Ambientes de Apoio a Desenvolvimento Transformacional

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela Sra. **Christina Brandão von Flach** e aprovada pela Comissão Julgadora.

Campinas, 23 de julho de 1992

Prof. Dr. Geovane Cayres Magalhães

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do título de MESTRE em Ciência da Computação.

A Guiga, que sempre esteve.
A Lourdes, que, de algum modo, soube estar.
A Beto, que chegou para ficar.
A Sandra, a Martha, a Neto.
Ao que vai chegar ...

Conteúdo

Agradecimentos vi

Resumo vii

1 Introdução 1
- 1.1 Evolução Histórica de Engenharia de Software . . . 1
  - 1.1.1 Linguagens . . . 1
  - 1.1.2 Paradigmas de Programação . . . 3
  - 1.1.3 A Manipulação de Programas . . . 4
  - 1.1.4 Metodologia de Desenvolvimento de Software . . . 6
  - 1.1.5 Métodos de Desenvolvimento de Software . . . 7
  - 1.1.6 Modelos e Meta-modelos . . . 9
  - 1.1.7 CASE . . . 11
  - 1.1.8 Formalismos . . . 14
- 1.2 Este Trabalho . . . 15
  - 1.2.1 Motivação . . . 15
  - 1.2.2 Resultados . . . 17
  - 1.2.3 Organização do texto . . . 18

2 Projeto e PAC 21
- 2.1 Introdução . . . 21
- 2.2 Estudos sobre Projeto . . . 22
  - 2.2.1 A Natureza do Problema de Projeto . . . 22
- 2.3 Desenvolvimento de Software como Instância de Projeto . . . 23
  - 2.3.1 Projeto Auxiliado por Computador . . . 23
  - 2.3.2 Crítica de Métodos Correntes para Desenvolvimento de Software . . . 24
  - 2.3.3 Alguns Aspectos Essenciais em Projeto de Software . . . 24
- 2.4 Aspectos Críticos a Considerar . . . 26
  - 2.4.1 Operações Básicas de Projeto . . . 26
  - 2.4.2 Mecanismos de Coordenação . . . 31
- 2.5 Considerações Finais . . . 36

3 Um Modelo Genérico do Processo de Software 39
- 3.1 Introdução . . . 39
- 3.2 O Modelo PW em Detalhe . . . 39

3.2.1 A Perna Esquerda 39
3.2.2 A Perna Direita 42
3.3 O Passo Canônico 44
3.4 Operações de Projeto e o Modelo PW 45
3.5 Considerações Finais 46

4 Transformações **47**
4.1 Sistemas Transformacionais 47
4.1.1 Terminologia Básica 47
4.1.2 Modelos 48
4.1.3 Papéis no Processo de Desenvolvimento de Software 50
4.1.4 Objetivos 51
4.1.5 Aspectos Relevantes 51
4.1.6 Tipos de Regras de Transformação 53
4.1.7 Exemplos de Regras de Transformação 54
4.1.8 Uma Taxonomia para Transformações 54
4.1.9 Transformações e Reusabilidade 55
4.1.10 Áreas Correlatas 56
4.2 O Sistema de Transformação do tipo Fold/Unfold 57
4.2.1 Tipos de Regras de Transformação 57
4.2.2 Um Exemplo 58
4.3 O Projeto CIP 60
4.3.1 Tipos de Regras de Transformação 60
4.4 DRACO 61
4.4.1 Tipos de Regras de Transformação 61
4.4.2 Funcionalidade 63
4.5 PSI 64
4.5.1 Tipos de Regras de Transformação 66
4.6 Considerações Finais 66

5 Ambientes de Apoio ao Desenvolvimento Transformacional **69**
5.1 Introdução 69
5.2 Por que "Transformacional"? 69
5.3 Requisitos para Ambientes de Apoio ao Desenvolvimento Transformacional . . . 70
5.4 Uma Arquitetura de Referência 72
5.5 Uma Instância Imediata 72
5.5.1 UNIX 72
5.5.2 DPSK 75
5.5.3 Repositório de Dados 78
5.5.4 Processadores de Linguagens 78

6 Um Experimento Concreto **79**
6.1 Introdução 79
6.2 Objetivos 79
6.3 Cenário de Utilização do Ambiente 80

6.3.1 Configuração do Ambiente 80
6.3.2 Operação 82
6.4 Arquitetura do Ambiente 82
6.4.1 Mecanismo de Controle 84
6.4.2 Objetos do Sistema 85
6.4.3 Mecanismo de Gerência de Dados 86
6.4.4 Servidor(es) de Regras 86
6.4.5 Mecanismo de Transformação 87
6.4.6 Mecanismo de Documentação 87
6.5 Detalhamento da Implementação 87
6.5.1 Módulo de Inicialização 88
6.5.2 Utilização do DPSK 88
6.6 Encadeamento Regressivo 89
6.6.1 Algoritmo 91
6.7 Um Exemplo 93
6.7.1 Domínios 93
6.7.2 Objetos do Sistema 93
6.7.3 Regras de Transformação 94
6.7.4 Resultados 95

7 Considerações Finais **97**
7.1 Contribuições deste Trabalho 98
7.2 Trabalhos Futuros 98

A QuickSort **99**
A.1 Introdução 99
A.2 Quicksort em Linguagem Natural 99
A.3 Quicksort em KRC 100
A.4 Quicksort em Notação Funcional 100
A.5 Quicksort em Notação Lógica 101
A.6 Quicksort em Notação Procedural 102

B De uma Visão Funcional a uma Visão de Objetos **103**
B.1 O Domínio DFD 103
B.1.1 Descrição em Linguagem Natural 104
B.1.2 Formalização de DFDs 104
B.2 O Domínio de Objetos 112
B.2.1 Modelo de Objetos 112
B.3 Métodos de Transformação DFD-MO 115
B.3.1 Método de Booch-Rothemberg 115
B.3.2 Método de Alabiso 115

C Sintaxe para Condições **117**

Bibliografia **119**

Lista de Figuras

1.1 Cadeia Típica de Desenvolvimento 5
1.2 O Modelo Clássico do Ciclo de Vida 7
1.3 Ponto de Partida do Modelo PW 10
1.4 Primeiro Nível de Decomposição do Processo de Desenvolvimento 10
1.5 Uma Instância do Modelo PW: Cadeia de Desenvolvimento 12
1.6 Um Ambiente CASE 13
1.7 Uma Instância do Modelo PW: Geradores de Aplicações em BDs 14
1.8 Objeto deste Trabalho 18

2.1 Classificação/Instanciação 27
2.2 Generalização/Especialização 28
2.3 Agregação/Decomposição 29
2.4 Evolução/Involução 30
2.5 Ortogonalidade entre operações de abstração 30
2.6 Exemplo da Estrutura Clássica de Sistemas Especialistas 32
2.7 Exemplo de um Modelo de Blackboard Simples 33
2.8 O Modelo de *Blackboard* 34
2.9 Resolução de Quebra-cabeças 34

3.1 Segundo Nível de Decomposição do Processo de Desenvolvimento 40
3.2 A Perna Esquerda 41
3.3 O Modelo LST do Processo de Reificação 44
3.4 O Passo Canônico 45

4.1 O Modelo do Paradigma Transformacional 49
4.2 O Modelo de Balzer 50
4.3 Hierarquia de Domínios 62
4.4 Transformações e Refinamentos 63
4.5 Esquema de Funcionamento de DRACO 63

5.1 Arquitetura Geral de Referência 73
5.2 Uma Instância Imediata 74

6.1 Estrutura do Domínio DFD 81
6.2 A classe DFD 82
6.3 Arquitetura do Protótipo 83

6.4 Domínios, Objetos e Relações 91

B.1 Transformação de DFD para Objetos 103
B.2 Uma Linguagem para Representação do Modelo de Objetos 113
B.3 Convenções de um FDC 114
B.4 Convenções de um OSC 114

Lista de Tabelas

1.1 Uma Classificação para Métodos 9

2.1 Métodos para atacar Incertezas 36

4.1 Arcabouço para Tecnologias de Reusabilidade 55

Agradecimentos

Este trabalho foi co-orientado pelo prof. Eduardo Tadao Takahashi, atualmente coordenador da Rede Nacional de Pesquisa.

A ele, expresso os meus sinceros agradecimentos pelas horas de dedicação – recurso tão escasso para ele hoje em dia e, por isso mesmo, tão valioso para mim – pelo grande interesse, pelo *make yourself comfortable* – permitindo o meu livre acesso a equipamentos, livros, material e pessoal – pelas críticas construtivas e, principalmente por ser uma pessoa tão admirável.

o o o

A Geovane, pelo grande apoio e incentivo desde os bons e idos tempos de CPD-UFBA.

Às professoras Anna Friedericka Silva, Célia Gomes e Lolita Dantas, pela amizade e incentivo e aos professores do DCC-UFBA pelo voto de confiança.

A Gledson e ao pessoal do ETHOS → IPS → RNP, pela amizade, pelas horas de descontração e pela boa vontade.

Aos irmãos e companheiros de república da rua Luverci Pereira de Souza, 273, pela *insustentável leveza do lar*. Obrigada Sandra, Sandra loira, Bruno, Valery e Carol, Inés, Cecil, Silvinha, Marquinhos, Ivonne e Pimpim ...

o o o

Aos amigos de lá, de cá, de todos os cantos.

Resumo

Este trabalho examina transformações como aspecto central em ambientes de projeto de software e propõe: (a) um conjunto básico de operações de projeto que constitui um núcleo de transformações, (b) um esquema de coordenação composto por dois mecanismos básicos – um genérico, baseado em *blackboards*, e um específico, baseado em um mecanismo de encadeamento regressivo sobre relações entre objetos, e (c) uma arquitetura genérica de referência para a implementação de ambientes de apoio a desenvolvimento transformacional.

As idéias propostas foram exercitadas em um protótipo concreto, programado em C.

Abstract

This work is meant to approach transformations as the main aspect of software design environments. We propose: (a) a basic set of design operations that constitutes a kernel of transformations, (b) a scheme of coordination composed of two kind of mechanisms – a generic one, based on the blackboard mechanism, and a specific one, based on a backward chaining mechanism over relationships among objects, and (c) a generic reference architecture from which we can discuss software development environments that support transformations.

Our proposal is demonstrated through a concrete prototype, written in C.

Capítulo 1

Introdução

Uma das maiores dificuldades em qualquer atividade de Engenharia é a **transformação** de uma especificação de requisitos possivelmente ambígua, incompleta, inconsistente e mal-estruturada em um projeto bem estruturado, completo, consistente e correto.

Isso é particularmente difícil em **Engenharia de Software**, uma área onde há muito pouco conhecimento e experiência acumulados para guiar o projetista na execução de um **conjunto de atividades** cuja natureza ainda não está bem definida.

O próprio termo "Engenharia de Software", como é de conhecimento geral, foi cunhado em caráter provocativo, no final da década de 60, visando estimular o enfoque da tarefa de construção de software segundo uma ótica de "engenharia", em oposição à "artesania" até então vigente.

Muitos anos se passaram desde então e muitos aspectos básicos de desenvolvimento de software permanecem obscuros, quase intocados, e têm sido abordados de forma sistemática somente em anos recentes: modelos e meta-modelos de projeto, modelos descritivos *versus* prescritivos, representação de *design*, transformações, etc.

Este trabalho se concentra no exame detalhado de apenas um aspecto básico: o de **transformações**. Este capítulo principia com uma revisão sobre alguns conceitos fundamentais em Engenharia de Software, como fio condutor para a introdução de transformações e de uma abordagem específica a transformações a ser desenvolvida nos capítulos subsequentes.

1.1 Evolução Histórica de Engenharia de Software

Apresentaremos aqui uma recapitulação crítica de conceitos em Engenharia de Software que irá se concentrar na rediscussão de oito termos de importância central, cuja evolução, ao nosso ver, reflete a evolução da área como um todo.

1.1.1 Linguagens

Linguagens de programação ocupam o papel mais importante em desenvolvimento de software desde os primórdios da programação. A evolução de linguagens tem tentado, a um só tempo, **ocultar do programador a crescente complexidade da arquitetura de hardware** onde o programa irá ser executado, e também **incorporar construções de nível cada vez mais alto**, que reflitam diretamente abstrações do domínio de aplicação de que o programa irá fazer parte.

O marco inicial nesta linha de evolução foi o surgimento das primeiras **linguagens de programação de alto nível**, a partir do final da década de 50. A relativa abstração obtida em relação à máquina, principalmente através das estruturas de dados e de controle, proporcionou o primeiro ganho efetivo em produtividade nas tarefas de criar e depurar programas. Outro grande resultado foi a definição de mecanismos de suporte a **abstrações procedurais**, que evoluiu para mecanismos de **abstração de dados** e, mais recentemente, para **abstrações de objetos**.

Em contrapartida, o processo de evolução de linguagens impõe um custo adicional: programas escritos em linguagens de alto nível precisam ser compilados – ou traduzidos – para programas equivalentes em linguagem de máquina. A **compilação** de um programa nada mais é do que um processo de **transformação** entre alguns níveis de representação, onde, quaisquer que sejam estes níveis, o programa descrito deverá apresentar o mesmo comportamento.

Uma linguagem de programação possui três aspectos básicos: a sua **sintaxe** – a estrutura das suas construções (objetos e operações) –, a **semântica** destas contruções e a **pragmática**.

O significado ou semântica de um programa equivale ao que a máquina executa sob o seu comando. Há, dessa forma, uma equivalência entre o comportamento da máquina e a semântica do programa. Em geral, a semântica de uma linguagem de programação pode ser caracterizada por uma descrição formal das classes de objetos e operações que podem ser manipuladas pela linguagem.

As questões de caráter pragmático referem-se aos aspectos relativos à implementação da linguagem de programação em hardware e sistema operacional específicos.

O processo de compilação é parte de um contexto de desenvolvimento de programas amplo, como mostra a figura 1.1.

O próximo grande salto qualitativo no conceito de linguagens é representado pelo desenvolvimento de **linguagens de especificação**, uma tentativa de superar o *gap* existente entre a verbalização de um problema e a sua solução algorítmica expressa em alguma linguagem de programação, através de construções mais abstratas, que ocultam a estrutura de controle sequencial e a representação de dados imposta pela arquitetura de von Neumann. Toda linguagem de programação de alto nível é, em certo sentido, uma linguagem de especificação.

Finalmente, a evolução de linguagens tenta atender a algumas necessidades do usuário "leigo" – aquele que não lida com programação como atividade "fim" – com o surgimento das **linguagens de quarta geração**[1], geralmente não-procedurais e ditas "orientadas para problemas". Em essência, tais linguagens permitem expressar uma aplicação através de primitivas de muito alto nível, em sintaxe e terminologia similares às utilizadas no domínio da aplicação.

o o o

Essa breve digressão sobre linguagens e sua evolução permite registrar os seguintes pontos fundamentais que serão úteis para este trabalho:

- *o desenvolvimento de software implica na utilização de um conjunto de linguagens, com distintos níveis de abstração;*
- *as linguagens guardam uma relação de ordem entre elas, desde a mais próxima do domínio da aplicação até a mais próxima do hardware em que os programas irão executar;*

[1] Fourth- generation Languages

- *a estruturação de um ambiente de apoio a desenvolvimento de software deve contemplar, entre outros requisitos, o suporte à codificação de construções em cada uma dessas linguagens e, em especial, a* **transformação automática sucessiva** *de construções de uma linguagem "mais abstrata" para a linguagem subsequente "mais concreta", até chegar à linguagem de máquina.*

1.1.2 Paradigmas de Programação

A disseminação da programação em diversas áreas de aplicação levou ao surgimento de "estilos específicos" de programar, que correspondem a diferentes **paradigmas de programação**. A relação entre paradigmas de programação e linguagens tem um duplo aspecto: por vezes, um paradigma motivou o projeto e implementação de uma linguagem; em alguns casos, uma linguagem inspirou o estabelecimento de um paradigma.

O paradigma **Procedural**, o pioneiro e ainda o mais utilizado por diversas comunidades, inclui linguagens imperativas como C, Pascal e FORTRAN e enfatiza a construção de procedimentos para a resolução de problemas. As linguagens que se adequam este estilo têm como característica principal a estreita correspondência entre as suas construções e o modelo de máquina de execução subjacente, em geral baseado nos conceitos de von Neumann.

O paradigma de **Programação em Lógica** enfatiza a estrutura lógica de um problema, mediante a descrição através de sentenças em uma linguagem simbólica, de fatos e propriedades relativos ao problema em questão. Este paradigma levou ao surgimento da linguagem PROLOG, cuja base matemática se apóia nos princípios do Cálculo de Predicados. O enfoque do paradigma de Programação em Lógica caracteriza um estilo mais abrangente de programação, conhecido como não-procedural ou **declarativo**, que também inclui o paradigma Funcional.

O paradigma **Funcional** foi estabelecido através do estilo de programação preconizado e difundido através do uso da linguagem LISP. Este paradigma utiliza eminentemente o conceito de função na resolução de problemas e dá tratamento uniforme a dados e programas. Nas linguagens funcionais ditas "puras" não existem os conceitos de atribuição, passagem de parâmetros por referência e comandos de controle tradicionais. Friedman afirma que as linguagens procedurais foram desenvolvidas por causa da arquitetura de von Neumann, enquanto que as linguagens funcionais foram desenvolvidas *apesar* dela [Frie92].

Na última década, o paradigma de **Programação Orientada a Objetos** tem se afirmado. Este paradigma preconiza a visão do mundo como um conjunto de objetos que podem ser agrupados em classes, através da identificação de propriedades "comuns". Cada classe possui estrutura interna privativa e comportamento próprio, implementado através de procedimentos conhecidos como métodos. A comunicação no mundo de objetos é feita através do envio/recebimento de mensagens - que ativam métodos. Classes podem ser agrupadas em classes mais genéricas (superclasses) através de um mecanismo de generalização, ou especializadas em subclasses. Tais mecanismos permitem a representação da estrutura de classes de forma hierárquica, suportando o conceito de herança. Smalltalk é a linguagem que melhor ilustra os conceitos deste paradigma.

o o o

Para este trabalho, o conceito de paradigma contribui com os seguintes pontos:

- *a sintaxe externa de uma linguagem de programação reflete um paradigma específico de programação e, portanto, não constitui uma convenção arbitrária.*
- *um conjunto de linguagens, idealmente, deve suportar de forma coerente um único paradigma (ou conjunto de paradigmas compatíveis) através de todo o processo de construção de um programa.*

1.1.3 A Manipulação de Programas

Programas estão sujeitos a diversas formas de manipulação. Após a criação de uma versão inicial de um programa, ele comumente passa por traduções, testes, depuração e eventuais correções, em um ciclo de atividades conhecido como **cadeia típica de desenvolvimento.**

A cadeia típica de desenvolvimento engloba atividades automatizadas com as quais qualquer programador está familiarizado, como mostra a figura 1.1. Algumas dessas atividades começaram a ser realizadas com suporte automatizado a partir do advento dos primeiros montadores e sistemas operacionais.

Além desses tipos mais comuns de manipulação, muitos programas duradouros passam por modificações para se adequarem a especificações alteradas ou estendidas. Segundo Dershowitz [Ders83], outros tipos de manipulação que um programa pode sofrer são:

- Síntese

  A **síntese** de um programa a partir da especificação formal de um problema é um tipo de manipulação de programas onde a construção é feita através de transformações sucessivas, em passos que acrescentam mais detalhe às versões sucessivas do programa em desenvolvimento, enquanto preservam propriedades, entre elas, a correção entre as versões. Em particular, o programa final será equivalente à sua especificação inicial.

- Abstração e Instanciação

  A **abstração** de um conjunto de programas para se obter um **esquema de programa** e a **instanciação** de **esquemas abstratos** para resolver problemas concretos podem ser vistas como tipos especiais de manipulação de programas. Esta perspectiva provê uma metodologia para a aplicação de conhecimento previamente adquirido a novos problemas.

- Modificação por Analogia

  A **modificação por analogia** é um tipo de manipulação de programas em que um novo programa é obtido através da modificação de um programa previamente criado, após a identificação de uma analogia entre a especificação do programa existente e a especificação do programa que se deseja obter. A analogia encontrada é utilizada para transformar o programa existente naquele desejado.

- Extensão

  A **extensão** de um programa para satisfazer uma especificação estendida é um outro tipo de modificação de programas. Deve-se escrever código que estenda o programa incompleto, de forma a se alcançar os novos objetivos, cuidando, no entanto, para que a especificação original continue a ser satisfeita.

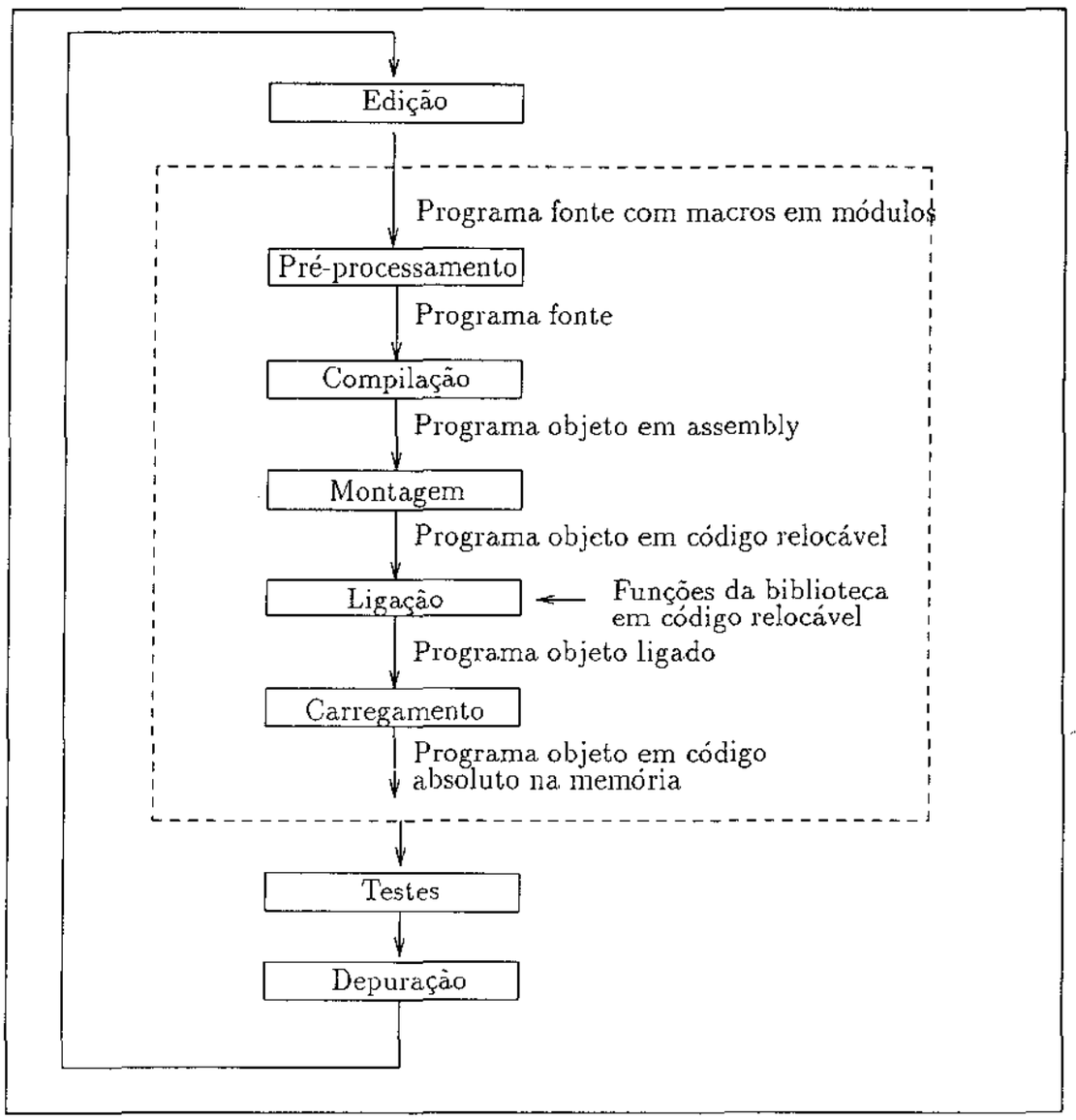


Figura 1.1: Cadeia Típica de Desenvolvimento

- Anotação e Análise

  A **anotação** um programa com assertivas é uma maneira de documentá-lo de forma a facilitar as atividades de depuração, verificação, modificação e análise da sua eficiência. As assertivas são colocadas em pontos específicos do programa e asseguram que alguma relação é verdadeira para os valores correntes das variáveis do programa sempre que o controle de execução passa por aquele ponto.

