocha, A.R.C. da et alli "Uma Estação de Trabalho para Desenvolvimento de Software"; Proposta de Projeto, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ; 1987.

(CRISPIM 1988) Crispim, M.E.H. "Definição de termos em Ambientes para o Desenvolvimento de Software": Relatório Técnico - **ES 175/88**. Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1988.

(CRISPIM 1991) Crispim, M.E.H. "Avaliação de Métodos para o Desenvolvimento de Software": Tese de Mestrado; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, maio 1991.

(CSTB 1990) Scaling Up: a Research Agenda for Software Engineering; Communications of the ACM, vol.33, nº 3, março 1990.

(DART 1987) Dart, S.A. et alli "Software Deevelopment Environments"; Computer, november 1987.

(DECLERFAYT 1989) Declerfayt, O.; Milgrom, E. "Requirements for the Next Generation of Case Tools", in CASE'89 - Third International Workshop on Computer Aided Software Engineering - Advanced Working Papers; London, England, july 1989.

(DE MARCO 1984) De Marco, T. "A Meta-Methodology for Systems Development", International Workshop on Models and Languages for Software Specification and Design"; Orlando, Florida, march 1984.

(DETIENNE 1989) Détienne, F. "L'Approche de L'Ergonomie Cognitive en Programmation Informatique" Toulouse 89 - Second International Workshop on Software Engineering & its applications; Toulouse, France, december 1989.

(DOWSON 1986) Dowson, M. "IStar - An Integrated Project Support Environment"; in Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments; 1986.

(DOWSON 1986) Dowson, M. "Integrated Project Support with IStar"; IEEE Software, november 1987.

(FOORGARD 1984) Foorgard, R.; Guttag, J.V. "REVE: a term rewriting system generator with failure-resistant Knuth-Bendix"; Proc. of an NSF Workshop on the rewrite rule laboratory, J.V. Guttag, D. Kapur and D.R. Musser Eds., Septembre 1984.

(FREEMAN 1980) Freeman, P. "The Nature of design", Tutorial on Software Design Techniques; Peter Freeman e Anthony Wasserman (eds), IEEE Computer Society, 1980

(GIAVITTO 1989) Giavitto. J.L.: Devarenne. A.; Rosuel. G.; Holvoet. Y. "ADAGE: Utilisation de la Genericité pour Construire des Environnements Adaptables"; Toulouse 89 - Second International Workshop on Software Engineering & its applications; Toulouse, France, december 1989.

(GRUDIN 1991) Grudin, J. "CSCW Introdution"; Communication of the ACM, vol. 34, nº 12, 1991.

(HOFFNAGLE 1985) Hofinagle, G.F.; Beregi, W.E. " Automating the Software Development Process", IBM Systems Journal, vol. 24; nº 2, 1985.

(ILLINGWORTH 1983) **Dictionary of Computing:** Valerie Illingworth, edt.; Oxford University Press, 1983.

(KRAMER 1989) Kramer, J.: Magee, J.; Sloman, M. "Configuration Support for System Description, Construction and Evolution": Proceedings of the Fifth International Workshop on Software Specification and Design; Pittsburgh, Pennsylvania, USA; may 1989.

(KYNG 1991) Kyng, M. "Design for Cooperation: Cooperating in Design; Communication of the ACM; vol 34, nº 12, 1991.

(LEWIS 1990) Lewis, T.G.: Oman, P.W. "The Challenge of Software Development", IEEE Software, nov 1990.

(LUBARS 1989) Lubars, M.D. "The IDeA Design Environment" 11th. International Conference on Software Engineering, Pittsburgh, Pennsylvania, USA; may 1989.

(LUCENA 1987) Lucena, C.J.P. de **Inteligência Artificial e Engenharia de Software:** PUC/RJ - IBM/Brasil; Jorge Zahar Editor; 1987.

(LUCENA 1989) Lucena, C.J.P.; Takahashi, E.T. "A Anatomia de um Ambiente de Desenvolvimento de Software"; Projeto Estra - V Reunião de Trabalho - Coletânea de Resultados de Pesquisas; Vol. I; abril 1989.