- Verificação e Validação

  A **verificação** de um programa corresponde à atividade de demonstrar formalmente que o mesmo satisfaz sua especificação, enquanto a **validação** corresponde à atividade de avaliar experimentalmente o programa em relação à sua **especificação informal**.

o o o

Para este trabalho, as seguintes observações são importantes:

- *há um vasto conjunto de operações de manipulação de programas, com terminologia e propriedades variadas, encontradas na literatura. Poucos trabalhos de taxonomização e investigação exaustivas foram feitos, mas o levantamento de Dershowitz [Ders83] acima mostra que uma abordagem mais genérica e sistemática é possível.*

- *a manipulação de artefatos de software nos níveis mais abstratos anteriores à obtenção de uma especificação ou programa completos é muito pouco estudada, mas possivelmente passível de taxonomização similar à feita para programas.*

1.1.4 Metodologia de Desenvolvimento de Software

Entende-se por **processo de desenvolvimento de software** o conjunto de atividades que têm como objetivo a criação de um programa[2] que resolva um problema do mundo real.

O processo de desenvolvimento de software é comumente considerado intrínseco a um **ciclo de vida** de software, conceito que apareceu já na década de 60, mas cuja discussão detalhada só aparece na década de 70 através do modelo de Boehm.

Um ciclo de vida de software é supostamente composto por **fases** sucessivas e, além das fases que compõem o processo de desenvolvimento, ele inclui as fases de **operação** e **manutenção** (ou **evolução**).

Segundo uma versão do modelo clássico do ciclo de vida apresentada na figura 1.2, o processo de desenvolvimento é composto pelas seguintes fases/atividades genéricas:

- **especificação**: descrição do que o programa deve fazer, a partir de uma análise dos requisitos do conceito da aplicação.

- **projeto**: descrição de como o programa deverá fazer o que foi especificado.

- **codificação**: implementação do que foi projetado em alguma linguagem de programação.

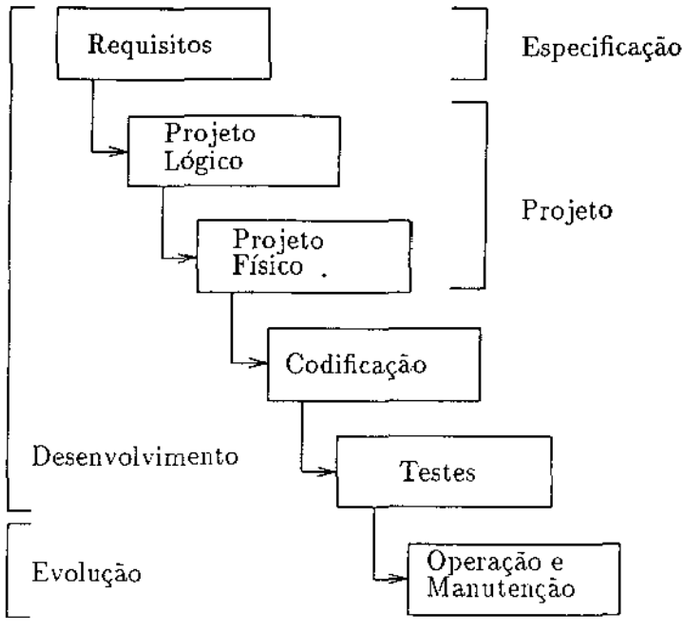


Figura 1.2: O Modelo Clássico do Ciclo de Vida

O modelo clássico enfatiza as obrigações de **verificação** ou de **validação** após cada atividade e também através de uma fase específica de **testes**.

o o o

Para este trabalho, vale observar que o conceito de ciclo de vida é considerado por muitos marginal em qualquer processo de construção de software, e está supostamente baseado em necessidades de ordem administrativa e/ou organizacional. Contudo:

- *o aspecto organizacional é intrínseco a qualquer atividade que envolva mais de uma pessoa ou mais de uma tarefa complexa para uma única pessoa. Isto implica em que o suporte automatizado a aspectos de desenvolvimento de software deve incluir apoio à estruturação de uma tarefa em fases.*

1.1.5 Métodos de Desenvolvimento de Software

Em geral, define-se o termo **método** como sendo uma forma de proceder para se atingir um certo resultado. No contexto de Engenharia de Software, um **método de desenvolvimento** é um conjunto de diretrizes para a seleção e utilização de **técnicas** e **ferramentas** para a construção de um artefato de software, onde as ferramentas dão suporte automatizado às técnicas.

Os métodos de desenvolvimento podem ser classificados em dois tipos básicos:

[2] O termo programa é usado para efeito de simplificação. Um termo mais adequado e genérico seria "artefato de software".

- **formais**

  1. se as técnicas e ferramentas disponíveis são formais, ou seja, têm significado matemático preciso;
  2. se permitem raciocínio matemático sobre os programas e o processo de desenvolvimento de software propriamente dito.

- **informais**, caso contrário.

Bjorner cita em [Bjor87] mais dois tipos de métodos de desenvolvimento, situando-os entre os dois tipos anteriores:

- **rigorosos**
  Um método é rigoroso se ele deixa de ser formal por omitir provas formais detalhadas.

- **sistemáticos**
  Um método é sistemático se ele deixa de ser rigoroso por omitir relações formais entre os estágios de desenvolvimento.

A década de 70 foi caracterizada pelo surgimento de inúmeros métodos para a análise e o projeto de sistemas baseados, em sua maioria, em abordagens *top-down* e estruturadas, com uso abundante de notações diagramáticas para a representação de projeto e comunicação visual de idéias, com destaque para os diagramas de fluxo de dados. Dentre os métodos mais difundidos, podemos citar [Gane84], [Jack75], [Your79] e [DeMa89].

Os métodos de desenvolvimento mais populares são informais. Embora proporcionem um arcabouço para desenvolvimento organizado de software, há incompatibilidade entre certas notações usadas para apoiar algumas fases de desenvolvimento. Além disso, eles permitem a construção de esp cificações ambíguas – e.g., o texto associado a figuras em diagramas é representado em linguagem natural e mesmo as figuras podem possuir interpretações múltiplas.

Os **métodos formais** são técnicas baseadas em modelos matemáticos para a descrição de propriedades de sistemas de computação. Eles oferecem um arcabouço no qual pode-se especificar, desenvolver e verificar sistemas de forma disciplinada [Wing90]. A base matemática de um método formal em geral se traduz diretamente em uma **linguagem de especificação formal**.

Segundo alguns pesquisadores, é necessária uma ênfase maior em trabalhos sobre formalização para que a sua importância seja devidamente compreendida [Baue80]. Somente uma abordagem formal poderá garantir rigor, integridade conceitual e coerência global ao processo de desenvolvimento de software e, por extensão, aos seus produtos.

Métodos formais têm as suas limitações: o seu uso efetivo requer suporte automatizado; as técnicas desenvolvidas não são facilmente aplicáveis a programas grandes e complexos – como é a maior parte dos programas encontrados na prática. Por outro lado, considerando que programas são objetos formais, suscetíveis a manipulação formal, os princípios subjacentes à especificação formal e verificação de programas oferecem um ótimo guia e podem ser bastante úteis para o desenvolvimento de programas corretos e de manutenção simples.

Na tabela 1.1 propomos uma classificação para métodos e apresentamos alguns exemplos. Métodos de "programação" e de "desenvolvimento" suportam uma metodologia de programação em ponto pequeno ou ponto grande, respectivamente.

	Programação	Desenvolvimento
Formais	CIP [Baue85]	
Rigorosos		RAISE
Sistemáticos		VDM [Bjor89]
Informais	DRACO [Neig80]	cadeia típica de desenvolvimento

Tabela 1.1: Uma Classificação para Métodos

1.1.6 Modelos e Meta-modelos

O surgimento do conceito de ciclo de vida e dos diversos modelos que tentam representá-lo foi fruto da necessidade de descrever e padronizar o **processo de software**[3], com o objetivo de permitir o estabelecimento de metodologias de apoio a tal processo.

O modelo clássico evoluiu muito pouco desde a sua proposta inicial, mas ainda é bastante utilizado. Ele e os seus derivados são objeto de diversas críticas, entre elas, as de que são representações informais e excessivamente prescritivas do processo de software.

Atualmente, argumenta-se que a definição de um modelo genérico que sirva como estrutura conceitual unificadora é fundamental [Haeb89]. Um modelo genérico adequado deveria, sob um ponto de vista "**meta**":

- esclarecer e aumentar a compreensão sobre o processo de software e seus passos;
- permitir o projeto de novos métodos de desenvolvimento através da instanciação de modelos específicos obtidos a partir dele.

O modelo proposto por Lehman [Lehm84a] – o modelo PW – é um dos primeiros passos na direção de um (meta-)modelo adequado[4].

Segundo o autor, os processos de software atualmente em uso apóiam a evolução de aplicações – tanto durante a fase inicial de desenvolvimento, como durante a sua utilização – de forma *ad hoc*. As atividades que compõem tais processos não possuem dois pré-requisitos básicos: base teórica comum e unificação através de um arcabouço conceitual único, não podendo ser, portanto, combinadas em um **processo coerente**.

[3] O termo processo de software é usado por Lehman [Lehm84a] para referir-se aos processos de desenvolvimento e evolução de programas.

[4] Seguiremos a terminologia adotada por Lehman, que o trata por "modelo", apesar de considerarmos o PW um meta-modelo do processo de software, ou seja, um arcabouço conceitual que permite raciocinar sobre modelos do processo de software.

Figura 1.3: Ponto de Partida do Modelo PW

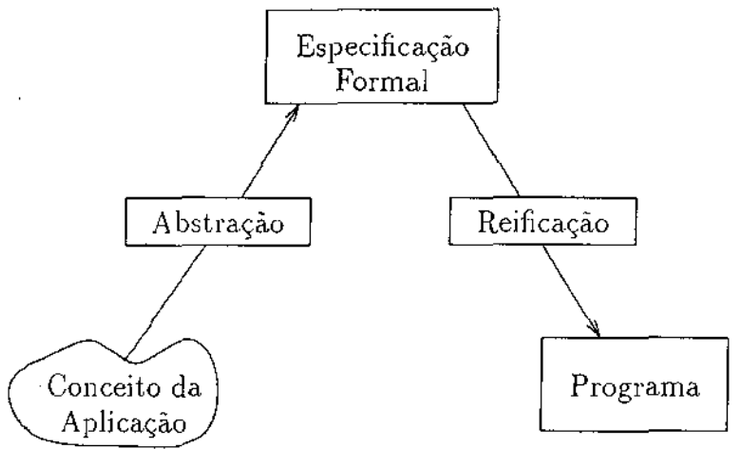


Figura 1.4: Primeiro Nível de Decomposição do Processo de Desenvolvimento

Em [Lehm84a] são descritos os primeiros passos na definição de processos de programação coerentes através do refinamento (decomposição) sucessivo de um modelo genérico de desenvolvimento e evolução de programas: o modelo PW. O modelo PW representa o processo de desenvolvimento de software como um **processo de transformação**, que parte de uma verbalização inicial do problema que se deseja resolver, geralmente informal e incompleta – **o conceito da aplicação** – para chegar a um **programa** que a satisfaça, como mostra a figura 1.3.

Como o processo de transformação do conceito da aplicação em um programa é excessivamente complexo para ser realizado em apenas um passo, ele é inicialmente decomposto em dois processos. Dessa forma, o modelo PW sofre o seu primeiro nível de decomposição. Um **domínio intermediário formal** é inserido, no qual se escreve uma **especificação** do conceito da aplicação. Neste sentido, uma especificação é uma representação abstrata de dois objetos reais: o conceito da aplicação e o programa final.

Os dois processos obtidos possuem características bem distintas: um processo de **abstração**, que leva o conceito da aplicação à especificação formal, e outro de **reificação**, que leva a especificação formal ao programa. A figura 1.4 [Haeb89] ilustra o primeiro nível de decomposição do processo de desenvolvimento.

Os dois processos obtidos através da decomposição do processo de desenvolvimento são mostrados no modelo PW como duas pernas de um "V" invertido. A perna esquerda representa o processo de abstração do conceito da aplicação no contexto de seu domínio de aplicação no

mundo real, com o intuito de gerar uma especificação formal do conceito da aplicação. As atividades envolvidas requerem operações de abstração comumente utilizadas para a modelagem de domínios. A perna direita modela o processo de reificação – ou concretização – o qual transforma a especificação obtida no processo anterior em um programa operacional.

Voltaremos a abordar este tema de forma mais detalhada no capítulo 3.

o o o

Destacamos dois pontos importantes em relação ao (meta-)modelo de Lehman, mesmo considerando o nível superficial em que foi abordado acima:

- *poucos métodos de desenvolvimento exploram sistematicamente a perna esquerda do modelo. É interessante tentar mapear métodos conhecidos como Gane&Sarson [Gane84], Jackson [Jack75] e outros, para os conceitos do modelo PW, não tanto para obter sucesso no mapeamento para métodos específicos, quanto para analisar deficiências fundamentais neles.*

- *um ambiente de desenvolvimento realmente genérico deve contemplar o suporte a um modelo genérico como o PW no nível mais geral, acrescido de facilidades de suporte a métodos concretos. Tal abordagem permitirá o destaque devido a aspectos essenciais em ambientes, em contraposição a aspectos secundários ou particulares a métodos específicos.*

1.1.7 CASE

O surgimento das primeiras ferramentas projetadas para o aumento da produtividade de quem desenvolve e para tornar o processo de software menos suscetível a erros, não proporcionou uma solução efetiva a nível global. Isto é justificado pelo fato de que essas ferramentas apóiam apenas a parte mais mecânica do processo, o ciclo iterativo de cadeias típicas de desenvolvimento constituído por edição-tradução-teste-depuração - também apelidado de *lower CASE*, por razões que veremos a seguir. Ferramentas típicas são: editores, compiladores, ligadores e depuradores.

A figura 1.5 mostra uma instância do modelo PW que representa a cadeia típica de desenvolvimento. As linhas pontilhadas representam atividades sem suporte automatizado.

O conceito de CASE (Computer-Aided Software Engineering) surgiu para cobrir essa deficiência. As ferramentas CASE são um conjunto de ferramentas integradas com o objetivo de apoiar todo o ciclo de vida do software e oferecer suporte automatizado à maior parte do processo de software. Assim, em geral, elas provêem uma estrutura para a determinação das necessidades do usuário, a elaboração de um projeto adequado para a aplicação e a decomposição/refinamento do projeto até a sua codificação. Dessa forma, as ferramentas CASE estenderam o apoio que vinha sendo dado às atividades do *back-end* às fases iniciais do processo de desenvolvimento ou atividades de *upper CASE* – análise de requisitos, especificação e projeto - com ênfase no suporte a métodos baseados nos conceitos de Análise e Projeto Estruturados.

Um ambiente CASE, na sua forma mais simples, é mostrado na figura 1.6. A camada mais interna consiste de um repositório de dados que contém todas as informações associadas aos diversos projetos. Estas informações ficam disponíveis às ferramentas da camada intermediária. A camada mais externa consiste dos componentes da interface, através da qual o usuário – analista, projetista ou programador - interage com o sistema.

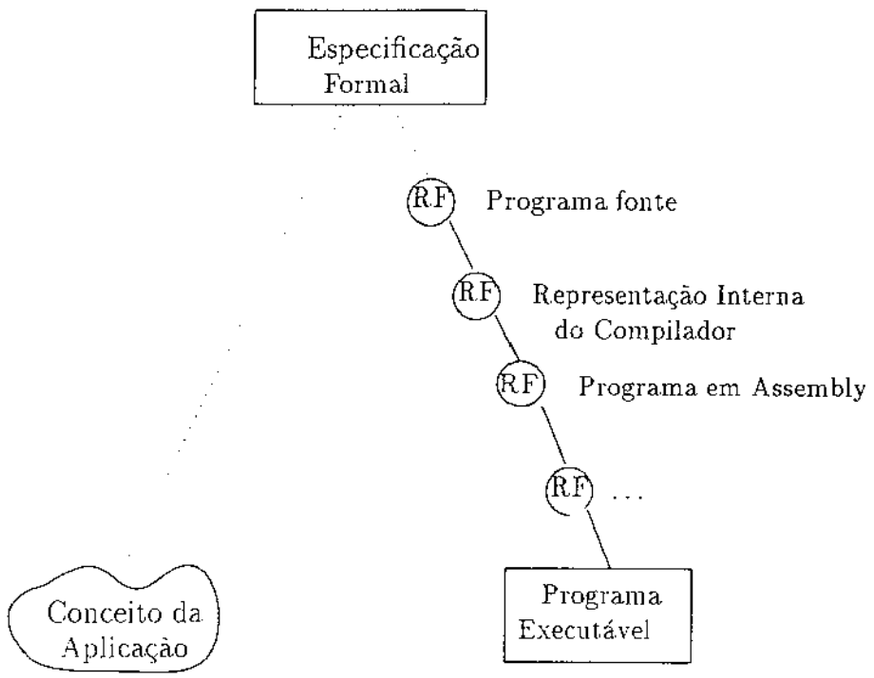


Figura 1.5: Uma Instância do Modelo PW: Cadeia de Desenvolvimento

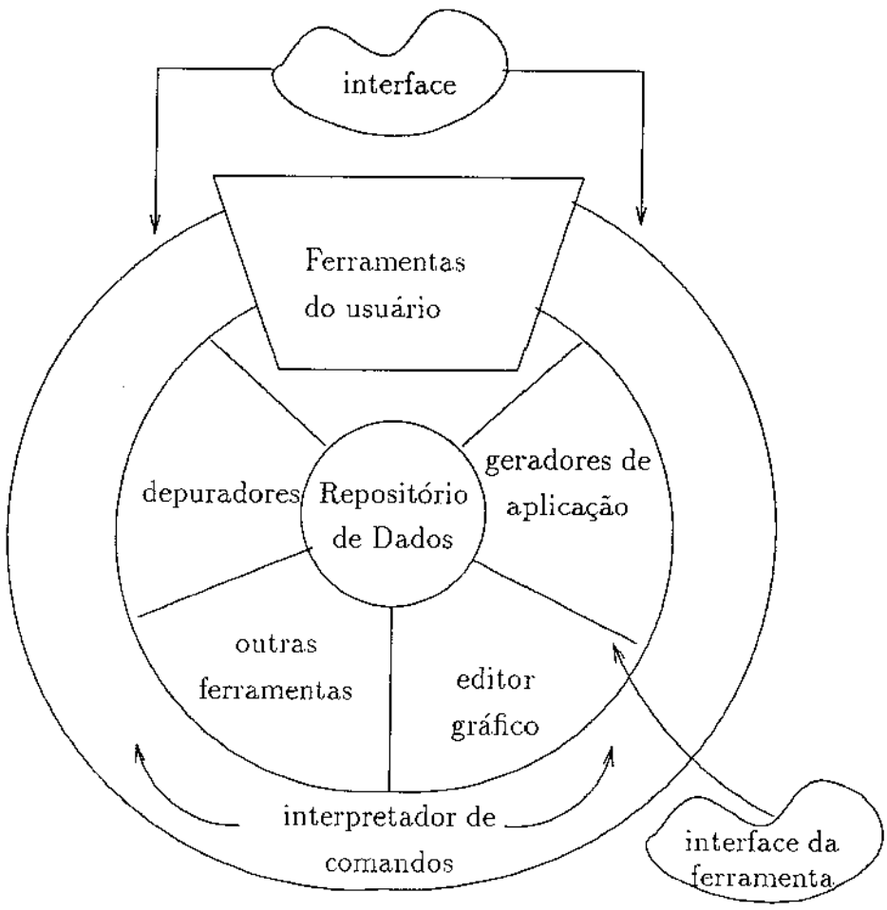


Figura 1.6: Um Ambiente CASE

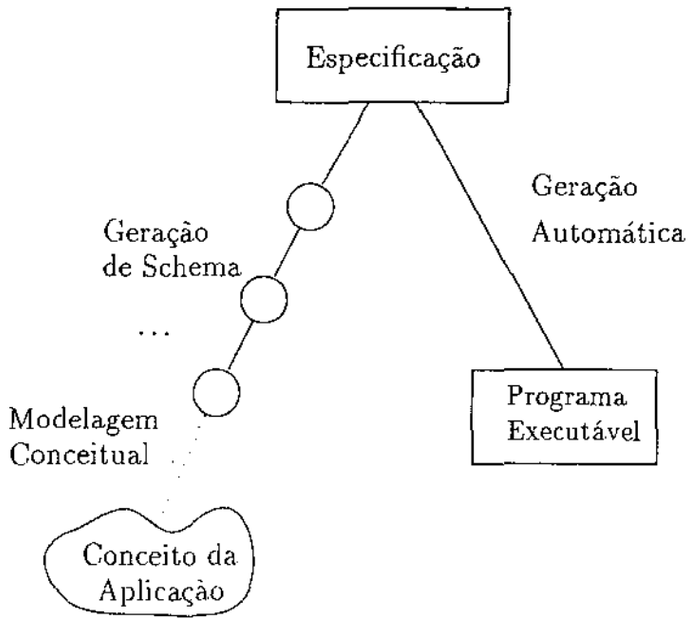


Figura 1.7: Uma Instância do Modelo PW: Geradores de Aplicações em BDs

Muitas ferramentas CASE possuem um gerador de aplicações integrado. **Geradores de aplicações** são ferramentas que surgiram para automatizar parte da processo de desenvolvimento, transformando especificações em programas de aplicação. As especificações podem ser definidas através de diálogos com o usuário ou de linguagens de quarta geração. Esta automação só foi possível porque os geradores lidam com domínios de aplicação restritos, com operações limitadas e E/S previamente definidas. Geradores de aplicações têm sido usados em diversas áreas que possuem as características mencionadas acima com relativo sucesso, com destaque para a área de Banco de Dados.

A figura 1.7 mostra outra instância do modelo PW, desta vez representando o desenvolvimento de aplicações na área de Banco de Dados, utilizando geradores de aplicações.

A evolução seguinte diz respeito a **Ambientes de Desenvolvimento de Software** (ADS). Um ADS é uma coleção de ferramentas integradas para dar apoio à tarefa de desenvolver e manter software, geralmente segundo algum método de desenvolvimento. Os ambientes de desenvolvimento de software facilitam a interação entre pessoas de um grupo de projeto (não necessariamente próximos), dando suporte integrado a várias fases do desenvolvimento, através de uma interface padronizada e amigável e de um repositório de dados comum.

1.1.8 Formalismos

Conforme se observou na seção 1.1.2, a noção de paradigma traz à tona um conceito mais básico que o de linguagem de programação. A sintaxe da linguagem é a externalização de um paradigma, e um paradigma é a "filosofia" subjacente à linguagem.

A noção de paradigma se estende naturalmente a desenvolvimento de sistemas. Por exemplo, a noção de **decomposição funcional** é a base para o método de Gane [Gane84]; o desenvolvimento por objetos se baseia na priorização da relação **generalização-especialização** em detrimento da relação **agregação-decomposição** em que a maior parte dos métodos se baseia.

Contudo, a "essência" de linguagens, métodos, etc., só é devidamente capturada através do uso de **formalismos**.

Em computação, como em qualquer outra área, formalismos se destinam a:

1. permitir a descrição precisa de fenômenos, conceitos, etc., estudar suas propriedades conhecidas, descobrir e generalizar outras propriedades ainda não conhecidas, e antecipar explanações , fenômenos e conceitos derivados dos originais, através de processos dedutivos aplicados aos sistemas formais utilizados.

2. oferecer suporte ao projeto e/ou geração de ferramentas de apoio à manipulação de modelos computacionais de tais conceitos, fenômenos, etc.

Em geral, é pouco comum o caso em que o formalismo adotado se presta a ambos os papéis. Uma exceção à esta regra que se constitui em um exemplo positivo é o uso de **gramáticas formais e autômatos** no estudo de linguagens de programação. Gramáticas formais de tipos específicos foram e são utilizadas para a descrição da sintaxe de construções de linguagens de programação. Por outro lado, os modelos de autômatos equivalentes a essas gramáticas permitem modelar e, em última análise, gerar partes significativas de ferramentas para manipular os artefatos descritos pelas gramáticas.

No estudo de métodos de desenvolvimento, várias abordagens distintas têm procurado modelar aspectos ou métodos específicos de desenvolvimento, com ênfase para os formalismos lógicos, como, por exemplo, a **Lógica de Predicados**. Contudo, alguns autores acreditam que dificilmente os mesmos formalismos usados para descrever poderão se adequar a apoiar a geração de ferramentas eficientes, de maneira análoga a autômatos em relação a processadores de linguagens.

1.2 Este Trabalho

1.2.1 Motivação

A natureza do processo de software vem sendo bastante estudada nos últimos anos. A partir do início dos anos 80, idéias sobre *design* concebidas em outras áreas começaram a ser aplicadas ao problema de desenvolvimento de software, levando a uma visão do processo de software como uma instância de projeto, em um sentido abrangente.

Esta perspectiva levou à identificação de alguns aspectos fundamentais do processo de software, em especial, questões de:

- **representações de projeto**, i.e., convenções linguísticas para a codificação sistemática de artefatos resultantes da atividade de projeto – o artefato de software só se torna "tangível" através de suas representações e estas interferem na capacidade das pessoas resolverem problemas [Free80].

- **transformação entre representações**, i.e., operações que permitirão mapear artefatos representados em um **nível linguístico**[5] para outro, e,
- **coordenação de atividades**, quando diversas transformações podem ser realizadas em paralelo.

o o o

Estudos metodológicos em desenvolvimento de software não têm investigado adequadamente a área de projeto mais amplo da qual ele certamente faz parte.

A generalização do estudo de métodos e a adoção de modelos genéricos como o PW de Lehman aproxima a discussão de desenvolvimento de software de estudos de projeto. Em particular, um aspecto interessante é a possibilidade de se estudar características básicas de projeto, a partir das quais **arquiteturas de referência** para ambientes de desenvolvimento possam ser prototipadas.

o o o

Outra linha de estudos parte do reexame dos conceitos de **transformação** comumente encontrados, e da adoção de um ponto de vista geral o suficiente para enquadrar transformações no contexto mais amplo de projeto e desenvolvimento de software.

1. Na literatura técnica, o conceito de transformação é relacionado a operações que permitem mapear incrementalmente especificações em programas equivalentes, sendo a raiz do termo **programação transformacional**.
2. O conceito de transformação pode ser generalizado para denotar operações que permitem mapear construções em distintos e sucessivos níveis linguísticos.
3. O conceito de transformação pode também se aplicar a operações dentro de um mesmo nível linguístico, como, por exemplo, o caso em que um programa em uma linguagem específica é continuamente refinado.
4. Finalmente, o conceito de transformação à luz do modelo PW de Lehman pode aplicar-se não somente ao processo de reificação, como também ao processo de abstração.

Para fins deste trabalho, o termo **transformação** corresponde a todos os conceitos acima enumerados e constitui o aspecto central a ser examinado.

o o o

Sob esta perspectiva, dois aspectos em transformações devem ser examinados:

- **transformações e projeto**

  Como associar transformações e projeto?

  Todos os modelos concretos de projeto propostos na literatura sugerem um conjunto de **operações básicas** que são aplicadas recorrentemente no projeto de artefatos em qualquer domínio. Um caminho a seguir é o estudo mais aprofundado de projeto e a investigação de tais operações básicas e de sua relação com a noção de transformações.

[5] vide capítulo 3

- **ambientes de apoio a desenvolvimento transformacional**

  Que aspectos são críticos em ambientes de apoio a desenvolvimento transformacional?

  O problema mais crítico é o de controle do leque de transformações possíveis em um determinado instante, problema que se torna mais crítico se transformações puderem ser aplicadas concorrentemente com vários projetistas trabalhando em paralelo. Mais do que controle, o problema central passa a ser a **coordenação** de atividades no ambiente.

o o o

Considerando os pontos introduzidos nesta seção, com destaque para transformações, operações básicas de projeto e mecanismos de coordenação, enumeramos quatro perguntas básicas que resumem o escopo e a motivação para este trabalho:

1. Quais são as características mais gerais do processo de projeto de software, independente de métodos específicos?

2. Como o conceito básico de transformação tem sido utilizado em desenvolvimento de software, incluindo:

   - definições diversas
   - propriedades
   - exemplos concretos de métodos e ambientes baseados em transformações

3. Há uma taxonomia de transformações geral o suficiente para que qualquer método de projeto possa ser expresso através da composição de operações identificadas em tal taxonomia?

4. Quais são as características fundamentais de um ambiente de suporte a desenvolvimento transformacional?

1.2.2 Resultados

Este trabalho examina **transformações** como aspecto central em ambientes de projeto de software e propõe:

- um **arcabouço conceitual** que toma emprestadas idéias do modelo PW e de estudos gerais sobre projeto para descrever o desenvolvimento transformacional de software de forma geral. Em particular, propõe-se que transformações específicas de métodos particulares sejam concebidas como composição e/ou variantes de um conjunto de **operações básicas de projeto**, e

- uma **arquitetura de referência** para a implementação de ambientes de apoio a desenvolvimento transformacional, apresentando como característica principal um **mecanismo de coordenação** de dois níveis, a saber:

  - um **geral**, baseado em *blackboards* e.

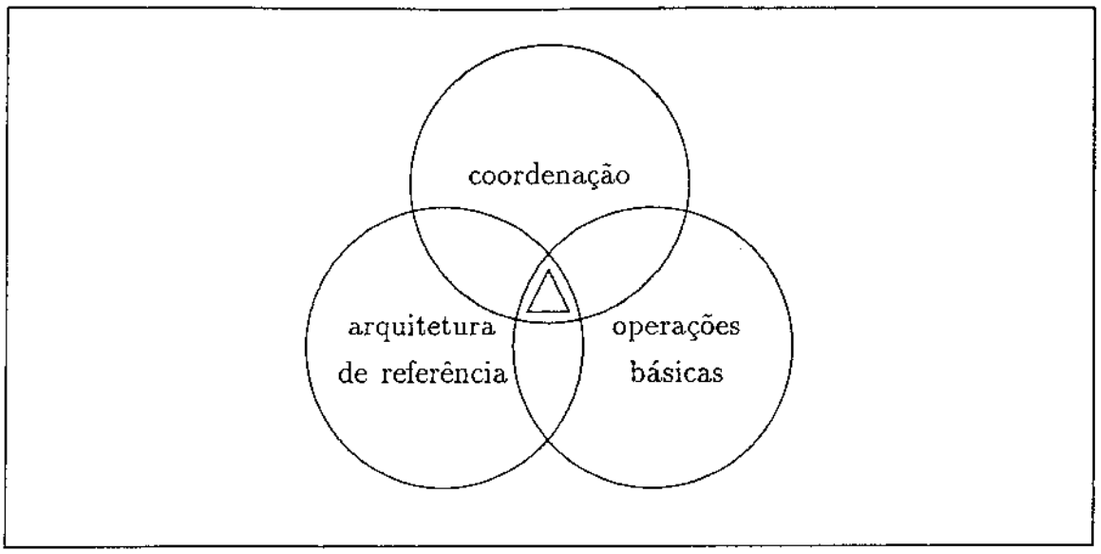


Figura 1.8: Objeto deste Trabalho

- um **específico**, baseado em um mecanismo de encadeamento regressivo[6] sobre relações entre objetos resultantes da aplicação das operações básicas (ou variantes) acima identificadas.

Essas idéias foram exercitadas em um protótipo concreto, programado em C em estações SUN.

A figura 1.8 ilustra o objetivo global deste trabalho.

1.2.3 Organização do texto

O capítulo 2 resume estudos gerais sobre projeto e analisa a tarefa de **desenvolver software** como uma instância específica de projeto, discutindo dois aspectos críticos a considerar em ambientes de apoio a projeto, a saber: **operações básicas** e **mecanismos de coordenação**.

O capítulo 3 descreve em maior detalhe o modelo PW de Lehman, com a preocupação básica de introduzir um **arcabouço conceitual** dentro do qual considerar **transformações**.

O capítulo 4 faz um apanhado geral sobre como o conceito de **transformações** tem sido proposto e tratado na literatura, incluindo a breve apresentação de experimentos significativos como os de Darlington, o CIP, DRACO e PSI.

O capítulo 5 propõe uma **arquitetura de referência** para ADSs baseada no modelo de *blackboards* e descreve uma instância concreta centrada no uso de DPSK[7] [Card87]. Nessa instância, um agente especial, sob a designação de AADT (sigla para Ambiente de Apoio ao Desenvolvimento Transformacional), corresponde ao experimento concreto deste trabalho e será descrito no próximo capítulo.

[6] backward chaining

[7] DPSK é uma plataforma para resolução distribuída de problemas.

Finalmente, o capítulo 6 descreve um experimento concreto de implementação das idéias dos capítulos anteriores, detalhando a estrutura implementada para os componentes lógicos de um sistema transformacional. Um aspecto particularmente destacado é a implementação de um mecanismo de encadeamento regressivo que se vale das operações (i.e., relações) básicas descritas no capítulo 2 para navegar entre objetos e domínios. O mecanismo de coordenação é genérico e, estando implementado no núcleo do ambiente, pode ser particularizado para qualquer método transformacional específico.

Capítulo 2

Projeto e PAC

2.1 Introdução

Métodos de desenvolvimento de software foram tradicionalmente tratados como disciplinas auto-contidas resultantes do "empacotamento" e generalização de técnicas concebidas em contextos bastante concretos de produção. Assim, não surpreende que cada um dos métodos mais conhecidos tenha sintaxe, jargão e prescrições bastante particulares para cada parte componente e que o todo, claramente, não é mais que a soma das partes[1].

Outra observação importante sobre métodos de desenvolvimento de software é o fato de que, até muito recentemente, não ocorria a sua discussão no contexto mais amplo de estudos sobre projeto[2], que existem de forma sistemática desde os anos 20.

Projeto, em uma acepção mais abrangente, dissociada do contexto específico de produção de software, é definido como um complexo processo de criação de um modelo para um artefato que se deseja produzir, como por exemplo, uma casa, um carro, um programa.

A partir do início dos anos 80, idéias sobre projeto concebidas em outras áreas começaram a ser aplicadas ao problema de desenvolvimento de software. Duas tendências importantes foram:

- a **visão organizacional**, que principiou com estudos como os de Simon [Simo73] sobre problemas mal-estruturados[3], sugerindo que tais problemas deveriam ser resolvidos através da decomposição em **sub-problemas tratáveis** que interagiriam segundo algum esquema de **coordenação** de partes para compor um todo que resolveria o problema original. Essa visão organizacional foi elaborada em trabalhos como o de Fox [Fox79] e Sathi [Sath85] e "empacotada" em forma simplificada em núcleos[4] baseados em modelos de *blackboards*.

- a **abordagem baseada em conhecimento** que, em essência, propunha o acúmulo e a utilização de estratégias *ad hoc* para a síntese de programas em domínios específicos,

[1] É exceção parcial ao comentário acima a evolução de técnicas de projeto de Bases de Dados Relacionais, em que ocorreu o caso pouco frequente de, dado um domínio restrito de aplicações - as então ditas "aplicações administrativas" - conceber-se um método de projeto adequado ao domínio e, por outro lado, fortemente baseado em um formalismo (Álgebra Relacional) que se mostrou útil não somente para apoio metodológico como também para a implementação correta de consultas em BDs. No caso mais geral, em que a diversidade de aplicações é ilimitada, vale a observação acima.

[2] design, no original

[3] ill-structured problems

[4] kernels

onde tal conhecimento pudesse ter sido capturado. Esta linha originou-se de experimentos com técnicas da área de IA em domínios específicos de aplicação, e técnicas simplificadas também chegaram ao estágio de "empacotamento" através de *shells* para os chamados sistemas especialistas.

Já em meados da década, principiou a se tornar familiar o termo **processo de software**, denotando a transcrição em um modelo prescritivo de um método específico de construção de software, como núcleo nervoso de um ambiente de apoio ao desenvolvimento. Em que pesem as críticas a essa linha, ela ganhou força nos anos seguintes e chegou à linha de frente com o *motto* de que *software processes are software too* [SPW84].

Finalmente, a associação entre metodologia de desenvolvimento de software e estudos gerais sobre projeto ganhou impulso definitivo com estudos como os de Bruns e Gerhart [Brun86], em que a literatura sobre projeto foi examinada e comentada, em busca de idéias e conceitos a serem experimentados no Projeto Leonardo, do MCC.

A preocupação deste capítulo é apresentar:

- um resumo das principais idéias encontradas em estudos gerais de projeto como uma disciplina independente de áreas de aplicação.
- uma revisão crítica de desenvolvimento de software, segundo a ótica de projeto.
- uma discussão sobre dois aspectos críticos em processos de projeto, a saber:
  - operações básicas de projeto e,
  - problemas de coordenação de atividades

2.2 Estudos sobre Projeto

2.2.1 A Natureza do Problema de Projeto

O primeiro grande aspecto a ressaltar se refere à natureza do problema de projeto: em geral, projeto é um problema de difícil trato, segundo argumentam, por exemplo, Simon [Simo73] e Rittel [Ritt73] de acordo com o resumo abaixo:

- **Projeto como um Problema Mal-estruturado**

  Para Simon, há uma vasta classe de problemas ditos "mal-estruturados" que, embora solúveis, são de difícil trato, porque o **espaço de soluções**[5] é demasiado grande (quando não infinito), e os processos para se cobrir tal espaço de forma exaustiva (e/ou chegar à solução ótima) não são computáveis em termos práticos. Por exemplo, Simon argumenta que xadrez e prova de teoremas não são bem-estruturados porque o espaço de soluções é excessivamente amplo.

  Embora não seja discutido, é evidente que o problema de projeto em geral, e de projeto de software em particular, é mal-estruturado no sentido utilizado por Simon.

[5] O espaço de soluções de um problema são todas as suas soluções parciais e finais.

- **Projeto como um Problema "Perverso"**

  Rittel [Ritt73] escreveu um trabalho bastante conhecido sobre problemas de **planejamento**, considerados por ele problemas perversos[6] pelas seguintes razões, entre outras:

  1. Não possuem formulação definitiva.
  2. O problema verdadeiro é chegar a uma formulação adequada do problema. Em outras palavras, a abordagem sistemática de problemas pressupõe uma abordagem de solução.
  3. Soluções não são avaliáveis por critério de VERDADEIRA/FALSA, mas por critério de BOA/MÁ.
  4. Toda solução é uma operação que afeta o domiínio do problema[7]. Em outras palavras, uma futura solução estará se referindo a um problema distinto, afetado pela solução anterior.
  5. Todo problema perverso pode ser considerado um sintoma de outro problema, provavelmente mais geral, cuja solução será afetada pela solução do primeiro problema.

  Considerados pelos critérios acima expostos, problemas de desenvolvimento de software são especialmente perversos na parte de formulação de requisitos, delimitação de funcionalidades e especificação de soluções.

A aceitação da visão de Simon e de Rittel como válida para o contexto de desenvolvimento de software explica e fortalece algumas tendências marcantes, incluindo:

- a ênfase corrente em aquisição de informação sobre o domínio de uma aplicação como parte essencial do ciclo de vida de desenvolvimento e anterior a qualquer definição de uma aplicação. Essa linha é adotada, por exemplo, no Projeto ITHACA, em aplicações de banco de dados e, mais recentemente, em sistemas especialistas.
- a dependência de qualquer método de desenvolvimento de software conhecido em relação a:
  - características dos domínios de aplicação cobertos pelos métodos e,
  - abordagens específicas de solução por decomposição funcional, orientada a objetos, etc.

2.3 Desenvolvimento de Software como Instância de Projeto

2.3.1 Projeto Auxiliado por Computador

Originária de esforços de automatização parcial do projeto de hardware, a área de PAC[8] – Projeto Auxiliado por Computador – durante mais de dez anos, não passou de uma coleção de

[6] wicked
[7] one-shot operation
[8] CAD (Computer-Aided Design)

técnicas de implementação de computação gráfica e de técnicas *ad hoc* para projeto de circuitos lógicos, e de seu mapeamento para arquiteturas concretas.

A recente ênfase de "CAD para Software" (CASE) acabou por revelar a precariedade conceitual dos esforços agrupados sob o rótulo de CAD.

Por outro lado, essa mesma ênfase em CASE também começa a mostrar que os métodos correntes de desenvolvimento de software necessitam de uma revisão abrangente, conceitual e prática, para que esforços com suporte automatizado ataquem aspectos essenciais e não secundários de projeto, quer de hardware, quer de software.

2.3.2 Crítica de Métodos Correntes para Desenvolvimento de Software

Métodos correntes de desenvolvimento de software se ressentem de diversas falhas, tanto de forma como de conteúdo, incluindo as seguintes:

- excesso de ênfase em aspectos sintáticos externos (terminologia, diagramação gráfica, etc.) em detrimento de aspectos semânticos.
- carência de embasamento conceitual e de explicitação de razões subjacentes e condicionantes impostas por determinadas ênfases metodológicas (por exemplo, decomposição funcional e orientação a objetos).
- excesso de natureza **prescritiva** em detrimento de natureza descritiva (limitação do número de níveis em hierarquias de DFDs, tamanho de módulos fonte, entre outros).
- imposição de restrições metodológicas divorciadas da realidade (e.g., a proposição de que o projeto de um sistema se faça *top-down* ou *bottom-up* de forma estrita, por mais interessante que seja do ponto de vista conceitual, não resiste à observação da realidade prática de ambientes de produção).

2.3.3 Alguns Aspectos Essenciais em Projeto de Software

Que aspectos são essenciais em qualquer método de desenvolvimento de software e que devem ser considerados na concepção de futuros ambientes de CAD para Software?

1. **Racionalidade Limitada**[9]

   A capacidade da mente humana para formular e resolver problemas é muito reduzida em comparação com a dimensão dos problemas do mundo real (cuja solução via computadores é desejada).

   Complexidade, dentro dessa ótica, se traduz como excesso de demanda sobre a racionalidade.

   A racionalidade limitada impõe, no caso de projeto de software:

   - a divisão de artefatos complexos em partes menores, com interfaces bem definidas, possibilitando consideração individual e independente de outras partes.

[9] Bounded Rationality

- a divisão de tarefas em subtarefas interrelacionadas, com explícita articulação de meios-fins e explícitas relações de precedências de execução no tempo.
- a necessidade de coordenar (individual ou coletivamente) a alocação de recursos (tempo, capacidade de processamento, etc.) para a resolução ordenada de um problema.

2. **Atividade Cooperativa**

   Exceto nos casos mais limitados, o desenvolvimento de um sistema de software é uma atividade de dimensões excessivas para ser executada em tempo hábil por um único indivíduo. A atividade é, pois, cooperativa, envolvendo um número variável de pessoas, tornando a tarefa de coordenação ainda mais complexa, por envolver a necessidade de negociar o comprometimento de cada ator em partes específicas do processo global.

3. **Representação Hierárquica e Incremental dos Dados**

   Como consequência das imposições da racionalidade limitada, há ao menos dois tipos de complexidade envolvidos no projeto de software, a saber:

   - **a complexidade de informações**

     Em particular, há o próprio artefato resultante do processo de projeto, cuja representação será necessária.

   - **a complexidade de tarefas**

     Em particular, há as tarefas de criação e manipulação direta de artefatos de projeto, cuja representação também será necessária.

   Em quaisquer dos casos, os requisitos impostos a uma convenção adequada de **representação de informações e tarefas** incluem:

   (a) **hierarquização** de representação segundo relações diversas como **parte-de**, **tem-especialização**, etc.

   (b) **evolução** no tempo, posto que um objeto sofrerá alterações ao longo de um processo de projeto, gerando **versões**, **revisões**, etc.

4. **Operações Básicas de Projeto**

   Há algumas operações básicas fundamentais de projeto que são recorrentemente utilizadas por projetistas de software em variantes específicas em diversos métodos, sob nomes variados.

   Por exemplo, uma operação mental óbvia é a **decomposição** de um objeto em suas partes componentes.

   A identificação de um **conjunto mínimo** de tais operações se reveste de particular interesse, para:

   - permitir a discussão de **modelos transformacionais** de desenvolvimento de software dentro de um arcabouço conceitual abrangente, e

- constituir o núcleo de um ambiente de desenvolvimento transformacional de software em que cada "operação fundamental" seja mapeada em uma ou mais transformações de um método específico.

2.4 Aspectos Críticos a Considerar

Para fins deste trabalho, i.e., o exame detalhado de desenvolvimento transformacional e ambientes de apoio, é importante discutir em detalhe dois aspectos enumerados na seção anterior, a saber:

- operações básicas de projeto, e
- mecanismos de coordenação

2.4.1 Operações Básicas de Projeto

Há um conjunto básico de operações tradicionalmente utilizadas para modelagem conceitual de domínios de aplicação. Essas operações correspondem a operações mentais usadas inconscientemente para identificar e categorizar coisas do mundo real, basicamente através de mecanismos de **abstração**.

Pesquisas em diversas áreas conduziram à escolha de três operações básicas (com suas respectivas operações inversas):

- classificação/instanciação
- generalização/especialização
- agregação/decomposição

Esse conjunto de operações é reconhecido e utilizado em diversas áreas, com destaque para IA [Sath85], BD [Peck88], [Smit77] e Programação Orientada a Objetos.

A aplicação dessas operações a um domínio de aplicação acaba por levar à sua representação através de uma hierarquia de abstrações.

Adotaremos tal conjunto básico de operações como sendo uma aproximação inicial para encontrarmos o conjunto mínimo procurado de operações de projeto. Assumiremos que estas operações são as mais gerais que um ambiente de apoio deverá suportar e passaremos a descrevê-las, juntamente com suas relações associadas, através de sintaxe simples, semântica em linguagem natural e em termos das suas propriedades algébricas.

As operações de básicas de projeto são:

- **Classificação/Instanciação**

  A operação de **classificação** corresponde à categorização de indivíduos em **classes**, com base em algumas propriedades comuns a todos. A operação inversa é a de **instanciação**, que corresponde ao processo de criar o indivíduo a partir do universal.

  A operação de instanciação tem como relação associada "tem-instância". A operação de classifição tem como relação associada "instância-de".

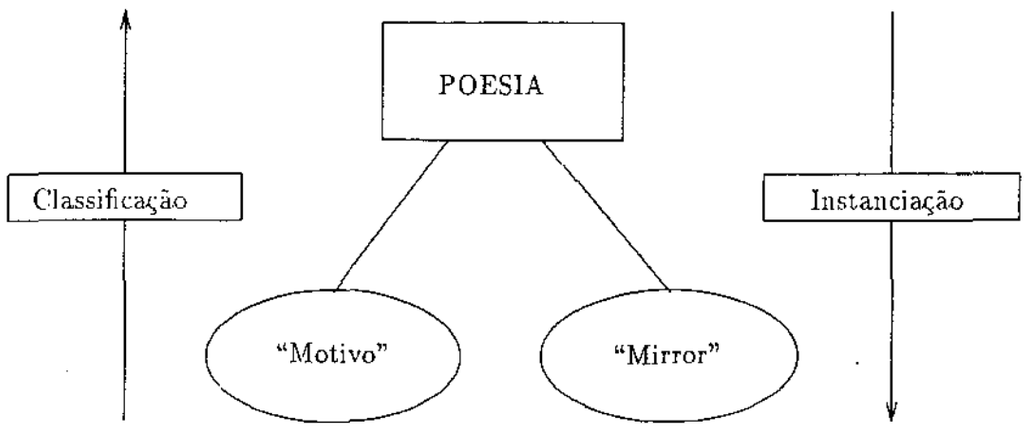


Figura 2.1: Classificação/Instanciação

Se considerarmos na figura 2.1[10] os poemas "Motivo" de Cecília Meireles e "Mirror" de Sylvia Plath, podemos classificá-los como instâncias da classe POESIA, sob a ótica de propriedades definidas em estilos literários, obtendo assim as relações: **instância-de("Motivo", POESIA)** e **instância-de("Mirror", POESIA)**.

Por outro lado, podemos aplicar a operação inversa e instanciar um poema a partir da classe POESIA, e.g., o "Soneto da Fidelidade" de Vinícius de Morais, obtendo: **tem-instância(POESIA, "Soneto da Fidelidade")**.

- **Generalização/Especialização**

  A **generalização** é uma operação que abstrai (ignora) propriedades de duas ou mais categorias, gerando outra mais genérica. A **especialização** é a operação inversa à generalização e consiste da criação de uma nova categoria através da distinção de pelo menos uma propriedade que a diferencie da mais genérica.

  As relações "tem-generalização" e "tem-especialização" estruturam os elementos que sofrem operações de generalização e especialização, respectivamente.