(MANNUCCI 1989) Mannucci, S.; Mojana, B.; Navazio, M.C.; Romano, V.; Terzi, M.C.; Torrigiani, P. "GRASPIN: A Structured Development Environment for Analysis and Design"; IEEE Software, pp. 35-43; november 1989.

(MATTOSO 1990) Mattoso, A. "TABA-OBJ: Um ambiente de Desenvolvimento de Software com Orientação a Objetos"; Tese de Mestrado; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, agosto 1990.

(MATTOSO 1992) Mattoso, M.L.Q. "Taxonomias de Ambientes de Desenvolvimento de Software: A Classificação do Ambiente TABA"; Relatório Técnico TABA RJ-8/92; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1992.

(MOURA 1989) Moura, L.M.V. "Ambientes de Desenvolvimento de Software" Relatório Técnico - ES-200/89; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, julho 1989.

(OSSER 1989) Osser, H.; Harrison, W. "An Environment for Building Environments"; in CASE'89 - Third International Workshop on Computer Aided Software Engineering - Advanced Working Papers; London, England, july 1989.

(PENEDO 1988) Penedo, M.H.; Riddle, W.E. "Guest Editors Introduction: Software Engineering Environment Architectures"; IEEE Transactions on Software Engineering, vol. 14, nº 6, june 1988.

(PERRY 1988) Perry, D.E.; Kaiser, G.E. "Models of Software Development Environments"; Proceedings of 10th International Conference on Software Engineering; 11-15 april 1988.

(PERRY 1989) Perry, D.E. "The Inscape Environment"; 11th. International Conference on Software Engineering, Pittsburgh, Pennsylvania, USA; may 1989.

(PERRY 1991) Perry, D.E.; Kaiser, G.E. "Models of Software Development Environments"; IEEE Transactions on Software Engineering, vol. 17, nº 3, march 1991.

(POSTEL 1989) Postel, C. de; Massié, H.; Mathis, A. "OASIS: Un Atelier de Gestion des Objets pour des Applications Evolutives"; Toulouse 89 - Second International Workshop on Software Engineering & its applications; Toulouse, France, december 1989.

(PUNCELLO 1988) Puncello, P.; Torrigiani, P.; Pietri, F.; Burlon, R.; Cardile, B.; Conti, M. "ASPIS: A Knowledge-Based CASE Environment"; IEEE Software, march, 1988.

(ROCHA 1987) Rocha, A.R.C. da; Aguiar, T.C.; Blaschek, J.R.S. "Ambientes para o Desenvolvimento de Software: Definição de Termos"; Relatório Técnico - **ES 137/87**, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1987.

(ROCHA 1988) Rocha, A.R.C.da; Souza, J.M. "TABA: Uma Estação de Trabalho para o Engenheiro de Software". <u>XIV Conferencia Latino Americana de Informatica</u>, Buenos Aires, Argentina, setembro 1988.

(ROCHA 1989a) Rocha, A.R.C.da; Aguiar, T.C.de; Souza, J.M.; D'Ipolitto, C. " O Meta Ambiente da Estação Taba"; Relatório Técnico - Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1989.

(ROCHA 1989b) Rocha, A.R.C. da et alli "O meta-ambiente da Estação TABA"; <u>XV Conferencia Latino Americana de Informatica</u>; Santiago, Chile; julho 1989.

(ROCHA 1990a) Rocha, A.R.C. da; Souza, J.M.; Aguiar, T.C. "TABA: a heuristic workstation for software development"; <u>Compeuro 90</u>; Tel Aviv, Israel; maio 1990.

(ROCHA 1990b) Rocha, A.R.C. da et alli "Uma visão funcional da Estação TABA"; Relatório Técnico do Projeto TABA, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, dezembro 1990.

(ROCHA 1991) Rocha, A.R.C. da; Souza, J.M (coordenadores) "O PROJETO TABA"; Relatório Técnico TABA **RJ-7/91**; Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1991.