  As relações "tem-generalização" e "tem-especialização" têm as seguintes propriedades:

  - reflexiva
  - transitiva
  - anti-simétrica

  Para exemplificar a operação de generalização, se considerarmos as categorias CRÔNICA e CONTO, delas podemos abstrair uma categoria mais genérica, a PROSA. Da mesma forma, generalizando POESIA e PROSA, obtemos a categoria mais genérica LITERATURA.

[10] Na notação utilizada, elipses correspondem a instâncias e retângulos correspondem a classes.

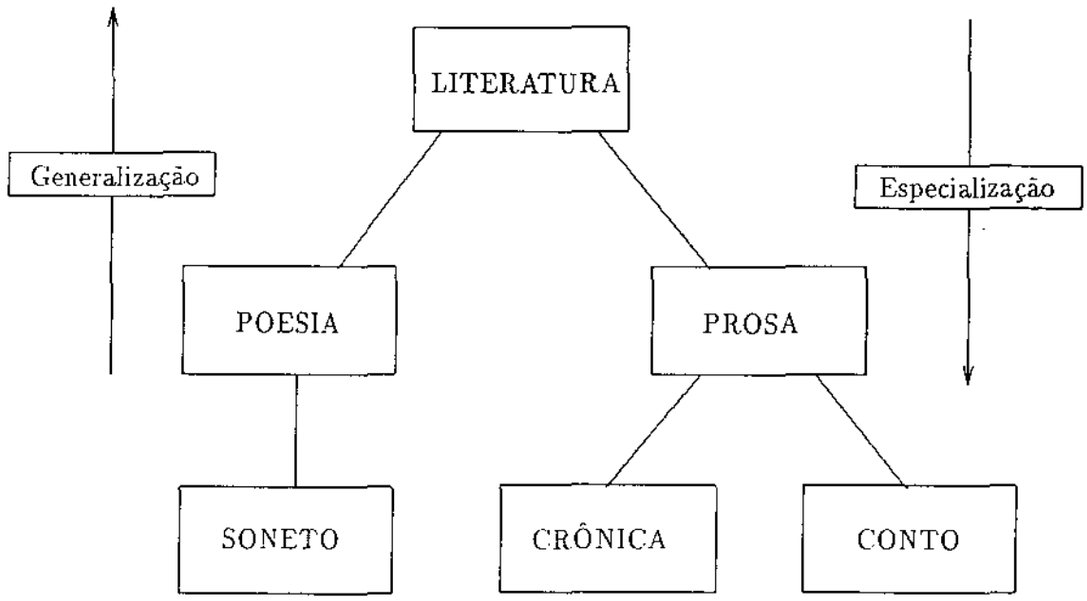


Figura 2.2: Generalização/Especialização

Por outro lado, podemos especializar a categoria POESIA, obtendo a categoria SONETO, que estabelece que todo soneto é um poema de quatro estrofes (duas de quatro versos e duas de três versos) com "chave-de-ouro"[11] na última estrofe.

- **Agregação/Decomposição**

  A **agregação** é uma operação que permite a obtenção de uma nova categoria a partir da composição de outras.

  A **decomposição** é a sua inversa e consiste do **detalhamento** de uma categoria ou de uma instância em suas partes constituintes.

  As relações "parte-de" e "tem-parte" estruturam os elementos que sofrem operações de agregação e decomposição, respectivamente.

  A relação "parte-de" tem as seguintes propriedades:

  - reflexiva
  - transitiva
  - anti-simétrica

  Por exemplo, instâncias da classe POESIA são compostas por, pelos menos, Título, AUTOR, Corpo do Poema e Ano de Publicação.

[11] termo usado para um "resumo" da temática do poema

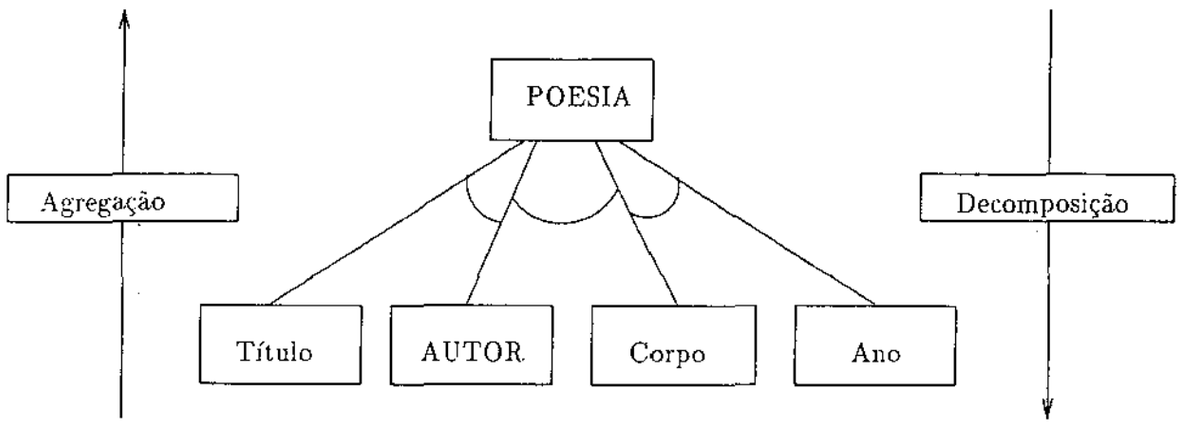


Figura 2.3: Agregação/Decomposição

Às operações básicas descritas, acrescentaremos o par evolução/involução, tratado como primitiva auxiliar.

- **Evolução/Involução**

  A **evolução** é uma operação que leva a uma versão nova de uma instância ao longo do tempo. A **involução**, sua operação inversa, retorna a uma versão anterior.

  As relações correspondentes às operações de evolução e involução são "tem-evolução" e "tem-involução", respectivamente.

  A relação "tem-evolução" tem as seguintes propriedades:

  - reflexiva
  - transitiva
  - anti-simétrica

  Por exemplo, "O Romanceiro da Inconfidência", um conjunto de poemas de Cecília Meireles, pode ser adaptado para o cinema, gerando um texto (roteiro), instância de PROSA.

Operações de Projeto e Paradigmas de Estruturação de Sistemas

Como se observa com frequência, as operações de **generalização/ especialização**, **agregação/ decomposição** e **classificação/ instanciação** são ortogonais entre si, permitindo representá-las operando segundo eixos independentes, como ilustra a figura 2.5.

Uma discussão interessante [Taka92] é a caracterização de métodos de programação/ desenvolvimento com respeito a essas operações.

Métodos tradicionais baseados em **decomposição funcional**, partem da visão de agregação/decompos como critério fundamental para a estruturação de sistemas. Já os métodos ditos **orientados a**

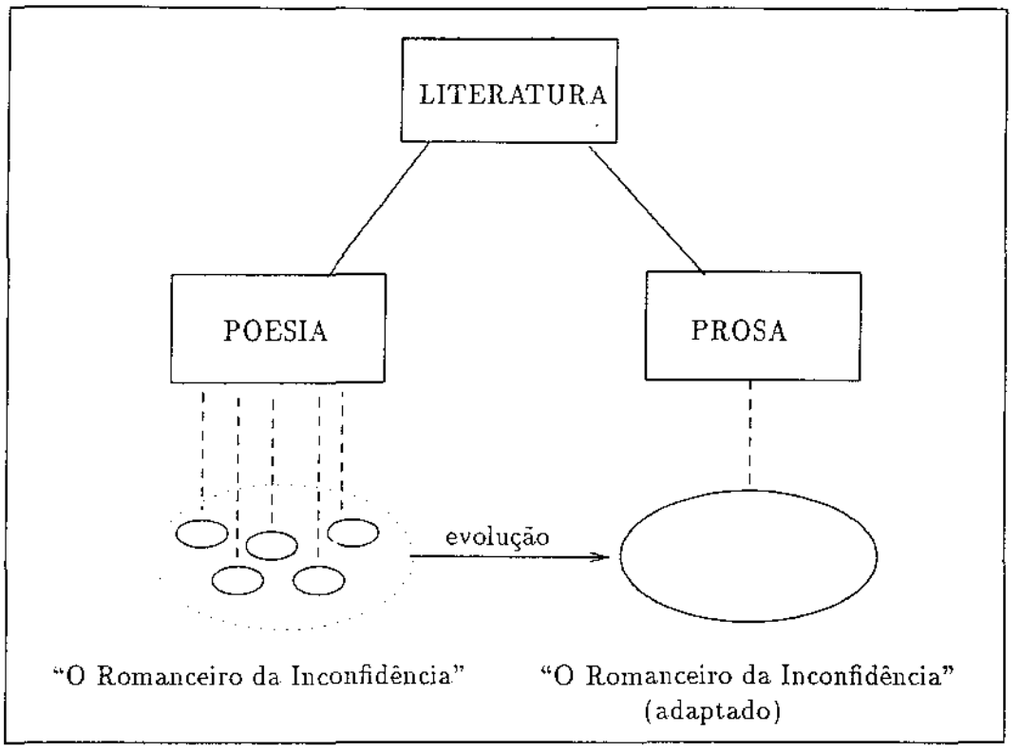


Figura 2.4: Evolução/Involução

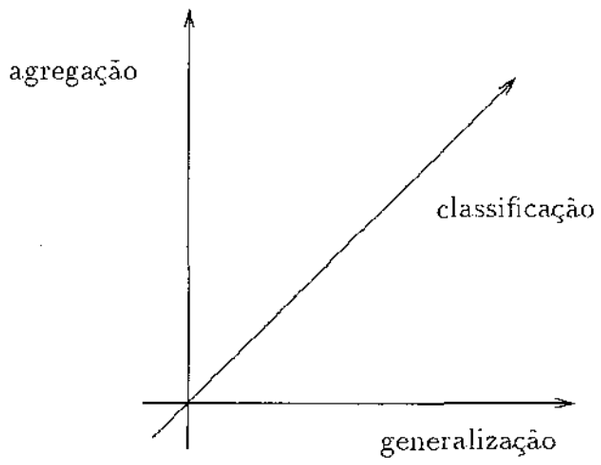


Figura 2.5: Ortogonalidade entre operações de abstração

objetos partem da visão de generalização/especialização. Por fim, implementações concretas de sistemas em linguagens operam no eixo classificação/instanciação.

No projeto concreto de sistemas, por mais importante que seja a escolha de uma das visões acima para a sua estruturação, o projetista alterna em sua mente operações básicas de todos os tipos e em vários níveis de granularidade.

A ênfase inicial em um eixo determinará a estrutura básica de um sistema, que, em geral, será mantida até o fim. A mudança de uma visão para outra no meio do processo é trabalhosa e, em geral, pouco útil. O apêndice B apresenta duas abordagens para o problema de mapear uma **visão funcional** para uma **visão de objetos** e ilustra as dificuldades conceituais encontradas nesse processo.

2.4.2 Mecanismos de Coordenação

Talvez o principal problema de um ambiente transformacional seja a explosão de opções de projeto a cada passo, constituídas:

- pelo conjunto de transformações possíveis e/ou desejáveis a cada instante, e
- pelo conjunto de projetistas e ferramentas disponíveis nesse passo, e
- pela complexa interação entre os diversos componentes do "quebra-cabeças".

O problema de coordenação, foi objeto de diversos estudos e propostas ao longo da década de 80, com resultados bastante satisfatórios.

Passamos a discutir aqui duas propostas que se conjugam para auxiliar na concepção e implementação de mecanismos de coordenação em ambientes transformacionais: *blackboards* e Teoria da Contingência.

Blackboards

O **modelo de** *blackboard* é um modelo de resolução de problemas[12] que prescreve a organização do conhecimento sobre domínio e de toda a entrada, das soluções parciais e intermediárias necessárias à resolução de um problema.

Para a discussão das principais características do modelo, dois modelos conhecidos são apresentados a seguir: o modelo de computação tradicional e o modelo clássico de sistemas especialistas.

- O modelo de computação tradicional consiste em um programa que atua sobre uma sequência de dados. Um programa é composto por um conjunto de procedimentos que, por sua vez, são conjuntos de comandos, cuja ordem de aplicação é determinada por mecanismos de controle conhecidos. O conhecimento sobre a estratégia de resolução do problema fica embutido nos procedimentos e estruturas de controle.

[12] Um modelo de resolução de problemas é um esquema para a organização de passos de raciocínio e conhecimento sobre domínio para a construção de soluções para um problema.

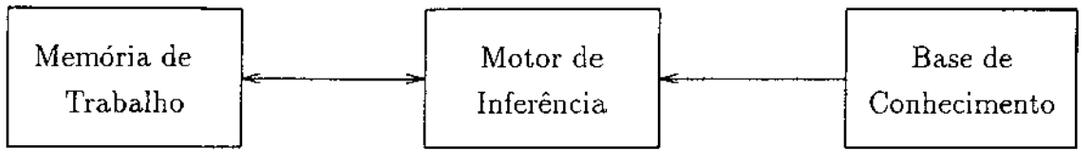


Figura 2.6: Exemplo da Estrutura Clássica de Sistemas Especialistas

- A estrutura clássica de sistemas especialistas, mostrada na figura 2.6 [Enge88], consiste em uma memória de trabalho, que guarda os dados de entrada para o sistema e soluções parciais e finais, um motor de inferência e uma base de conhecimento. O conteúdo da memória e as informações da base de conhecimento são usadas pelo motor de inferência para derivar novas hipóteses que são colocadas na memória de trabalho. Neste modelo, o conhecimento fica separado do motor de inferência que o utiliza e apresenta duas falhas principais:
  - O controle sobre a aplicação do conhecimento fica embutido na estrutura da base de conhecimento – e.g., na ordenação das regras, em sistemas baseados em regras.
  - A representação do conhecimento depende da natureza do motor de inferência – e.g., um interpretador de regras prescreve a representação sob a forma de regras.

O modelo de *blackboard* é considerado uma evolução natural que procura eliminar os pontos fracos da estrutura de sistemas especialistas [Enge88]. Nele,

- O conhecimento necessário à resolução de um problema é particionado em módulos, com um motor de inferência associado a cada um. Dessa forma, não se exige que as diversas partes da base de conhecimento compartilhem um mesmo tipo de representação nem que os motores de inferência utilizem o mesmo mecanismo de inferência.
- A comunicação entre os módulos é feita apenas através de operações de leitura/escrita na memória de trabalho.

O modelo descrito acima é mostrado na figura 2.7. O *blackboard* corresponde à memória de trabalho, enquanto que os pares (motor de inferência, base de conhecimento) correspondem às fontes de conhecimento.

Assim, o modelo de *blackboard* consiste em duas partes principais:

1. as **fontes de conhecimento**, os módulos independentes que encapsulam o conhecimento necessário à resolução de um problema.
2. o **blackboard**, a estrutura de dados global na qual são mantidos os dados e soluções de um problema. As fontes de conhecimento produzem mudanças no *blackboard* que levam à solução do problema de forma incremental. Toda a comunicação e interação entre elas é feita através do *blackboard*.

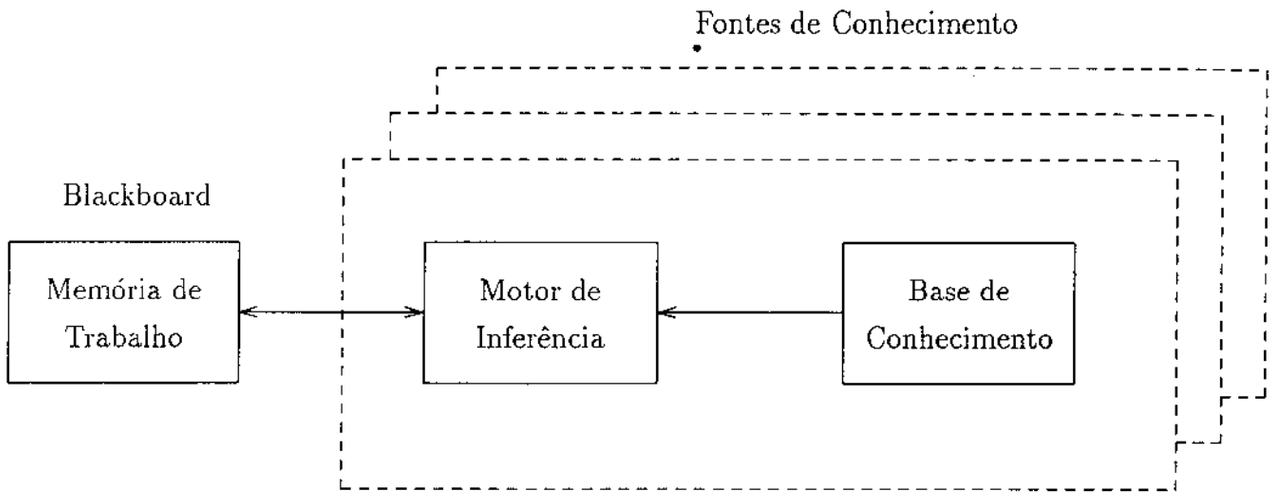


Figura 2.7: Exemplo de um Modelo de Blackboard Simples

Não há mecanismo de coordenação especificado no modelo de *blackboard*. O modelo simplesmente especifica um modo genérico de resolução: a resolução **oportunística** de problemas. Em um modelo de resolução oportunística, partes do conhecimento necessário à resolução de um problema são aplicadas no momento mais "oportuno". A execução de atividades é auto-motivada, pois cada agente conhece as condições em que pode colaborar, de acordo com as informações contidas no *blackboard*.

A figura 2.8 mostra o modelo de *blackboard*. Note-se que não há fluxo de controle: as fontes de conhecimento respondem a mudanças no *blackboard*, e se ativam autonomamente[13].

O exemplo que melhor ilustra a aplicação de resolução oportunística no modelo de *blackboard* é o de um grupo de pessoas que tenta montar um quebra-cabeças em um quadro colocado na parede de uma sala, como mostra a figura 2.9, extraída de [Nii86].

Cada pessoa - ou agente - tem em seu poder um conjunto de peças do quebra-cabeças. Alguns voluntários iniciam a montagem do quebra-cabeças indo até o quadro e colocando peças. Os outros agentes devem olhar para o quadro e para as peças que têm em seu poder e colocar aquelas que casam com partes das peças já colocadas no quadro. A montagem do quebra-cabeças é feita passo-a-passo pelos agentes, em completo silêncio e de forma oportunística (um agente vai até o quadro quando descobre uma oportunidade para adicionar uma de suas peças ao quadro). As informações de que cada agente necessita estão registradas no quadro e nas peças que possui.

O modelo de *blackboard* é uma entidade conceitual e não uma especificação computacional; a implementação de sistemas de *blackboard* requer a adequação do modelo às limitações de alguns sistemas de computação. Assim, foi definido um modelo de referência[14] para *blackboards*, cujo objetivo é o de prover diretrizes de projeto apropriadas para sistemas de *blackboard* em ambientes de computação não-paralelos.

[13] self-activating
[14] framework

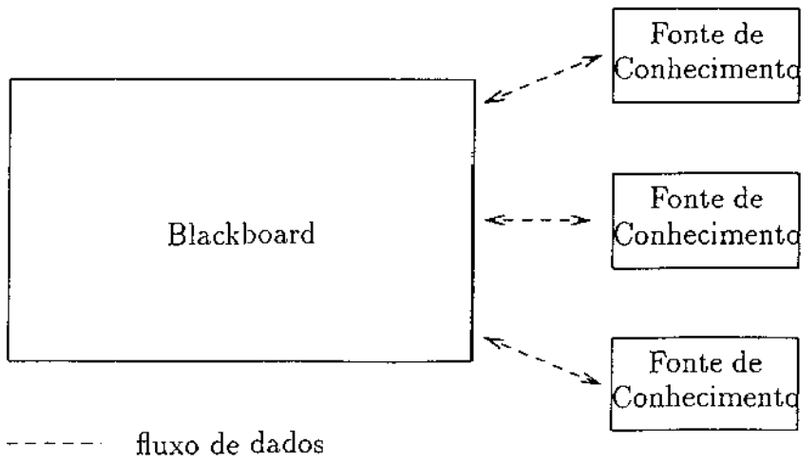


Figura 2.8: O Modelo de *Blackboard*

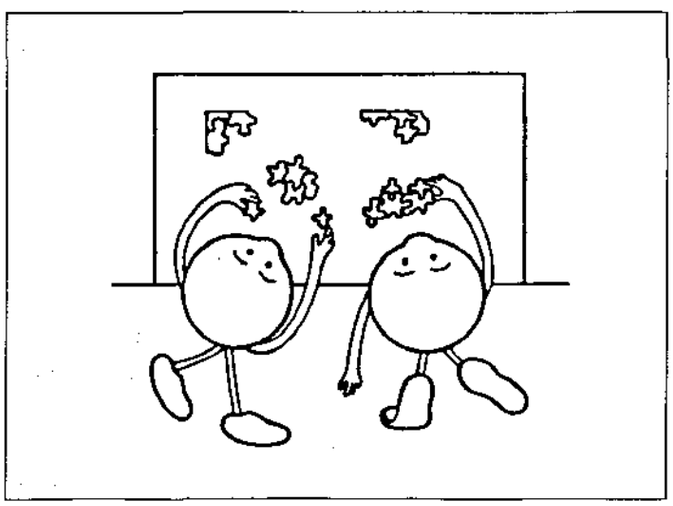


Figura 2.9: Resolução de Quebra-cabeças

O modelo de referência de *blackboards* adota a seguinte estratégia de controle:

- Há um conjunto de módulos de controle que faz o monitoramento das mudanças no *blackboard* e decide quais as próximas ações.
- Vários tipos de informação estão globalmente disponíveis para os módulos de controle no *blackboard*.
- A informação de controle é usada pelos módulos de controle para determinar o próximo objeto a ser processado[15] (quais as fontes de conhecimento a serem ativadas, quais os objetos do *blackboard* que serão manipulados ou uma combinação das duas anteriores).
- A solução é construída passo-a-passo, com a aplicação de qualquer tipo de estratégia de raciocínio, e a ativação das fontes de conhecimento é feita de forma dinâmica e oportunística.
- As atividades para resolução de problemas ocorrem na seguinte sequência:
  1. Uma fonte de conhecimento realiza mudanças em objeto(s) do *blackboard*. As mudanças efetuadas são registradas em uma estrutura de dados global que guarda informação de controle.
  2. Cada fonte de conhecimento indica a contribuição que pode dar ao novo espaço de soluções.
  3. Usando a informação fornecida nos passos anteriores, um módulo de controle seleciona um foco de atenção.
  4. A depender da informação contida no foco de atenção, um módulo de controle apropriado o prepara para a execução da seguinte forma:
     (a) Abordagem por escalonamento de conhecimento: se o foco de atenção for uma fonte de conhecimento, então um objeto do *blackboard* é selecionado para servir de contexto à sua ativação.
     (b) Abordagem por escalonamento de evento: se o foco de atenção for um objeto do *blackboard*, então uma fonte de conhecimento é selecionada para processar o objeto.
     (c) Se o foco da atenção for uma fonte de conhecimento e um objeto, então a fonte de conhecimento está pronta para execução com seu contexto definido (o objeto).
- São fornecidos critérios para se determinar quando terminar o processo.

Teoria da Contingência

Galbraith propôs a chamada Teoria da Contingência em um livro que discutia a **estruturação de organizações** de uma maneira geral. A idéia central era a de que a estrutura de uma organização se fundamentava no ataque a **fatores de incerteza** através de **respostas diversas**. As idéias de Galbraith foram apropriadas e conjugadas às de mecanismos de *blackboard* e aplicadas por Fox [Fox79] ao problema de desenvolvimento de software de larga escala, partindo da

[15] o foco de atenção

Incerteza	Métodos
consumidor	*broadcasting* sistemas de *blackboard*
produtor	identificação análise de contexto
função	gerência de recursos correção
...	...
comportamento	controle de conflitos oportunismo

Tabela 2.1: Métodos para atacar Incertezas

premissa que *estruturas de software podem ser condicionadas de forma similar a organizações humanas.*

A figura seguinte, extraída de [Brun86] resume a idéia de Fox: **métodos** específicos são acionáveis para atacar incertezas específicas (relativas a consumidores e produtores – agentes que consomem e produzem recursos, entre outros).

Por exemplo, a incerteza quanto a **consumidores** (identidade, localização, reação, etc.) pode ser atacada através de:

- envio de mensagem para **todos** os possíveis consumidores[16],
- postagem de informação através de um sistema de *blackboard*, supondo que todos os consumidores interessados consultarão o quadro,
- envio de mensagem seletivamente a alguns consumidores, supondo que estes farão algum tipo igualmente seletivo de retransmissão.

As incertezas quanto a **produtores** podem ser atacadas solicitando a **identificação** de todos os produtores ou através de **análise de contexto.**

2.5 Considerações Finais

Conforme discutido acima, há diversos resultados interessantes oriundos de esforços de pesquisa em áreas distintas e que parecem candidatos plausíveis para incorporação em um experimento de

[16] broadcasting

desenvolvimento transformacional: operações básicas de projeto e mecanismos de coordenação baseados em *blackboards*.

É válido perguntarmos qual a utilidade de embutir suporte a operações básicas em um ambiente sob o ponto de vista de coordenação. Conforme veremos no capítulo 6, um mecanismo básico de encadeamento regressivo que precede a adoção de quaisquer estratégias de coordenação, se vale da identificação de operações (i.e., relações) específicas entre objetos dentro de uma representação hierarquizada de informações sobre projeto.

Capítulo 3

Um Modelo Genérico do Processo de Software

3.1 Introdução

Neste capítulo, discutiremos a natureza de atividades genéricas que têm lugar no processo de software, através de uma estrutura conceitual unificadora: o modelo PW.

3.2 O Modelo PW em Detalhe

Como foi dito anteriormente, o modelo PW é um modelo genérico do processo de software, que em sua versão mais abstrata, o decompõe em dois processos com características distintas: um de abstração, que transforma o conceito da aplicação (ou **aplicação conceitual**) em uma especificação formal, e outro de reificação, que transforma a especificação obtida em um programa.

Na prática, dificilmente uma especificação completa do conceito da aplicação será concebida como um todo, uma estrutura monolítica. É provável que a especificação consista de um conjunto de subespecificações que, devidamente estruturadas, irão compor a especificação global desejada. Cada subespecificação obtida é uma **especificação parcial** e um ponto de partida para o processo de reificação, como mostram as figuras 3.1 e 3.2.

Os processos de abstração e reificação ainda são muito complexos para serem realizados em um só passo. Muito provavelmente, eles passarão por decomposições sucessivas, obtendo-se, assim, uma sequência de sub-transformações que, em conjunto, realizam os processos correspondentes. A figura 3.1 mostra o segundo nível de decomposição do modelo PW.

3.2.1 A Perna Esquerda

A perna esquerda do modelo PW representa a atividade de criar uma especificação formal a partir do conceito da aplicação. Neste processo, muito provavelmente, serão aplicadas recorrentemente as operações básicas descritas no capítulo anterior, com o intuito de modelar o conceito da aplicação e derivar uma especificação satisfatória.

Uma das operações básicas utilizadas na perna esquerda é a **decomposição**. O seu objetivo é o de transformar um problema complexo em um conjunto de problemas mais simples. O

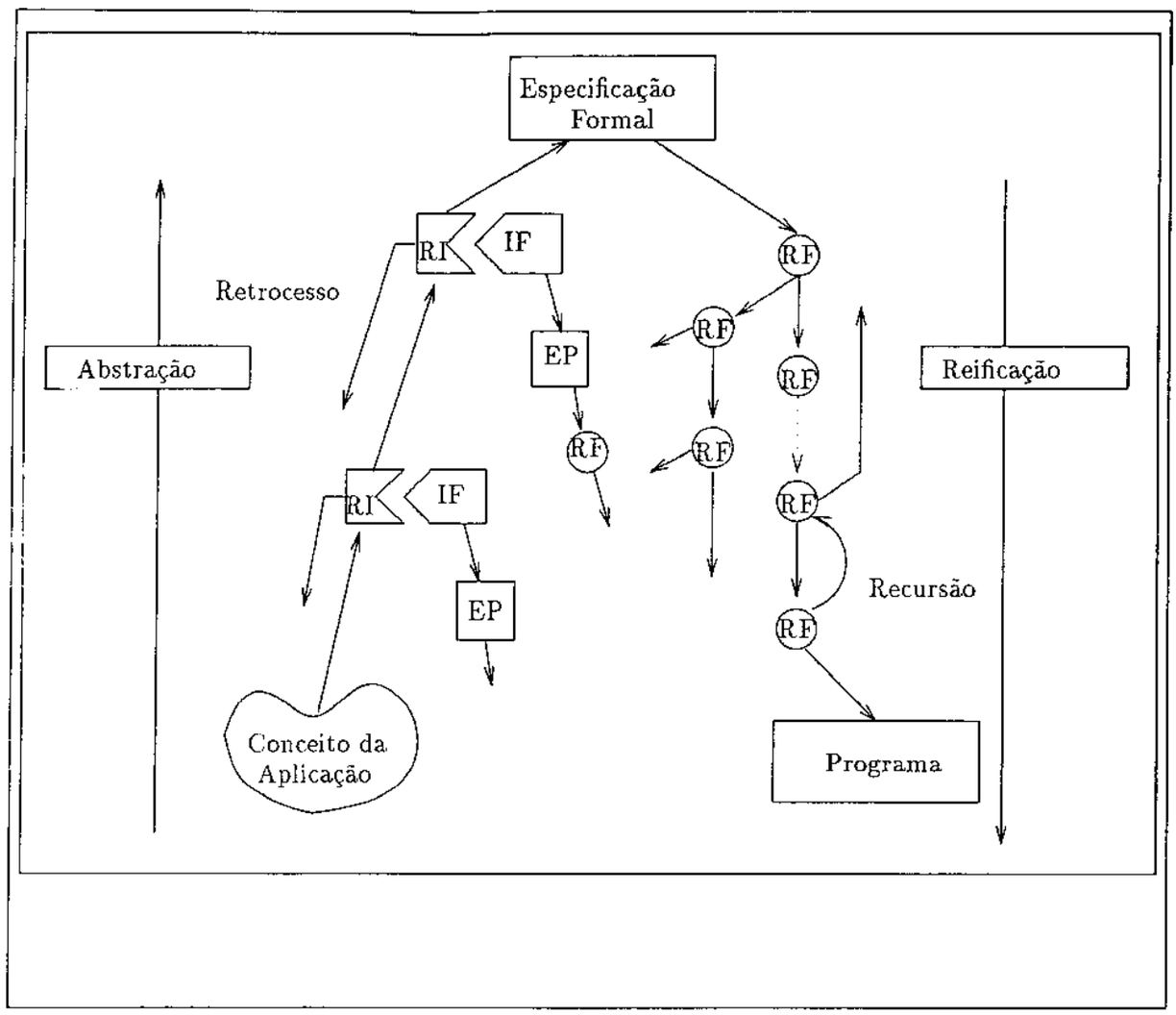


Figura 3.1: Segundo Nível de Decomposição do Processo de Desenvolvimento

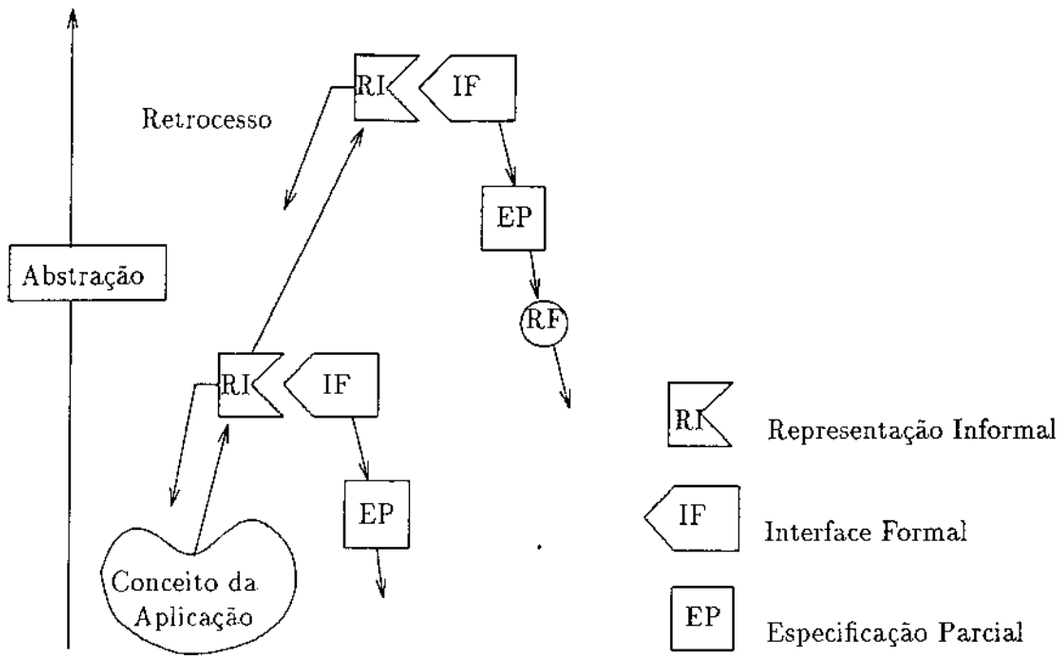


Figura 3.2: A Perna Esquerda

sucesso de uma decomposição depende da escolha adequada dos componentes. Esta escolha deve se basear em uma visão genérica do conceito da aplicação, obtida através de operações de abstração que permitem a criação de modelos genéricos do conceito da aplicação e seus componentes.

Nestas circunstâncias, o processo representado pela perna esquerda toma a forma de **decomposição via abstração**, onde problemas são decompostos em subproblemas através da identificação de abstrações úteis.

Assim, o problema inicial é decomposto em subproblemas que podem ser tratados mais facilmente e de maneira mais ou menos independente. A partir de cada subproblema são obtidas representações informais parciais (RI), a partir das quais serão construídas especificações formais parciais (EP). A especificação formal que aparece no vértice do "V" invertido da figura 3.1 é, então, um conjunto estruturado de especificações parciais, que representam pontos de partida para vários processos de reificação.

A perna esquerda é formada por especificações formais parciais obtidas de representações informais. Cada RI serve como base para o passo seguinte, onde o processo descrito acima é repetido. Além disso, interfaces formais (IF) provêem uma ligação entre cada especificação formal obtida e sua representação informal correspondente, como mostra a figura 3.2. O objetivo de cada IF é o de relacionar os termos teóricos da EP à qual está ligada aos objetos informais que compõem a RI.

Como em qualquer processo de desenvolvimento realizado passo-a-passo, pode haver necessidade de retrocesso no processo realizado na perna esquerda, para permitir a revisão e modificação

de decisões prévias que conduziram a situações indesejadas. O procedimento de retrocesso deve permitir a revisão das interfaces formais e especificações parciais obtidas nos passos anteriores.

3.2.2 A Perna Direita

A perna direita do modelo PW é conhecida como **modelo LST do processo de reificação** e representa o processo de construção de um programa a partir de uma especificação formal (possivelmente composta por várias especificações parciais). Este processo, apesar de possuir natureza diversa, é realizado através da aplicação dos mesmos pares de operações básicas de projeto válidos para a perna esquerda.

A construção de um programa raramente cobre toda a distância entre o sistema linguístico[1] da especificação e o sistema linguístico do programa em apenas um passo. Geralmente, a construção prossegue através de uma sequência de **passos** intermediários. Nem o número nem a natureza dos passos podem ser fixados *a priori*.

Cada passo representa uma transformação entre dois sistemas linguísticos: um sistema linguístico de nível mais alto (fonte) - a partir de agora denotado por SL_i - e um sistema linguístico de nível mais baixo (objeto) - denotado por SL_{i+1}.

Um passo se inicia com um sistema linguístico fonte pré-estabelecido e um conjunto estruturado de sentenças que corresponde à representação formal do estágio corrente do programa em desenvolvimento - a base corrente R_i. A representação desse conjunto de sentenças no sistema linguístico fonte corresponde à especificação para o passo corrente e deve possuir as propriedades[2] fundamentais a qualquer especificação - e.g., correção e consistência[3].

O sistema linguístico objeto pode ou não ser definido *a priori*. [Lehm84b] considera que a seleção do SL objeto é o aspecto criativo do passo de construção de programas e constitui a principal decisão de projeto de um passo[4]. Através da seleção de um sistema linguístico objeto, o projetista decide quais os conceitos extralógicos que serão adicionados ao vocabulário,

[1] Um sistema linguístico é um sistema formal para a expressão de idéias e consiste de três partes:

- uma gramática G
- um sistema lógico I, composto por um conjunto de axiomas lógicos e um conjunto de regras de inferência
- um conjunto de axiomas extralógicos A.

Em geral, as sentenças dos sistemas linguísticos são definidas de forma indutiva a partir de construções mais simples, onde as partes constituintes são definidas pelas regras de produção no estilo BNF da gramática G. As regras de inferência são usadas para se tirar conclusões e derivar novas sentenças.

As sentenças de um sistema linguístico, conhecidas como **tautologias** ou **axiomas lógicos**, são consideradas verdadeiras independente do domínio de aplicação representado, formando um núcleo a partir do qual são derivadas novas sentenças do sistema linguístico através da aplicação de regras de inferência. Estas sentenças são consideradas puramente lógicas. Porém, a veracidade ou falsidade de outro grupo de sentenças depende de suposições que não podem ser verificadas apenas através de raciocínio no escopo do próprio sistema linguístico, sendo denominadas **axiomas extralógicos**. Em particular, quando um sistema linguístico é usado para a descrição de um domínio do mundo "real", a veracidade de algumas sentenças deve ser determinada naquele domínio de aplicação específico.

Tanto a especificação como o programa são conjuntos de sentenças em seus respectivos sistemas linguísticos.

[2] As propriedades devem ser verificadas a partir dos axiomas do sistema linguístico pela aplicação das regras de inferência.

[3] A consistência de uma especificação é uma propriedade que assegura a ausência de informação contraditória.

[4] Apenas o sistema linguístico da especificação e o do programa são estabelecidos *a priori*.

os conceitos primitivos que serão decompostos, etc. Uma vez escolhido SL_{i+1}, o passo corrente pode prosseguir através da criação de um conjunto adequado de sentenças representado em SL_{i+1} - a representação R_{i+1}.

A representação corrente R_i deve permitir dupla interpretação: como um programa - implementação correta em relação a R_{i-1} expressa em S_{i-1} - e como uma especificação para o próximo passo e para a representação R_{i+1}. Ao final de cada passo, deve-se demonstrar a correção de R_{i+1} em relação R_i, a menos que o próprio processo crie uma representação R_{i+1} correta por construção, mediante transformações que preservam a correção.

Assim, dados uma representação corrente R_i, o sistema linguístico objeto SL_{i+1} e uma obrigação de verificação, o passo corrente pode ser completado, gerando um dos resultados abaixo:

1. um resultado **negativo**, ou seja, demonstra-se de forma convincente que nenhuma implementação correta de R_i pode ser obtida em SL_{i+1}.

2. um resultado **positivo**, ou seja, a representação corrente R_{i+1} é obtida, correta em relação a R_i. Este resultado pode ser submetido a uma análise de aceitação empírica (validação), podendo ser considerado **aceitável** ou **inaceitável**.

A especificação inicial do problema é distinta das representações subsequentes em apenas um aspecto: não há obrigação de mostrar que ela é uma implementação correta para alguma outra especificação. Esta obrigação é substituída pela demonstração de que tal especificação é uma abstração adequada do conceito da aplicação.

Como o objetivo do processo de reificação é a obtenção de um programa "executável", é normal imaginarmos que os sistemas linguísticos escolhidos são, a cada passo, cada vez mais concretos[5]. Assim, podemos afirmar que a transformação entre dois sistemas linguísticos envolve:

- *o mapeamento entre as contruções envolvidas*, ou seja, a representação dos objetos e operações do primeiro em termos de objetos e operações do segundo sistema;

- *a modificação da granularidade da descrição* - geralmente, um objeto (operação) do primeiro é mapeado para um conjunto de um ou mais objetos (operações) no sistema seguinte;

- *o acréscimo de termos e/ou fenômenos* a representações expressas no segundo sistema linguístico, não representáveis no primeiro sistema.

O passo descrito acima é **canônico** [Turs87] e é considerado a unidade de trabalho construtiva do modelo LST.

A Necessidade de Retrocesso

A construção de programas não é um processo monotônico de transformação entre sistemas linguísticos. Em certas circunstâncias, pode acontecer que a representação R_{i+1} obtida, apesar de correta, possua características indesejáveis (ou não possua algumas desejáveis), ou mesmo

[5]Neste contexto, as construções do sistema linguísticos estão próximas das construções da linguagem da máquina subjacente.

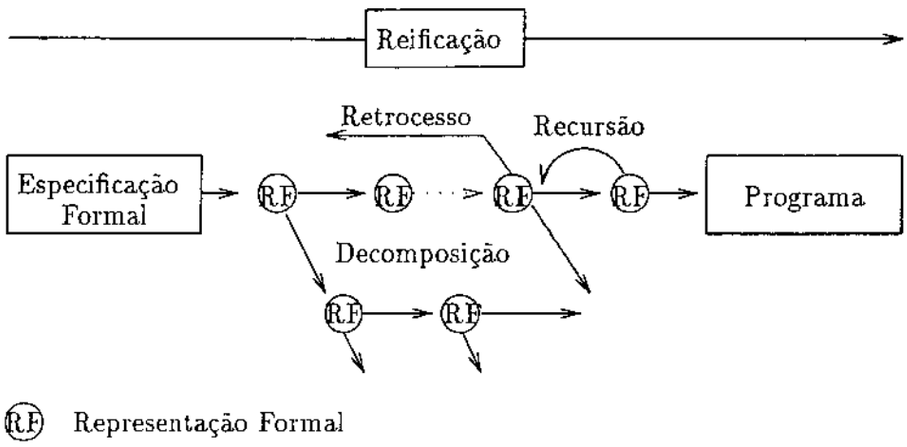


Figura 3.3: O Modelo LST do Processo de Reificação

que tal representação não consiga ser obtida em SL_{i+1}. Nestes casos, é necessário aplicar um procedimento de **retrocesso** que permita o retorno a um sistema linguístico SL_j, com $j < i$, tal que uma modificação feita em R_j (que preserve a correção em relação a R_{j-1}) leve à remoção da causa que provocou o retrocesso.

Uma abordagem inicial poderia ser a de tentar redefinir o sistema linguístico objeto SL_i de forma a permitir que se crie nele uma representação R_i correta em relação a R_{i-1}. Esta abordagem não é vista como um procedimento de retrocesso e sim como uma tentativa recorrente de reparar SL_i. O procedimento de **recursão** é mostrado na figura 3.3.

Uma segunda abordagem consiste em, após o retrocesso, tentar modificar a representação R_j construída em SL_j.

Tendo retrocedido do nível i ao nível j, deve- se, em princípio, desprezar os passos $j+1$, ..., $i-1$, já que uma modificação em R_j pode levar a uma escolha distinta para SL_{j+1}.

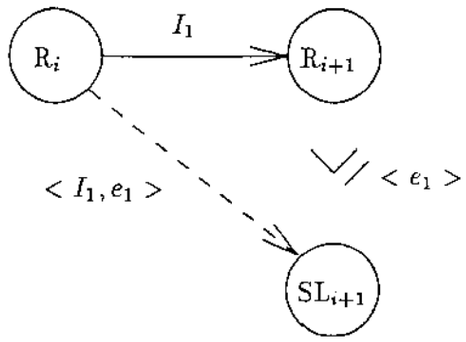
As modificações descritas até agora são denominadas de **endógenas**, pois a necessidade de incorporá-las surgiu no decorrer do processo de construção do programa propriamente dito.

Além das endógenas, há outro tipo de modificação, decorrente de mudanças no conceito da aplicação, que são denominadas de **exógenas**.

3.3 O Passo Canônico

Seja R_i a representação corrente construída sobre o sistema linguístico SL_i, após i aplicações do passo canônico (descrito anteriormente) à especificação inicial.

A representação corrente R_i é denominada de **nível linguístico**. Um nível linguístico é uma extensão do sistema linguístico subjacente através do enriquecimento da sua parte extralógica. Sendo assim, um passo consiste na implementação da representação corrente R_i (expressa através de SL_i) em um sistema linguístico objeto SL_{i+1}, gerando uma representação R_{i+1}. Uma **implementação** de R_i em SL_{i+1} é um par $< I_1, \epsilon_1 >$, como mostra a figura 3.1.



e_1: é uma extensão de R_i em R_{i+1}
I_1: é uma interpretação entre as representações.

Figura 3.4: O Passo Canônico

Um passo canônico consiste, então, em três fases, a saber [Haeb89]:

1. definição do sistema linguístico objeto.
2. extensão do sistema linguístico objeto a nível linguístico.
3. tradução entre o nível linguístico fonte e o nível linguístico objeto.

A **extensão** de um sistema linguístico a nível linguístico corresponde à utilização do sistema linguístico para definir novas construções, e.g., tipos, operações, etc. A **tradução** entre dois níveis linguísticos adjacentes consiste de duas atividades:

- o estabelecimento de interpretações para traduzir os elementos das representações (constantes, variáveis, procedimentos, etc.).
- a demonstração de que as interpretações definidas permitem que a tradução produza uma **extensão conservativa** entre as representações, ou seja, que todas as consequências deriváveis no nível fonte também o são no nível objeto.

O que foi exposto acima delineia uma forma canônica para os passos individuais em um processo de reificação passo-a-passo. O processo de reificação é, então, potencialmente coerente, pois descreve o processo que produz uma sequência de representações formais em termos de um único paradigma.

3.4 Operações de Projeto e o Modelo PW

As operações de projeto descritas no capítulo 2 são suficientemente gerais para serem aplicáveis tanto no processo de abstração representado pela perna esquerda do modelo PW, como no processo de reificação. Na perna esquerda, a granularidade das operações é **grossa**, posto que o sistema ainda está sendo gradativamente estruturado. Na perna direita, a granularidade é **fina**,

posto que as transformações operam sobre descrições mais concretas de sistemas, mapeando-as para níveis linguísticos cada vez mais detalhados. É razoável supor que cada transformação na perna direita possa ser representada pela **composição de inúmeras operações básicas de projeto em alguma sequência específica** e acrescentando-se semântica apropriada.

Generalizando a suposição feita acima, podemos imaginar que, dada qualquer transformação específica de qualquer método, será possível "traduzir" tal transformação em uma sequência de operações básicas.

3.5 Considerações Finais

O modelo PW enfatiza a formalização do processo de reificação, prescrevendo, entre outros, a utilização de uma especificação formal do conceito da aplicação como ponto de partida para o processo e o descrevendo como uma combinação de passos canônicos.

A noção de construção de programas sob a ótica de transformação entre sistemas linguísticos – mediante a aplicação de passos canônicos – certamente captura um aspecto importante da essência dos processos de projeto e implementação de software e proporciona uma base conceitual unificadora para métodos de desenvolvimento de software[6].

Vale a observação de que o sentido de "canônico" neste contexto não é o mesmo dado à derivação canônica ou à forma canônica em linguagens formais. A definição do processo de reificação como uma combinação de passos canônicos não implica um processo de derivação determinístico de um programa a partir de uma especificação formal.

o o o

Por fim, neste trabalho, adotaremos a suposição feita na seção anterior, considerando qualquer transformação de qualquer método como sendo uma sequência de operações básicas de projeto. Isso permitirá abstrairmo-nos de métodos específicos para a investigação de ambientes genéricos de apoio a desenvolvimento transformacional, abordando tanto o processo de abstração como o de reificação.

[6]Turski comenta que, segundo essa ótica, a distinção entre projeto e implementação de programas torna-se irrelevante – pelo menos conceitualmente.

Capítulo 4

Transformações

4.1 Sistemas Transformacionais

À medida em que a tecnologia de Compiladores foi sendo dominada e solidificada, imaginou-se que ela poderia ser estendida para cobrir um estágio anterior no processo de desenvolvimento de software – ou mesmo para cobrir todo o processo – em especial para transformar uma especificação de alto nível, escrita em alguma linguagem de especificação, em um programa expresso em alguma linguagem de programação. Houve trabalhos interessantes, principalmente no sentido de prover transformações automáticas para certos domínios limitados e bem definidos, como foi o caso dos geradores de aplicações. Entretanto, para os casos mais genéricos, há um grande número de pesquisas, porém com resultados ainda modestos.

Esta seção constitui-se, em sua maior parte, em uma revisão baseada em [Part83], um *survey* sobre sistemas transformacionais que estabelece uma classificação arbitrária – segundo os autores - em função de alguns sistemas transformacionais conhecidos e implementados.

4.1.1 Terminologia Básica

Define-se **programação transformacional** como sendo um método de construção de programas através da aplicação sucessiva de regras de transformação que preservam a correção.

Uma **transformação** pode ser vista como uma função cujos argumentos e valores são programas ou esquemas de programas. O conceito de **programa** é o convencional: um algoritmo (sequência de passos bem definidos para resolver um determinado problema) expresso em alguma linguagem de programação. Um **esquema** (de programa) é uma representação de uma classe de programas afins, obtido a partir da abstração de algumas de suas propriedades.