(ROCHA 1992) Rocha, A.R.C. da; Passos, M.C.F.; Ramos, I.; Belchior, A.; Campos, G.; Fonseca, A. "An Environment for Software Quality Evaluation"; submetido a "International Conference on Software Quality"; Jerusalém, Israel; november 1992.

(SAITO 1989) Saito, N. "The Software Engineering Environment" in **Japanese Perspectives in Software Engineering**; Yoshiro Matsumoto and Yutaka Ohno (eds.), 1989.

(SARDINHA 1992) Sardinha, E. "Gerência de Projetos na Estação TABA"; Exame de Qualificação ao Mestrado, USP-São Carlos, maio 1992.

(SHNEIDERMAN 1980) Shneiderman, B.; **Software Psycology**; Wintthrop Publishers; Cambridge, Ma.; 1980.

(SHAW 1990) Shaw. M. "Properties for an Engineering Discipline of Software"; IEEE Software. nov 1990.

(SORENSON 1988) Sorenson. P.G.; Tremblay. J.P.; McAllister. A. "The Metaview System for Many Specification Environments", IEEE Software. march 1988.

(STAHL 1991) Stahl, M.M.; Rocha. A.R.C. da "UMBOÉ: O Ambiente de Desenvolvimento de Software Educacional"; Anais do 2º Simpósio Brasileiro de Informática na Educação; Porto Alegre-RS, novembro 1991.

(STINSON 1989) Stinson, W. "Views of Software Development Environments: Automation of Engineering and Engineering of Automation"; ACM SIGSOFT - Software Engineering Notes. vol. 14,, nº 6. october 1989.

(TAKAHASHI 1989) Takahashi. T.; Introdução à Programação Orientada por Objeto; Apostila do IX Congresso da SBC- VIII Jornada de Atualização em Informática; Uberlândia. MG. julho 1989.

(TAYLOR 1988) Taylor. R.N. et alli "Foundations for the Arcadia Environment Architecture"; ACM SIGSOFT'88: Third Symposium on Software Development Environments (SDE3); Boston, Massachusetts, USA; november 1988.

(THOMAS 1989) Thomas. I. "Tool Integration in the Pact Environment" 11th. International Conference on Software Engineering, Pittsburgh, Pennsylvania, USA; may 1989.

(TRAVASSOS 1990) Travassos, G.H. "Ferramentas Gráficas para Engenharia de Software"; Tese de Mestrado, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ. 1990.

(TRAVASSOS 1992) Travassos. G.H. "OCA: Um modelo para representação das informações manuseadas pelas ferramentas da Estação TABA", Exame de Qualificação para Doutorado; COPEE/UFRJ, 1992.

(TSICHRITZIS 1990) Tsichritzis. D. "Towards Intregated Software Communities" in Object Management: Gestion d'Objets; Centre Universitaire d'Informatique. Université de Genève. juillet 1990.

(WASSERMAN 1981) Wasserman, A.I. "The Ecology of Sofware Development Environments" in IEEE Computer Society Press Technology Series - Computer-Aided Software Engineering (CASE); ed. E.J. Chikofsky, 1988.

(WEIDERMAN 1986) Weiderman, N.H.; Habermann, A.N.; Borger, M.W.; Klein, M.H. " A Methodology for Evaluating Environments"; in Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments; 1986.

(WERNECK 1990) Werneck, V.M.B. "Taxonomia de domínios de Aplicação"; Tese de Mestrado, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1990

(WERNER 1992) Werner, C. " Reutilização de Software no Desenvolvimento de Software Científico" Tese de Doutorado, Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ, 1992

(WILANDER 1986) Wilander, J. "An Interactive Programming System for Pascal"; in **Interactive Programming Environments**: editors: Barstow, D.R.; Shrobe, H.E.; Sandewall, E.; McGraw Hill; 1986.

(YOUNG 1988) Young, M.; Taylor, R.N. "Software Environment Architectures and User Interface Facilities"; IEEE Transactions on Software Engineering , vol. 14, nº 6, june 1988.