Neste paradigma, a especificação formal dos requisitos é a base para a construção de programas corretos e eficientes mediante transformação gradual.

Seja θ uma transformação entre dois esquemas (ou programas) π e π'. θ é dita correta se alguma relação semântica φ for mantida entre π e π'. A relação semântica mais importante no contexto de transformações é a de **equivalência**. Outras relações importantes são: **equivalência fraca** (não se responsabiliza pelos efeitos de transformações sobre esquemas de programa incorretos) e a de **descendência** para programas não- determinísticos (é suficiente que π' esteja incluído em π, isto é, que os possíveis resultados de π' sejam um subconjunto dos possíveis resultados de π).

Qualquer que seja a relação semântica mantida entre os esquemas π e π' após a aplicação de uma transformação, ela deve ser reflexiva (pois a identidade deve ser uma transformação válida), transitiva (para permitir a aplicação sucessiva de transformações) e monotônica (para permitir transformações válidas sobre uma parte ψ de um esquema π).

O processo de desenvolvimento transcorre mediante a aplicação de transformações que derivam um programa equivalente à especificação formal. Para que a correção seja preservada, são utilizadas regras de transformação. Uma **regra de transformação** é um mapeamento parcial entre esquemas, correspondendo a transformações corretas. Por ser um mapeamento parcial, condições de ativação[1] – predicados sobre esquemas de programas – são usadas para restringir o domínio da regra de transformação.

Os formatos básicos de uma regra de transformação são:

1. $a \Longleftrightarrow b(c)$,

   onde a e b são esquemas e c é uma condição, significando que b pode substituir a (e vice-versa) se a condição c for verdadeira.

2. $a \Longrightarrow b(c)$,

   significando que b pode substituir a se c for verdadeira.

Há três tipos de condições de ativação [Broy81]: as **condições sintáticas** apenas estabelecem que certas variáveis do esquema valem para algumas entidades sintáticas. As **condições de contexto** geralmente são predicados tais como "OCCURS (x in E)", afirmando que o identificador x ocorre livremente na expressão E. E, finalmente, as **condições semânticas**, que, em geral, não podem ser verificadas automaticamente e necessitam de interação com o programador/usuário.

A idéia central do método de transformação de programas é que, em geral, um definição recursiva de um problema é a expressão mais natural da definição desse problema. A partir desta versão inicial e pela aplicação de transformações, tanto em direção a um nível mais baixo de abstração, quanto pela aplicação de transformações em um mesmo nível linguístico (em busca de versões mais eficientes do algoritmo), o programa passa por versões intermediárias até que seja obtida uma versão eficiente do problema – implementação final – equivalente à definição original.

4.1.2 Modelos

A introdução do paradigma transformacional como método de desenvolvimento de software levou à criação de novos modelos descritivos para caracterizá-lo. Tais modelos são variações sobre um mesmo tema: o modelo clássico do ciclo de vida do software mostrado no primeiro capítulo.

Um modelo do ciclo de vida que corresponde ao paradigma transformacional é mostrado na figura 4.1 [Haeb89].

Os aspectos a destacar neste modelo são:

- ênfase na especificação formal;
- manutenção/validação feitas sobre a especificação formal;

[1] enabling conditions

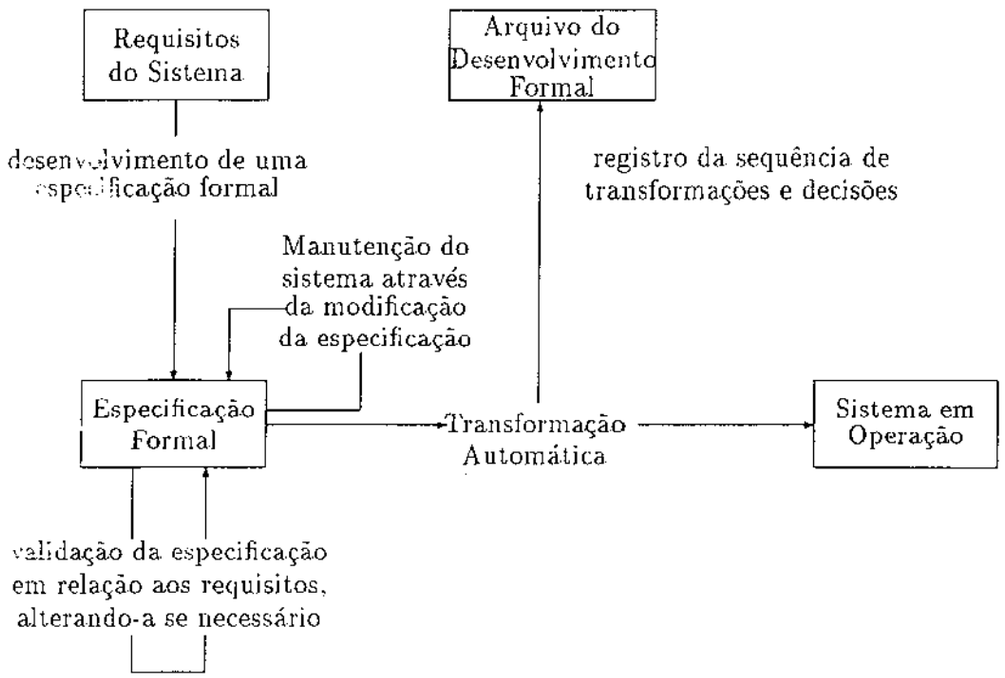


Figura 4.1: O Modelo do Paradigma Transformacional

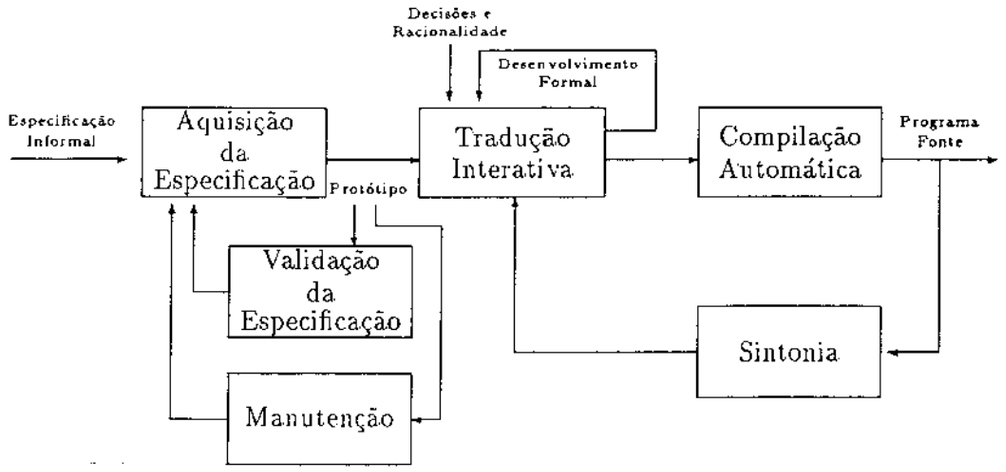


Figura 4.2: O Modelo de Balzer

- utilização de um sistema automático para a aplicação da transformação;
- utilização de um Sistema Especialista para guiar interativamente a seleção de transformações;
- registro do desenvolvimento formal através de um histórico de derivações.

Um outro modelo conhecido é o de Balzer, mostrado na figura 4.2 [Luce87], que, além dos aspectos destacados acima, enfatiza também a criação de um protótipo, que será usado para validação e como ponto de partida para a obtenção de um programa concreto, mediante transformações automáticas.

4.1.3 Papéis no Processo de Desenvolvimento de Software

Os sistemas transformacionais podem desempenhar diferentes papéis no processo de desenvolvimento de software. Em geral, pode-se classificá-los segundo este aspecto em dois grupos:

1. Sistemas **próprios**, onde o aspecto transformacional tem uma posição de destaque no sistema.

   Neste grupo, podemos identificar dois tipos de sistemas:

   (a) genéricos
   São sistemas que não estão restritos a tipos específicos de regras de transformação nem a uma fase ou aspecto específicos do processo de desenvolvimento de software.

   (b) de propósito específico
   São sistemas transformacionais com um objetivo específico. Os sistemas deste tipo podem ser caracterizados pelo tipo de atividade que desempenham (síntese, otimização, verificação, etc.) e pelos tipos de programas que manipulam.

2. Sistemas **genéricos para o desenvolvimento de software**, onde o aspecto transformacional e o seu desempenho correspondem apenas a uma pequena parte do sistema como um todo.

4.1.4 Objetivos

Os sistemas transformacionais utilizam transformações com um objetivo específico ou uma combinação deles.

O objetivo mais comum é o de dar suporte genérico à **modificação de programas**. Isto inclui a otimização de estruturas de controle, a implementação eficiente de estruturas de dados e a adaptação de programas a estilos específicos de programação. Em geral, a motivação de sistemas transformacionais com este objetivo é a de permitir que programadores escrevam seus programas para a resolução de um problema, sem preocupações com questões mais específicas, tais como eficiência, estilo de programação, etc.

Um segundo objetivo é a **síntese de programas**, ou seja, a geração de um programa a partir de uma descrição inicial do problema. Sistemas transformacionais com este objetivo variam de acordo com a formulação de sua entrada. A síntese pode ser iniciada a partir de especificações em linguagem natural restrita ou a partir de especificações de alto nível expressas em alguma linguagem formal.

Outros objetivos são: **adaptação de programas** a ambientes específicos, **descrição de programas** (através da exibição do seu histórico de derivação) e **verificação de programas**.

4.1.5 Aspectos Relevantes

O papel que os sistemas transformacionais desempenham no processo de desenvolvimento de software e os seus objetivos são aspectos importantes para a sua caracterização. Além desses, existem outros aspectos relativos a estes sistemas que também devem ser mencionados.

- **Organização das regras de transformação**

  A maior parte dos sistemas possui uma coleção pré-definida e possivelmente extensível de regras de transformação.

  Há duas abordagens para a organização da coleção de regras de transformação em um sistema transformacional: através de um **catálogo de regras** ou de um **conjunto gerador**.

  - Catálogo de Regras

    É uma coleção estruturada de forma linear ou hierárquica de regras de transformação, em geral específicas a um domínio de aplicação ou a um aspecto específico do processo de desenvolvimento. Os sistemas que utilizam esta abordagem são chamados de *knowledge-based systems*. Os problemas principais desta abordagem estão relacionados a sua completude e estrutura. Em particular, faz-se necessário garantir a existência, a disponibilidade e o acesso rápido às regras.

  - Conjunto Gerador

    Consiste em pequeno conjunto contendo regras de transformação elementares e poderosas, independentes de um domínio ou linguagem específicos, utilizadas como base

para a construção de novas regras. Em geral, este conjunto é utilizado em conexão com outros conjuntos de regras específicas aos domínios envolvidos. O maior problema enfrentado nesta abordagem é o de decidir a **ordem de aplicação das regras elementares** para realizar transformações mais amplas.

- **Adição de novas regras**

  Em geral, os sistemas transformacionais permitem a extensão do conjunto de regras de transformação disponível. A maior parte deixa a cargo do usuário esta tarefa, inclusive em relação à garantia de correção das novas regras. Poucos sistemas permitem a prova ou derivação automática de novas regras, corretas por construção, através de mecanismos de composição e verificação.

- **Seleção de regras e de locais de aplicação**

  Todos os sistemas transformacionais aplicam, sucessivamente, regras de transformação em algum local. Nas implementações mais simples, a responsabilidade da seleção de regras a serem aplicadas e do local de aplicação fica a cargo do usuário. O sistema simplesmente permite ou não a aplicação de uma regra mediante a avaliação da condição de ativação associada a ela.

  Sistemas **semi-automáticos** realizam transformações automaticamente para algumas tarefas pré-definidas, consultando o usuário sempre que se defrontam com questões que não podem resolver.

  Sistemas **totalmente automáticos**[2], manipulam satisfatoriamente aplicações em domínios restritos, sempre determinando a regra e o local da aplicação automaticamente, através de heurísticas embutidas no sistema.

  Alguns sistemas provêem algum tipo de orientação para a seleção das próximas regras que podem ser aplicadas e de possíveis locais para a aplicação.

- **Documentação do processo de desenvolvimento**

  Um dos pontos fortes do paradigma transformacional é o suporte à documentação do processo de desenvolvimento, permitindo a reutilização de decisões de projeto. O suporte dado inclui o registro do programa fonte, da saída, de versões intermediárias e das regras selecionadas e aplicadas durante o processo. A documentação pode ser feita através de mecanismos simples - *logs* das sessões - ou mais sofisticados, utilizando bases de dados. Os sistemas mais inteligentes são capazes de refazer automaticamente o desenvolvimento a partir de uma especificação levemente modificada, reutilizando as informações registradas relativas a decisões de projeto (expressas através das seleções de regras). O registro do processo de desenvolvimento é conhecido como **histórico de derivação**[3].

- **Tipos de linguagens manipuladas**

  Muitos sistemas são conceitualmente independentes dos tipos de linguagens ou estilos de programas manipulados, apesar de que, na sua implementação, a maior parte acaba por se prender a algum estilo ou tipo específicos.

[2] fully automatic
[3] refinement history

Em geral, há dois tipos básicos de linguagens manipuladas pelo sistema: **linguagens de especificação**, que dão suporte à especificação formal do problema a ser resolvido e **linguagens de programação**, usadas para a formulação da solução do problema.

Alguns sistemas utilizam apenas uma única linguagem durante todo o processo de desenvolvimento, de espectro suficientemente amplo para conter diversos estilos de programação sob uma única estrutura sintática. Estas são conhecidas como **linguagens de largo espectro**[4].

- **Grau de interação com o usuário**

  Praticamente todos os sistemas transformacionais são **interativos**. Mesmo aqueles com maior nível de automatização requerem algum tipo de entrada inicial para o sistema (geralmente na forma de um objetivo a ser alcançado) e dependem da decisão do usuário para a resolução de situações imprevistas.

4.1.6 Tipos de Regras de Transformação

Em [Part83] encontramos a seguinte classificação – baseada em contexto e tipo de representação da regra – para regras de transformação:

- Regra Esquemática

  É uma regra de transformação representada como um par ordenado de esquemas de programa, geralmente separados por algum símbolo que indica substituição ("$\Longleftrightarrow$" para equivalência e "$\Longrightarrow$" para descendência).

  É uma representação orientada por sintaxe, adequada à percepção humana, mas inadequada para expressar conhecimento semântico ou informação global. Esse tipo de regra é usado geralmente num contexto local.

- Regra Procedural

  É uma regra de transformação – representada através de um algoritmo – que, a partir de um programa produz um novo programa (um compilador, por exemplo).

  É geralmente utilizada em um contexto global.

- Regra Global

  Também conhecidas como regras semânticas, fazem análise de fluxo, verificações de consistência, operações de *cleanup* global ou representação de técnicas e paradigmas de programação (por exemplo, "dividir-para-conquistar").

- Regra Local

  São regras aplicadas localmente e servem, entre outras coisas, para:

  1. Relacionar construções da linguagem (regras sintáticas), tais como:
     L: if B then S; goto L fi $\Longleftrightarrow$ while B do S od

[4] wide-spectrum languages

2. Descrever propriedades algébricas que relacionam construções distintas da linguagem, como:
   1 + if B then X else Y fi $\Longleftrightarrow$ if B then 1 + X else 1 + Y fi
3. Expressar conhecimento sobre um domínio na forma de propriedades de tipos de dados, tais como:
   pop(push(s,x)) $\Longleftrightarrow$ s (para uma pilha ilimitada)
   b $\wedge$ b $\Longleftrightarrow$ b (para booleanos)

- Regra Híbrida

  São regras de transformação que se situam entrem as locais e as globais. São usadas para codificar algum conhecimento de programação específico. Duas regras híbridas comuns são as de *Unfold* e *Fold*. As regras híbridas também são utilizadas para representar detalhes de implementação, tais como: "uma pilha com limite deve ser implementada através de um vetor e um índice".

4.1.7 Exemplos de Regras de Transformação

1. Regras para a manipulação de expressões booleanas
   (a) Associatividade do $\wedge$
   (A $\wedge$ B) $\wedge$ C $\Longleftrightarrow$ A $\wedge$ (B $\wedge$ C)
   (b) Lei de De Morgan
   $\neg$ (A $\wedge$ B) $\Longleftrightarrow$ $\neg$ A $\vee$ $\neg$ B
2. Regras para condicionais
   if B then C_1 else C_2 endif $\Longleftrightarrow$ C_1,(valor(B) = verdade)
3. Regras para o nível procedural
   (a) Definição do TO
   to c do S enddo $\Longleftrightarrow$ if c = 0 then "nop" else to c-1 do S enddo; S
   (b) Soma de iterações
   to c do S enddo; to d do S enddo $\Longleftrightarrow$ to c+d do S enddo

4.1.8 Uma Taxonomia para Transformações

Em função da definição do conjunto básico de primitivas de projeto proposto anteriormente – que constitui um núcleo de transformações para um ambiente de apoio ao desenvolvimento – estabelecemos a seguinte classificação genérica para transformações:

- transformações de classificação/instanciação
- transformações de generalização/especialização
- transformações de agregação/decomposição
- transformações de evolução/involução

Características	Abordagens para Reusabilidade				
Componente Reutilizado	Blocos de Construção		Padrões		
Natureza do Componente	Atômico e Imutável Passivo		Difuso e Maleável Ativo		
Princípio de Reutilização	Composição		Geração		
Ênfase	Bibliotecas de Componentes de Aplicação	Princípios de Organização e Composição	Geradores baseados em Linguagens	Geradores de Aplicações	Sistemas Transformacionais
Exemplos	Bibliotecas de Subrotinas	Pipes POO	VHLLs POLs	STAGE	CIP

Tabela 4.1: Arcabouço para Tecnologias de Reusabilidade

4.1.9 Transformações e Reusabilidade

Os sistemas transformacionais classificam-se, em termos de reusabilidade, como esquemas generativos, com componentes de natureza ativa, difusa e maleável. Os componentes reutilizáveis são **ativos**, i.e., geram o programa objeto. O termo **difuso** refere-se ao fato de que, neste grupo, a identificação dos componentes reutilizáveis é difícil, pois os efeitos deles no programa objeto são, em geral, globais e difusos. A tabela 4.1 [Bigg84] apresenta essa classificação.

Dentro do grupo de Padrões, podemos distinguir três classes de sistemas, que representam esforços correlatos de pesquisa na área:

1. **Sistemas Baseados em Linguagens**
   São sistemas que, apesar de possuirem um mecanismo de geração, dão maior ênfase à linguagem que será usada para descrever o sistema objeto. As linguagens podem ser de propósito geral (chamadas de VHLLs[5]) ou orientadas para problemas (POLs[6]). Como exemplo, citamos a linguagem MODEL [Pryw77].

2. **Geradores de Aplicações**
   Os componentes reutilizáveis podem ser padrões de código existentes no próprio gerador. A ênfase é dada ao processo de geração de programas de aplicação em domínios restritos. Como exemplo citamos STAGE [Clea88].

3. **Sistemas Transformacionais**
   Os componentes reutilizáveis são padrões encontrados nos conjuntos de regras de transformação. A ênfase é dada no processo de transformação automática de uma especificação formal em um programa eficiente, sob controle do usuário. Como exemplo citamos CIP [Baue85], [Baue87].

[5] Very High Level Languages
[6] Problem Oriented Languages

Em geral, há partes do software que queremos reutilizar, mas que não se encontram na forma correta desejada. Uma forma de fazer isto seria aplicando transformações de programas. Em [Gogu84], um esquema genérico de transformação estruturada de programas é proposto para produção de módulos através da combinação e modificação de outros módulos existentes.

A idéia básica desta abordagem consiste na maximização da reutilização de programas, partindo da premissa de que se deve armazenar programas na sua forma mais genérica. A construção de um novo módulo a partir de um existente é feita através da instanciação de um ou mais parâmetros do módulo. Os módulos podem ser modificados, antes ou depois da instanciação, de modo a se adequarem a uma maior variedade de aplicações.

As modificações possíveis são:

1. **enriquecer**[7] um módulo, incrementando a sua funcionalidade;

2. **renomear** alguma parte da interface de um módulo;

3. **restringir** um módulo, eliminando parte da sua funcionalidade.

Alguns dos sistemas transformacionais existentes preconizam a reutilização de informações de análise e de projeto ao invés da tradicional reutilização de código. Um histórico da derivação do programa é mantido, contendo as decisões tomadas durante o processo.

4.1.10 Áreas Correlatas

É uma tarefa difícil caracterizar algum sistema como sendo "transformacional", já que essa área possui muitas intersecções com outras áreas afins, tais como: sistemas genéricos de IA, sistemas de produção, sistemas especialistas, geradores de aplicações, sistemas de verificação e ambientes de desenvolvimento de software.

Prova Automática de Teoremas

Fazer uma prova (demonstração) para um teorema em matemática requer, além da habilidade de fazer deduções a partir de hipóteses, certo grau de intuição para a escolha da sequência de lemas que precisam ser provados para auxiliar a prova completa do teorema.

Um sistema para prova automática de teoremas possui, geralmente, de forma limitada, o mesmo tipo de habilidade que um matemático utiliza para a seleção de lemas úteis para a prova desejada e para decompor o problema em subproblemas mais fáceis de serem resolvidos.

Verificação Automática de Programas

A verificação de programas corresponde à atividade de provar que programas estão corretos em relação à sua especificação, independente da sua entrada.

[7] enrich

Programação Automática

Um sistema para Programação Automática pode ser considerado como um "supercompilador". Trata-se de um programa que recebe como entrada uma descrição de nível muito alto do problema que um programa deve resolver e produz automaticamente tal programa.

A tarefa de escrever um programa de forma automática para resolver um problema especificado está fortemente relacionada à verificação de programas. Adicionalmente, sistemas de Programação Automática também produzem a verificação do programa gerado.

Sistemas de Produção

Sistemas de Produção[8] são uma designação genérica para sistemas na área de IA que implementam um modelo computacional que prescreve uma separação nítida entre dados, operações e controle, entidades computacionais nem sempre claramente identificáveis em sistemas de computação convencionais.

Os dados são mantidos em um repositório de dados global e modificados através de operações, que, por sua vez, são aplicadas de acordo com alguma estratégia de controle global.

4.2 O Sistema de Transformação do tipo Fold/Unfold

O sistema de transformação do tipo *Fold/Unfold* é fruto de um trabalho realizado por [Burs77] e é um exemplo típico da abordagem de conjunto gerador de regras.

4.2.1 Tipos de Regras de Transformação

O sistema contém apenas seis tipos de regras de transformação de programas:

- Definição

  Esta transformação permite que uma equação E = E' seja adicionada ao programa se E não for uma instância do lado esquerdo de nenhuma equação existente.

- *Unfolding* (Expansão)

  Seja E = E' a definição a ser expandida, onde a expressão E é definida como E', e seja F = F' a equação a ser transformada. Para que isto seja possível, uma instância de E, por exemplo E_1, deve ocorrer em F'. Seja E_1' a instância correspondente a E'. Para expandir a definição E = E' na equação F = F', substitui-se a ocorrência de E em F', no caso E_1, por E_1'. Chame o resultado dessa substituição de F" e pode-se adicionar a equação F = F" ao programa.

- *Folding* (Contração)

  Seja E = E' a definição a ser contraída, onde a expressão E é definida como E', e seja G = G' a equação a ser transformada. Se uma instância de E, por exemplo E_1', ocorrer em G', isto será possível. Seja E_1 a instância correspondente a E. Para contrair a definição E =

[8] Production Systems

E' na equação G = G', substitui-se a ocorrência de E' em G', no caso E_1', por E_1. Chame essa substituição de G" e pode-se adicionar a equação G = G" ao programa.

- Instanciação

  Seja E = E' uma equação do programa. Esta transformação permite que uma instância de substituição de E = E' seja adicionada ao programa. É também conhecida por "substituição".

- Abstração

- Laws

  Seja E = E' uma equação do programa. Esta transformação permite que E = E" seja adicionada ao sistema de equações, onde E" é obtida a partir de E', através do uso de qualquer uma das propriedades dos operadores primitivos da linguagem. Por exemplo, pode-se usar do fato de que a adição é comutativa e associativa para transformar E' em E"

O funcionamento deste sistema gira em torno da aplicação das regras de *Fold/Unfold*. Novas funções são adicionadas através da regra de Definição, as quais podem ser estruturadas através da regra de Abstração. O sistema permite a síntese de novas funções e a otimização de funções definidas pelo programador, através da aplicação de *unfolding*, seguida pela manipulação do programa através da regra de Instanciação (substituição) e *Laws*, e posterior *folding*.

4.2.2 Um Exemplo

O exemplo abaixo foi retirado de [Dill88]. Sejam as definições para as funções *length* e *append*:

$$length\ [\,] = 0, \tag{4.1}$$

$$length\ (a : x) = 1 + length\ x, \tag{4.2}$$

$$append\ [\,]\ y = y, \tag{4.3}$$

$$append\ (a : x)\ y = a : append\ x\ y. \tag{4.4}$$

Vamos definir uma função para calcular o comprimento total de duas listas l_1 e l_2. Uma primeira tentativa de resolver o problema poderia ser a seguinte:

$$lot\ l_1\ l_2 = length\ (append\ l_1\ l_2). \tag{4.5}$$

Esta solução não é muito eficiente, mas podemos melhorá-la, usando o método de transformação de programas.

Substituindo [ ] por l_1 na equação (4.5), obtemos a equação:

$$lot\ [\,]\ l_2 = length\ (append\ [\,]\ l_2). \tag{4.6}$$

Isto é um exemplo de aplicação de uma transformação do tipo **instanciação**. Agora, utilizando uma transformação do tipo **unfold** e considerando a equação (4.3) como E = E' e a equação (4.6) como F = F', temos:

$$\overbrace{append\ [\,]\ y}^{E} = \overbrace{y}^{E'},$$

$$\underbrace{lot\ [\,]\ l_2}_{F} = \underbrace{length\ \overbrace{(append\ [\,]\ l_2)}^{E_1}}_{F'}.$$

Isto resulta na equação:

$$\underbrace{lot\ [\,]\ l_2}_{F} = \underbrace{length\ \overbrace{l_2}^{E_1'}}_{F''}. \tag{4.7}$$

Em seguida, aplicando novamente **instanciação** através da substituição de $(a : x)$ por l_1 na equação (4.5), obtemos:

$$lot\ (a : x)\ l_2 = length\ (append\ (a : x)\ l_2). \tag{4.8}$$

Usando **unfold** novamente, com a equação (4.4) sendo E = E' e a equação (4.8) sendo F = F', temos:

$$\overbrace{append\ (a : x)\ y}^{E} = \overbrace{a : (append\ x\ y)}^{E'},$$

$$\underbrace{lot\ (a : x)\ l_2}_{F} = \underbrace{length\ \overbrace{(append\ (a : x)\ l_2)}^{E_1}}_{F'}.$$

Esta transformação nos leva a:

$$\underbrace{lot\ (a : x)\ l_2}_{F} = \underbrace{length\ \overbrace{(a : (append\ x\ l_2))}^{E_1'}}_{F''}. \tag{4.9}$$

Usando **unfold** na definição (4.2), temos:

$$lot\ (a : x)\ l_2 = 1 + length\ (append\ x\ l_2). \tag{4.10}$$

Agora, aplicando uma transformação do tipo **fold** com a definição (4.5) sendo E = E' e (4.10) sendo G = G', temos:

$$\overbrace{lot\ l_1 l_2}^{E} = \overbrace{length\ append\ l_1\ l_2}^{E'},$$

$$\underbrace{lot\ (a : x)\ l_2}_{G} = \underbrace{1 + \overbrace{length\ (append\ x\ l_2)}^{E_1'}}_{G'}.$$

Esta transformação nos leva à equação:

$$\underbrace{lot\ (a : x)\ l_2}_{G} = \underbrace{1 + \overbrace{lot\ x\ l_2}^{E_1}}_{G''} \tag{4.11}$$

Assim, chegamos a uma versão mais eficiente de *lot*:

$$\begin{aligned} lot\ [\,]\ y &= length\ y, \\ lot\ (a : x)\ y &= 1 + lot\ x\ y. \end{aligned}$$

4.3 O Projeto CIP

O Projeto CIP (Computer-Aided Intuition-Guided Programming) foi iniciado em 1975 na Technical University of Munich por um grupo de pesquisa coordenado por Bauer e Samelson. Este projeto se concentra em duas partes principais: o projeto de uma linguagem de espectro largo, a CIP-L [Baue85] e o desenvolvimento de um sistema para transformação de programas [Baue87].

As características principais desta abordagem são:

- O projeto tem como foco básico transformações de programa *source-to-source* que preservam a correção, em todos os níveis de formulação – do aplicativo ao procedural.
- A linguagem de manipulação é descrita na forma de uma hierarquia de tipos abstratos de dados (algébricos).
- Para manter coerência conceitual, o sistema de transformação CIP-S também está fundamentado em uma visão algébrica da linguagem de formulação e está especificado algebricamente.
- Além de realizar a **aplicação** de regras de transformação, o sistema também provê auxílio à **expansão** e **composição** de regras.
- O sistema não dispõe de meios automáticos para a seleção automática de transformações.

4.3.1 Tipos de Regras de Transformação

As operações básicas do sistema permitem a derivação de regras mais complexas a partir de regras elementares do sistema. Segundo a definição da linguagem, o conjunto de regras elementares é composto por:

- **regras fundamentais** – tipo FOLD/UNFOLD – do núcleo da linguagem que são submetidas à verificação semântica;
- **regras de definição** para construções adicionais da linguagem;
- **os axiomas dos tipos algébricos de dados** – que caracterizam certos domínios – que podem ser definidos em programas.

4.4 DRACO

DRACO é uma abordagem para construção de sistemas de software a partir de componentes reutilizáveis [Neig80], [Neig84a], [Neig84b], [Neig87]. Um "componente" é, em essência, um tipo abstrato de dados que descreve a semântica de um objeto ou operação de um domínio. Um **domínio** para DRACO corresponde ao encapsulamento de uma área de aplicação, através da identificação de objetos e operações comuns a vários sistemas nessa área específica.

DRACO implementa duas funções principais:

- definição e implementação de linguagens de alto nível orientadas para problemas – **linguagens de domínio**;
- assistência e automação parcial da tarefa de transformar a especificação de um sistema de software descrita através de uma linguagem de domínio para outra mais concreta.

Em termos gerais, os aspectos mais importantes desta abordagem são:

- **Reutilização de análise e de projeto.**

  É possível capturar e reutilizar o conhecimento sobre uma área de aplicação através do uso de linguagens de domínio; o conhecimento sobre o projeto pode ser capturado pelos componentes que proverão a semântica dessas linguagens.

- **Definição de linguagens de domínio.**

  Uma linguagem de domínio será o resultado da análise de uma área de aplicação (ou domínio), modelando-a através de objetos e operações que encapsulam o conhecimento adquirido.

- **Hierarquia de Domínios**

  A semântica dos componentes de um domínio é descrita através da tradução desses componentes para componentes de outros domínios, estabelecendo assim uma hierarquia de domínios, representada por um grafo dirigido no qual os nós são domínios, como mostra a figura 4.3.

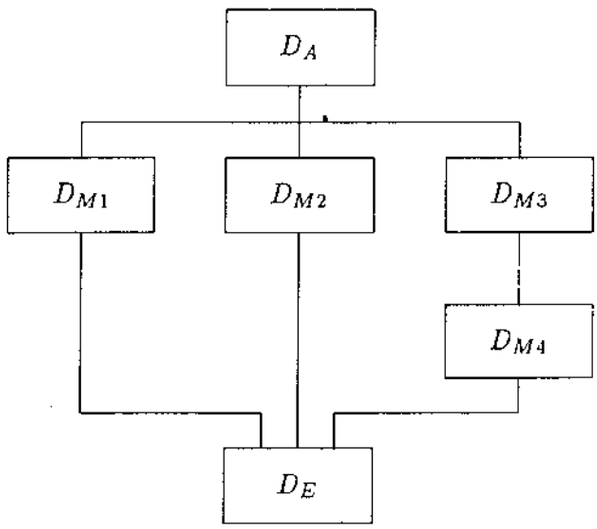
  Em geral, os domínios que se localizam no nível mais baixo da hierarquia implementam linguagens de programação e são chamados de **domínios executáveis**. Os domínios no nível mais alto da hierarquia são chamados de **domínios de aplicação** e aqueles em um nível intermediário são tratados por **domínios de modelagem** – e.g., o domínio de base de dados.

4.4.1 Tipos de Regras de Transformação

DRACO classifica as regras de transformação usadas em dois tipos básicos:

- Transformações

  Transformações *source-to-source* otimizam e especializam os componentes de software para uso em um sistema específico. Este termo, segundo a terminologia adotada por DRACO, pressupõe uma transformação intra-domínio, como mostra a figura 4.4.



D_A - Domínio de Aplicação
D_M - Domínio de Modelagem
D_E - Domínio de Execução

Figura 4.3: Hierarquia de Domínios

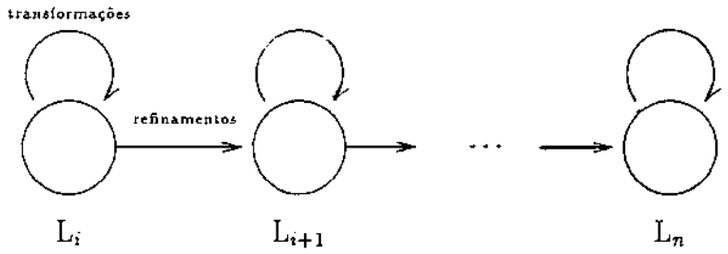


Figura 4.4: Transformações e Refinamentos

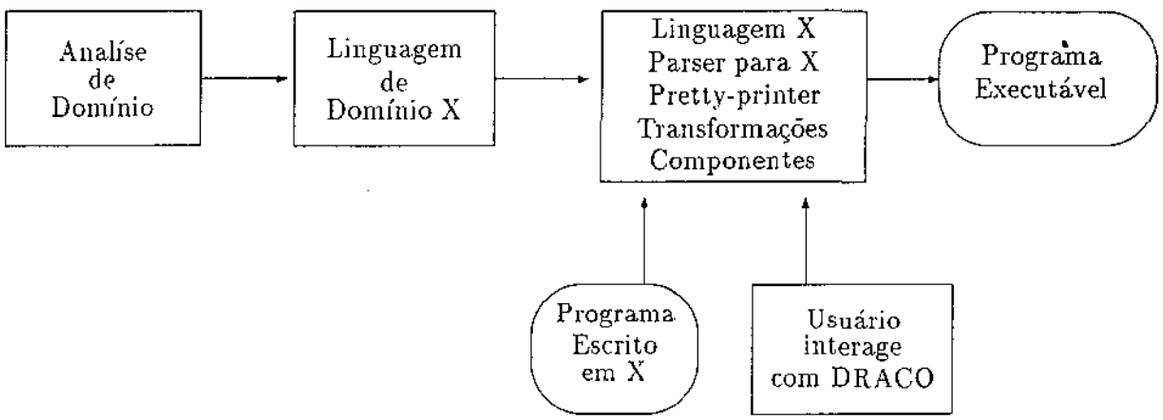


Figura 4.5: Esquema de Funcionamento de DRACO

- **Refinamentos**

  Refinamentos traduzem sentenças de uma linguagem de domínio para sentenças em outra linguagem, o que pressupõe transformações entre domínios distintos.

4.4.2 Funcionalidade

O esquema de funcionamento de DRACO é mostrado na figura 4.5. Nele, podemos distinguir três atividades principais:

1. **Análise de Domínio.**

   Consiste na compreensão do domínio de aplicação, identificando objetos, operações e restrições nas interações entre eles, definindo, então, uma linguagem de domínio. DRACO não dá suporte a esta atividade.

2. **Descrição de Domínio.**

   A descrição completa de um domínio consiste em 4 partes:

   - **Parser**
     A descrição do parser define a interface entre o domínio e o mecanismo, e consiste em duas partes:
     - A *forma externa* (sintaxe) e a *forma interna* do domínio são descritas em uma notação BNF convencional, estendida com mecanismos de controle para recuperação de erros e retrocesso. A forma interna é uma árvore cujos nós possuem um prefixo de identificação além de dados.
     - Descrição de mecanismos para verificação semântica sobre a combinação de objetos e operações do domínio.
   - **Pretty-printer**
     A descrição do pretty-printer informa a DRACO como produzir a forma externa do domínio a partir de todos os tipos de nós da forma interna.
   - **Transformações**
     Transformações são relações de simplificação (otimizações) para os objetos e operações do domínio. Atuam apenas sobre o domínio para o qual foram definidas.
   - **Componentes**
     Componentes de software especificam a semântica do domínio. Há um componente para cada objeto e operação do domínio. Estes componentes representam decisões de implementação. Cada componente consiste em um ou mais refinamentos (que provêem diferentes implementações). Cada refinamento mapeia a semântica do objeto ou operação em termos de uma ou mais linguagens de domínio conhecidas por DRACO.

3. **Desenvolvimento de uma Aplicação.**

   Quando um novo programa em um domínio já conhecido por DRACO deve ser desenvolvido, ele é descrito através da linguagem de domínio. Em seguida, ele é traduzido para a forma interna, a qual é submetida a transformações e refinamentos, interativamente, até se conseguir expressá-lo em uma linguagem executável. O grau de interação com o usuário pode ser diminuído através da definição de *táticas de refinamento*. Durante este processo, é criado um *histórico de refinamentos*, que registra as decisões de implementação. Isto possibilita a reutilização de projeto.

4.5 PSI

PSI é um sistema transformacional de propósito geral implementado em LISP, projetado para sintetizar programas eficientes. A entrada para o sistema é uma especificação obtida a partir de um diálogo com o usuário. PSI é considerado um conjunto de especialistas em diversas tarefas. Dentre os especialistas, podemos destacar:

- Grupo de Aquisição

É o responsável pela obtenção de especificações mediante interação com o usuário, consistindo em: um interpretador para o inglês, um especialista de *trace*, um especialista de discurso e um especialista no domínio da aplicação. Todos eles extraem informação a partir do diálogo e a convertem em fragmentos de programa que servem de entrada para outro especialista de aquisição, o construtor de modelos de programa. Este, por sua vez, produz um modelo de programa completo e o utiliza como interface entre o grupo de aquisição e o grupo de síntese.

- Construtor de Modelos de Programa

  A função principal do PMB (Program Model Builder) é a construção de um modelo de programa completo e consistente – um programa abstrato, independente de aspectos relativos a implementação, "anotado", em uma linguagem de alto nível – que corresponda às necessidades expressas pelo usuário. A entrada para o PMB é composta por fragmentos de programa, possivelmente incompletos, ambíguos, inconsistentes e ordenados arbitrariamente. A construção do modelo completo é um processo incremental de extração de informações e atualização de um modelo parcial intermediário. Enquanto atualiza, o PMB faz verificações de consistência para garantir que o modelo é legal (segundo a semântica da linguagem) e correto (de acordo com as intenções do usuário). Se o PMB não consegue resolver alguma questão, ele pede auxílio a outros especialistas ou ao usuário, em último caso.

  O conhecimento do PMB é implementado em INTERLISP, através de cerca de 200 regras procedurais, escalonadas por um interpretador de regras.

- Grupo de Síntese

  Consiste em um especialista em codificação e de outro em eficiência.

  - Especialista em Codificação

    Este especialista, conhecido como PECOS, recebe uma descrição abstrata de programa produzida pelo PMB e a refina sucessivamente, através da aplicação de regras de transformação que expressam conhecimento sobre codificação. O especialista consiste em duas partes:

    * uma coleção de cerca de 400 regras de transformação sobre programação simbólica, organizadas em uma base de conhecimento extensível;
    * uma estrutura de controle orientada por tarefa, baseada na abordagem *step-by-step*.

    O catálogo extensível contém regras de três tipos: regras de refinamento (conhecimento sobre codificação em programação simbólica), regras de propriedade (especificação de propriedades adicionais de uma descrição de programa) e regras de consulta (respostas a consultas dos especialistas a respeito de uma descrição).

    O desenvolvimento corresponde a refinamentos sucessivos da descrição do programa até que se chegue a um programa em LISP.

4.5.1 Tipos de Regras de Transformação

- regras procedurais
- regras de refinamento (conhecimento sobre codificação em linguagem simbólica)
- regras de propriedade (especificam propriedades adicionais de uma descrição de programa)
- regras de consulta (para responder a consultas de especialistas sobre uma descrição)
- regras de planejamento (derivadas de análise prévia de como realizar decisões de implementação específicas)
- regras para agrupar decisões afins
- regras sobre escalonamento
- regras de estimativa de custos

4.6 Considerações Finais

Os sistemas transformacionais existentes e conhecidos foram desenvolvidos em instituições de pesquisa. São ferramentas experimentais e limitadas. Tal limitação prática associada às dificuldades enfrentadas em função do formalismo subjacente a muitos deles é motivo de pesada crítica por parte de alguns pesquisadores.

Na verdade, a busca por software **confiável** e **verificado** motiva toda a linha de experimentação com desenvolvimento formal. À medida em que sistemas automatizados vão sendo introduzidos em áreas onde o fator correção é um ponto crítico (medicina, controle de processos, aeroportos, etc.), em tarefas cada vez mais complexas e sob a operação de usuários não-especializados, a confiabilidade requerida é muito alta.

Além disso, há a constatação de que muito do que vem sendo feito manual e repetidamente pode sofrer uma drástica redução nos esforços dispendidos para realizá-los se puderem ser feitos (e re-feitos com pequenas modificações) automatica ou semi-automaticamente.

Em especial, alguns projetistas e programadores subestimam a complexidade do problema com o qual estão lidando e sobrecarregam sua capacidade mental, quando, em parte, muito poderia estar sendo feito pela máquina[9], deixando os programadores e o projetistas com mais tempo livre para a realização de tarefas criativas.

Há vários grupos de pesquisa na área, porém não há consenso a respeito de algumas questões fundamentais; o sucesso do desenvolvimento transformacional de software depende de muitos fatores, entre eles:

- maior conhecimento a respeito da natureza do processo de software;

[9]Nos primórdios da Computação, talvez os primeiros exemplos de reutilização de software tenham sido as cópias manuais de rotinas básicas de E/S, ainda escritas em linguagem de máquina, de um programa para outro, cuidando para que os endereços de memória fossem devidamente alterados. O interessante é que todo este trabalho era realizado manualmente para a operação de uma máquina que sabe copiar e somar rapida e precisamente, como nenhum ser humano é capaz de fazer!

- maior conhecimento a respeito da análise e especificação de domínios de aplicação;
- o estabelecimento de um arcabouço conceitual para as atividades realizadas no processo de software e de uma base teórica firme;
- o estabelecimento de uma arquitetura de referência que permita a comparação entre sistemas e a identificação de semelhanças e diferenças dissociadas de terminologia e outros aspectos irrelevantes em um contexto mais amplo de discussão;
- o estabelecimento de uma taxonomia básica para transformações;
- um estudo mais aprofundado sobre mecanismos de seleção e coordenação de transformações;
- maior disseminação de técnicas formais ou, pelo menos, rigorosas, para o projeto de software;
- apoio automatizado.

Capítulo 5

Ambientes de Apoio ao Desenvolvimento Transformacional

5.1 Introdução

O objetivo deste capítulo é o de propor uma arquitetura de referência para ambientes de apoio a desenvolvimento transformacional de software, com base em tendências de novas aplicações e em algumas premissas extraídas de revisões e discussões realizadas nos capítulos anteriores, com destaque para a natureza do processo de software, características de métodos de desenvolvimento, características essenciais de projeto de software e o papel do conceito de transformação em desenvolvimento de software.

Finalmente, apresentamos uma instância da arquitetura de referência proposta, com destaque para um agente especial, o AADT (sigla para Ambiente de Apoio ao Desenvolvimento Transformacional), que corresponde ao experimento concreto deste trabalho.

5.2 Por que "Transformacional"?

Dentre as abordagens existentes para a obtenção de software verificado formalmente, a programação transformacional é, certamente, a mais avançada e a mais flexível, apesar de nem sempre manipular formalmente artefatos e fases de desenvolvimento – o sistema DRACO é um exemplo disto.

Mesmo se considerarmos que o desenvolvimento de software verificado formalmente não corresponde ao enfoque prático adotado neste trabalho, a opção por transformações reflete a crença de que esta tendência inquestionavelmente se concretizará nos próximos anos e que, no futuro, sistemas transformacionais se tornarão partes integradas a ambientes de desenvolvimento de software.

5.3 Requisitos para Ambientes de Apoio ao Desenvolvimento Transformacional

Nos capítulos anteriores, estabelecemos um contexto para a discussão que virá a seguir: quais são as características desejáveis que um ambiente de apoio ao desenvolvimento transformacional de software deveria possuir?

- **arcabouço conceitual unificador**

  Um ambiente de apoio ao desenvolvimento de software deve estar fundamentado em um único arcabouço conceitual, que permita uma maior integração entre as diversas atividades realizadas como também entre as ferramentas de apoio.

  Consideramos que a visão do processo de software como um processo de transformação é o primeiro passo para o estabelecimento de um arcabouço conceitual unificador. A idéia de passo canônico e as teorias matemáticas associadas encontradas em [Turs87] determinam uma estrutura e forma universais para o processo de reificação e um ponto de partida para a definição de um processo coerente para o processo representado pela perna esquerda do modelo PW de Lehman.

- **apoio ao trabalho cooperativo**

  Desenvolver e manter software são atividades complexas que requerem cooperação.

  O processo de software é, em geral, realizado por um grupo de pessoas com funções específicas, responsáveis por tarefas tais como análise, projeto e programação. Em geral, sistemas são decompostos em subsistemas, que por sua vez, são designados a diversos grupos de trabalho (equipes). Obviamente, é necessário prover meios para a comunicação entre as pessoas de uma mesma equipe e entre as de equipes distintas, bem como para a coordenação e a integração entre os vários subsistemas, tarefas associadas e pessoas responsáveis.

- **compromisso com a natureza das novas aplicações**

  As aplicações de software emergentes, além de complexas, de grande porte e diversificadas requerem:

  - execução sobre uma plataforma heterogênea, composta por diversos tipos de equipamento interligados através de uma rede e,
  - facilidade de modificação, extensão e reconfiguração para atender à evolução natural dos seus requisitos e do ambiente de execução.

- **apoio à resolução distribuída de problemas**

  Considerando novamente as características das aplicações emergentes, é provável que os agentes que venham a compor um ambiente de apoio ao desenvolvimento estejam distribuídos pelos nós de uma rede.

  Assim, é desejável a presença de algum mecanismo que apóie o desenvolvimento e a execução distribuída de aplicações.

- **apoio à documentação do processo de desenvolvimento**

  Um ambiente de apoio ao desenvolvimento transformacional de software deveria prover algum mecanismo para apoiar o registro do processo de desenvolvimento em paralelo à execução de suas atividades. Este registro deveria ser facil e imprescindivelmente atualizado sempre que se fizesse necessário. Seria ideal que, além de manter as diversas versões de artefatos gerados durante o processo de desenvolvimento, também fossem registradas as decisões de projeto e as operações que conduziram a tais versões.

- **independência em relação a métodos e modelos de resolução**

  Um ambiente de desenvolvimento não deveria estar restrito a um método de desenvolvimento específico nem a modelos de resolução de problemas. Como argumentamos anteriormente, em desenvolvimento de software os fatos não ocorrem de modo estritamente *top-down* ou *bottom-up*, e distintas aplicações podem requerer distintos mecanismos de raciocínio. Da mesma forma, um método (ou facetas de um método) pode ser mais ou menos adequado para diferentes tipos de aplicação (e.g., processamento de dados convencional, aplicações em tempo real). Entretanto, algum mecanismo de configuração deveria existir que permitisse a instanciação de ambientes específicos.

- **interface amigável e padronizada**

  O ambiente deveria prover um conjunto de interfaces uniformes e amigáveis, porém personalizáveis – sem que isso venha a desrespeitar convenções básicas estabelecidas para a interação com o usuário. É desejável que a interface dê suporte à interação via multimeios (o estado-da-arte prescreve suporte gráfico, através de janelas e ícones) e disponha de *help on-line*.

- **papel ativo**

  Quanto maior o papel ativo de um ambiente, maior será o número de funções realizadas por ele sem a necessidade de interação intensa com o usuário. Um ambiente deveria ser útil, antecipando os próximos passos do usuário e tomando a iniciativa de executar algumas funções em situações já conhecidas.

- **mecanismos de coordenação de atividades**

  A complexidade intrínseca às aplicações requer a decomposição de tarefas em subtarefas e artefatos em partes mais tratáveis. É necessário definir (ao menos) um mecanismo global para a coordenação de atividades (tarefas) no ambiente.

- **suporte a um núcleo básico de transformações**

  O suporte a um núcleo básico de transformações, proporciona ao ambiente um tipo de arcabouço conceitual abrangente que permite a discussão de modelos transformacionais. Tal núcleo básico de transformações corresponde a um conjunto de operações básicas, no qual cada operação básica é mapeada a uma ou mais operações de métodos específicos.

5.4 Uma Arquitetura de Referência

A figura 5.1 apresenta uma arquitetura de referência para ambientes de apoio ao desenvolvimento de software, inspirada em [Taka90] e nos requisitos apresentados na seção anterior.

A arquitetura global é visualizada como um *framework* de *blackboard*, composto por mecanismos de comunicação, mecanismos de coordenação e controle, e agentes:

- Comunicação

  Há um *bus* de software principal, um mecanismo global para articulação e comunicação entre agentes. Este *bus* pode ser uma rede local onde cada agente executa em um processador distinto ou, simplesmente, uma estação de trabalho com processos que se comunicam através de algum mecanismo conhecido (*pipes*, *streams*, etc.)

- Coordenação

  Há um ou mais agentes especiais, responsáveis pela coordenação de outros agentes que apóiam o desenvolvimento cooperativo de atividades.

- Agentes

  Os agentes são os componentes da arquitetura global e correspondem a ferramentas ou processos utilizados para o apoio ao desenvolvimento de aplicações.

  Cada agente, por sua vez, pode ser considerado como um sistema de *blackboard* com o seu *bus* local, seus agentes e mecanismos de coordenação.

5.5 Uma Instância Imediata

A figura 5.2 apresenta uma instância da arquitetura de referência proposta.

Nessa instância, um agente especial, sob a designação de AADT (sigla para Ambiente de Apoio ao Desenvolvimento Transformacional), corresponde ao experimento concreto deste trabalho e será descrito no próximo capítulo.

5.5.1 UNIX

Idealmente, qualquer agente que faça parte de uma arquitetura de referência deveria poder executar em qualquer tipo de processador e sob qualquer sistema operacional. Uma forma de satisfazer esse requisito seria através da personalização do ambiente para hardware/software específicos. Entretanto tal requisito pode ser alcançado sem tamanha restrição, através da escolha de um sistema operacional que seja padrão em diversas máquinas.

O sistema operacional UNIX foi projetado com o objetivo de dar suporte natural e eficiente à atividade de construção de programas, atuando também como ambiente de programação. Em especial, os arquivos têm formato uniforme, permitindo a compatibilidade entre programas que produzem e consomem dados, e há tratamento homogêneo entre arquivos e mecanismos de E/S.

O UNIX, devido às suas inúmeras qualidades, que não fazem parte do escopo deste trabalho, tornou-se um padrão, sendo utilizado por diversos fabricantes, tanto em estações de trabalho como em *mainframes*. Além disso, o sistema operacional UNIX oferece vários serviços de comunicação entre processos através de uma rede.

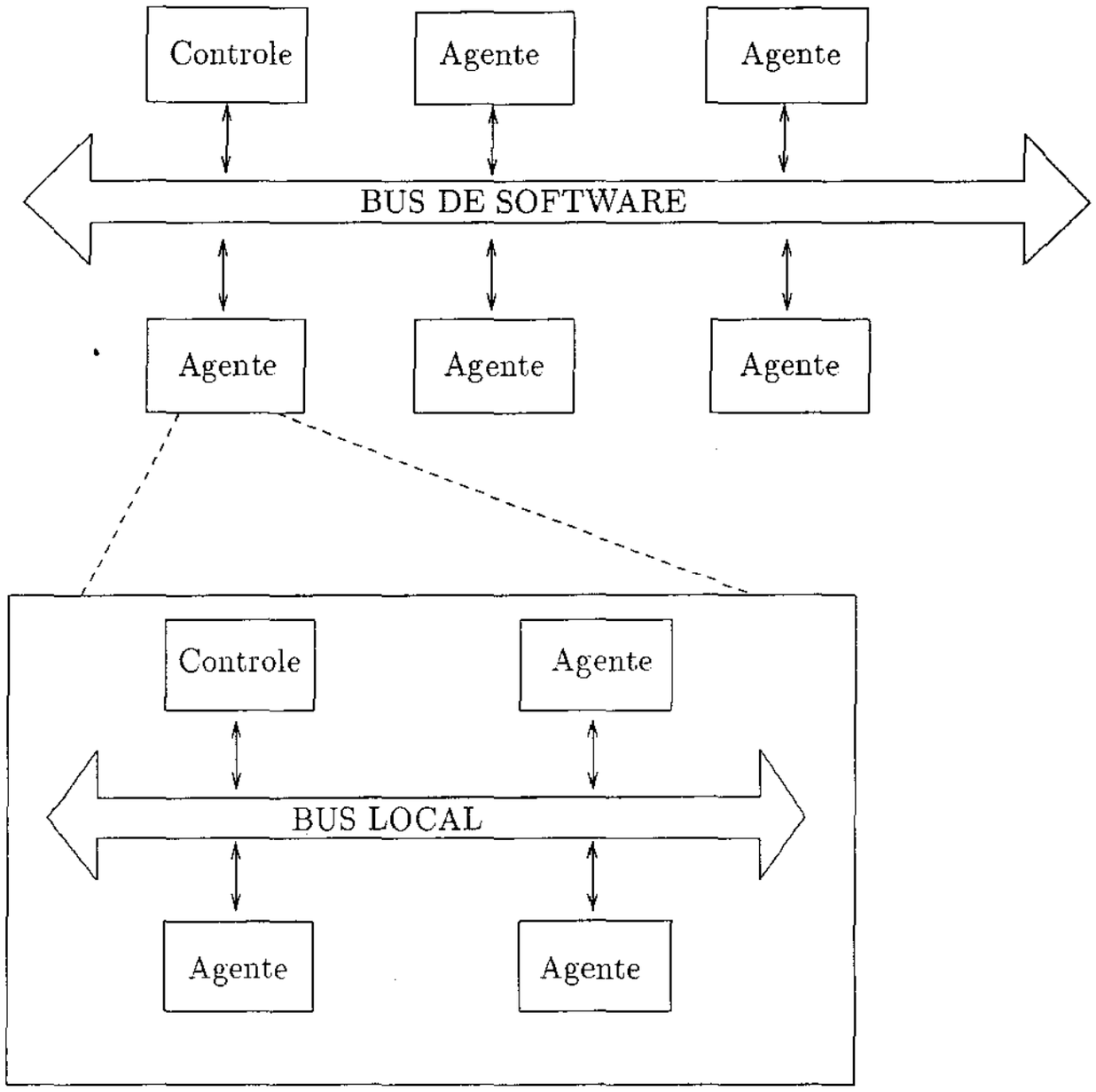


Figura 5.1: Arquitetura Geral de Referência

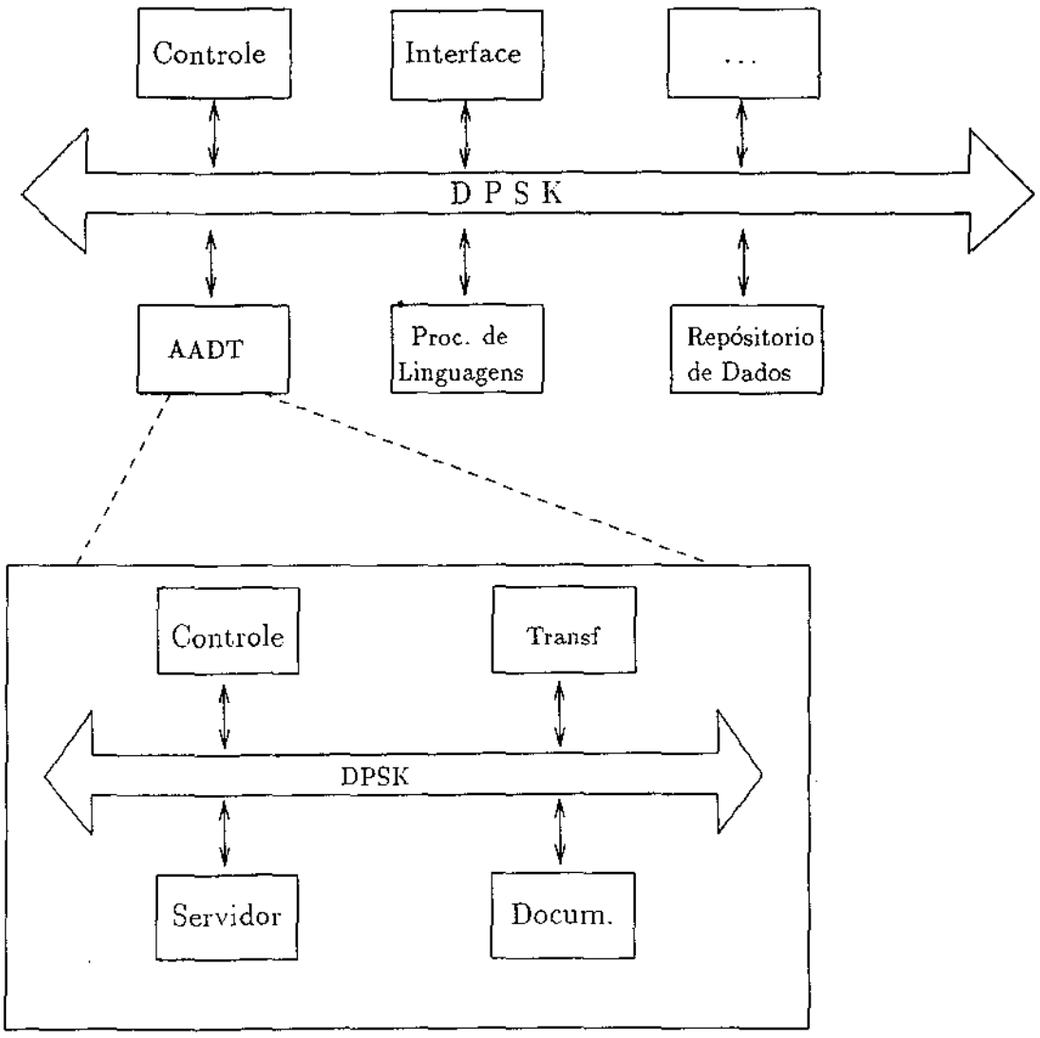


Figura 5.2: Uma Instância Imediata

5.5.2 DPSK

O DPSK [Card87] – Distributed Problem Solving Kernel – é um núcleo cujo objetivo é o de dar suporte ao desenvolvimento de software para a resolução de problemas complexos – problemas cuja solução requer a interação entre diversos agentes de resolução com conhecimento em áreas específicas – provendo um arcabouço para computação distribuída no qual ambientes para a resolução distribuída de problemas[1] possam ser implementados.

O DPSK implementa as atividades básicas encontradas em resolução distribuída de problemas – comunicação, controle de processos, sincronização e tratamento de exceções – e propõe um modelo para comunicação e controle nos ambientes de apoio a essas atividades, permitindo o desenvolvimento de aplicações com controle centralizado ou descentralizado e com estrutura de comunicação síncrona ou assíncrona.

O DPSK implementa as atividades básicas citadas acima através de um pequeno número de primitivas, com interface para linguagens para processamento numérico ou simbólico, tais como C, C++, FORTRAN, OPS5 e LISP. Certas construções – por exemplo, *Blackboards*, *Schedulers* e Monitores – podem ser facilmente implementáveis com as primitivas básicas do núcleo.

Comunicação

- Tipos de Dados

  Os elementos de comunicação do DPSK são **objetos**, os tipos de dados mais abstratos que podem ser suportados por agentes de resolução de problemas numéricos e simbólicos. Um objeto é um conjunto de pares de <atributo,valor>[2] e pertence a uma classe.

- Mecanismos

  O DPSK realiza a comunicação entre agentes de resolução através de **memória compartilhada**. Este mecanismo, além de proporcionar comunicação assíncrona – requerida por diversas aplicações de resolução de problemas, em especial as que adotam o modelo de *blackboard* – também torna a comunicação entre agentes independente dos endereços (lógicos ou físicos) de cada agente.

  Com o intuito de favorecer a disponibilidade dos dados, a memória compartilhada é **descentralizada**. O acesso às memórias compartilhadas descentralizadas é controlado através de **transações**. Uma transação é uma operação definida sobre um segmento de uma base de dados (uma classe para o DPSK).

  Em resumo, os agentes de resolução comunicam-se através da escrita e leitura de objetos, por meio de transações, em uma memória compartilhada descentralizada.

- Gerente de Transações (GT)[3]

[1] Distributed Problem Solving Environments constituem uma classe especial de sistemas distribuídos, onde um conjunto de módulos (ou agentes de resolução de problemas) executam concorrentemente em uma rede de processadores. Em geral, os módulos apresentam alta complexidade (módulos realizam tarefas complexas), alta granularidade (módulos gastam mais tempo processando do que se comunicando) e alta heterogeneidade (módulos realizam tarefas variadas).

[2] tratados por *slots*

[3] Transaction Manager

Todas as rotinas do núcleo chamadas pelo usuário falam apenas com o GT. O GT tem um endereço permanente de comunicação em cada processador em que está executando em *background*. Suas atividades principais são:

1. Controlar o status da transação corrente;
2. Enviar para outros processadores pedidos de transação que envolvem classes armazenadas nas suas memórias;
3. Controlar a execução de seus módulos.

O GT é composto por vários módulos, responsáveis pelas funcões de comunicação e controle entre os agentes e pelo acesso à memória compartilhada.

- Primitivas de Comunicação

1. **Begin-Transaction(classe,modo)**: inicia o acesso aos objetos localizados na memória compartilhada designados por **classe**. Os modos possíveis são: leitura e escrita. Retorna um identificador de transação.
2. **End-Transaction(id)**: termina uma sessão de acesso à memória compartilhada.
3. **Abort-Transaction(id)**: interrompe uma transação designada por **id**.
4. **Op-Transaction(id,tipo,padrão)**: realiza o acesso à memória compartilhada. Os valores possíveis para **tipo** estão enumerados no item abaixo. O acesso é feito a todos os objetos que satisfaçam o **padrão** fornecido.

- Operações de Gerência de Dados

As primitivas básicas para gerência dos objetos são:

1. **MakeObject(objeto)**: cria um objeto.
2. **DeleteObject(padrão)**: elimina todos os objetos que casam com o padrão fornecido, onde **padrão** corresponde a um conjunto de pares de <atributo,valor> de uma classe.
3. **GetObject(padrão)**: retorna os objetos que casam com o padrão fornecido.
4. **AddSlot(padrão, slot)**: adiciona um slot aos objetos que casam com o padrão fornecido.
5. **DeleteSlot(padrão, atributo)**: elimina o slot especificado por um atributo de todos os objetos que casam com o padrão fornecido.
6. **UpdateSlot(padrão, atributo, valor)**: atualiza o valor de um slot especificado por atributo para todos os objetos que casam com o padrão fornecido.
7. **GetSlot(objeto, atributo)**: retorna o slot especificado por atributo no objeto fornecido.

Controle

É a habilidade de um agente (processo) de iniciar, suspender, continuar e abortar outro agente (processo) em qualquer nó (processador) da rede.

As atividades de controle são implementadas no DPSK também através dos gerentes de transação.

A primitiva básica de controle é:

- **Proc-Control(ação, processador, arquivo, pid)**: permite que um determinado agente possa controlar outro (representado por um identificador de processo contido em **pid**), localizado em qualquer **processador**. As ações de controle são:

  1. RUN
  2. SUSPEND
  3. RESUME
  4. KILL
  5. STATUS

Sincronização

A **sincronização** é a habilidade que um agente de resolução possui de, em algum ponto de sua execução, esperar que outro(s) agente(s) atinjam pontos específicos de sua(s) execução(ões). No DPSK, ela pode ser obtida através de definição de varáveis globais chamadas de **eventos**.

Um evento pode ser entendido como sendo uma variável que pode se encontrar em um de dois estados: ativo ou inativo. Dentre as suas propriedades, podemos destacar:

1. Qualquer agente pode ativar um evento.
2. Qualquer agente pode desativar um evento.
3. Um processo pode esperar que um evento se torne ativo.

As primitivas básicas de sincronização são:

1. **Affirm-Event(evento)**: insere **evento** na memória compartilhada.
2. **Negate-Event(evento)**: retira **evento** da memória compartilhada.
3. **Check-Event(evento)**: checa se **evento** já se encontra na memória compartilhada.
4. **Wait-Event(evento, timeout)**: aguarda que **evento** seja inserido na memória compartilhada até o limite estabelecido em **timeout**.

Tratamento de Exceções

O **tratamento de exceções** é um mecanismo através do qual agentes de resolução podem ser interrompidos e forçados a desviar as suas execuções para uma parte do código específica para o tratamento do problema identificado (exceção).

As primitivas básicas são:

1. **Set-Group(grupo)**: associa o agente a **grupo.**
2. **Set-Handler(handler)**: considera o procedimento especificado em **handler** como sendo a rotina de tratamento de exceções do agente.
3. **Sig-Group(signal, grupo)**: envia um interrupção, identificada por **signal**, ao grupo de agentes representado por **grupo.**

5.5.3 Repositório de Dados

É onde ficará toda a informação referente a aplicações desenvolvidas utilizando o ambiente, incluindo objetos e versões, documentação do processo de desenvolvimento, etc.

5.5.4 Processadores de Linguagens

Uma instância de ambiente deverá incluir linguagens e processadores de linguagens necessários ao desenvolvimento de aplicações. Os processadores tradicionais incluem, entre outros:

- editores
- compiladores
- link-editores
- depuradores

Capítulo 6

Um Experimento Concreto

6.1 Introdução

Este capítulo apresenta:

- uma descrição da implementação de um protótipo de ambiente de apoio a desenvolvimento transformacional e,
- alguns resultados obtidos através da experimentação com um exemplo simples.

O protótipo implementado não cobre inteiramente os requisitos apresentados no capítulo anterior, detendo-se apenas em alguns focos básicos. Além disso, assumimos que algumas atividades devem preceder o uso efetivo do protótipo; estas serão agrupadas em uma fase de configuração do ambiente, a qual, no entanto, não é objeto deste trabalho.

6.2 Objetivos

O objetivo do experimento é o de demonstrar parte da funcionalidade definida para ambientes de apoio ao desenvolvimento transformacional de software, de acordo com alguns requisitos estabelecidos no capítulo anterior, com ênfase em três focos básicos: generalidade, operações primitivas e coordenação.

O suporte a um conjunto de primitivas básicas de projeto – que se constitui em um núcleo de transformações – a partir do qual serão derivadas as regras de transformação do ambiente, permite a representação sucinta e hierarquizada das informações de projeto, estabelecendo um arcabouço conceitual unificador através das relações estruturais definidas entre objetos e domínios mediante a aplicação dessas operações primitivas e suas derivadas.

A coordenação de atividades é realizada em dois níveis:

1. A nível de relações entre objetos e domínios, mediante um mecanismo de encadeamento regressivo que será descrito na seção 6.6.
2. A nível de agentes do sistema, mediante a coordenação global do ambiente. Este, apesar de independente de modelos de resolução específicos pode ser devidamente personalizado, se necessário, através de um conjunto de regras de coordenação.

6.3 Cenário de Utilização do Ambiente

Há algumas atividades que precedem obrigatoriamente o uso **operacional** de muitos ambientes de apoio a desenvolvimento de software.

Alguns ambientes podem dar algum tipo de apoio automatizado a tarefas paralelas – por exemplo, documentação – mas não apóiam de forma automatizada a finalidade específica de cada atividade. Outros ambientes apóiam de algum modo algumas dessas atividades, sob a prescrição de uma metodologia, mas não sabem utilizar as representações obtidas para derivar automaticamente outras.

Sendo assim, o esforço dispendido na obtenção das várias representações nas diversas atividades não é minimizado, já que as representações iniciais, apesar de aumentarem a compreensão sobre a aplicação e de servirem como guia para as próximas atividades (que irão gerar novas representações) não são passíveis de transformações automáticas.

Argumentamos que um ambiente genérico de apoio ao desenvolvimento transformacional pode dar suporte automatizado a diversas atividades e de acordo com diversos métodos, desde que seja devidamente configurado para isso.

Abaixo, descrevemos algumas atividades que, obrigatoriamente, comporão a sua configuração e que precederão o uso operacional do ambiente proposto.

6.3.1 Configuração do Ambiente

Análise do Problema

Um problema (ou área de aplicação) é "dissecado", com o objetivo de identificar todos os objetos (dados) e operações (processos) encontrados em qualquer situação ou contexto. O resultado dessa análise prossegue através da categorização desses objetos e operações em **domínios** e do estabelecimento de relações entre eles, em especial, relações que definem uma **hierarquia de domínios** envolvidos no problema. Possivelmente alguns dos domínios identificados terão sido anteriormente definidos e poderão ser reutilizados, enquanto que outros deverão ser definidos.

Uma hierarquia de domínios é um grafo dirigido cujos nós são domínios e cujas arestas são relações entre domínios.

Análise de Domínio

O objetivo da análise de domínio é a identificação de objetos, operações e relações pertinentes a um domínio de aplicação específico. Os elementos identificados serão utilizados para a especificação do domínio em questão, utilizando uma linguagem de especificação.

Neste trabalho, definimos um **domínio** como sendo um conjunto de tipos ou categorias de objetos. Nada mais natural, então, do que representá-lo como uma classe que contém outras classes, como mostra a figura 6.1.

A figura 6.1 representa o domínio *DFD* como sendo composto das classes *DFD*, *Entidade*, *Processo*, *Depósito* e *Fluxo*. A relação entre a classe *DFD* e as outras é de **agregação** e significa que as variáveis de instância da classe *DFD* são instâncias ou conjuntos de instâncias das outras classes.

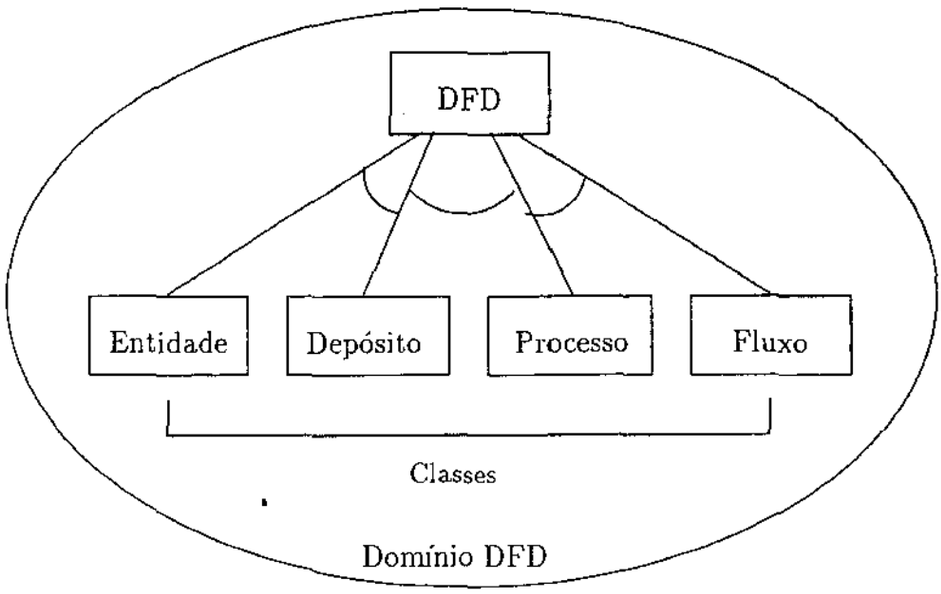


Figura 6.1: Estrutura do Domínio DFD

Um objeto de um domínio pode ser uma instância de qualquer uma de suas classes. Por exemplo, podemos instanciar objetos das classes *Entidade*, *Processo*, *Depósito* e *Fluxo* e também da classe *DFD*. Qualquer um deles será um objeto do domínio *DFD*.

Especificação de Domínios

A representação estrutural dos dados obtidos pela análise de domínio é feita através de uma linguagem de especificação. Neste trabalho, apesar de não estarmos diretamente preocupados com a representação de domínios, assumiremos que esta será feita através de uma linguagem de especificação que incorpora conceitos de modelagem de objetos, tais como: classes, variáveis de instância, métodos e herança.

A figura 6.2 mostra a representação da classe *DFD* através de uma notação descrita no apêndice B.

Definição das Transformações

Tendo estabelecido os domínios da aplicação e uma hierarquia entre eles, será necessário definir as transformações entre domínios. Cada domínio definido tem um conjunto de transformações associado. As transformações são definidas **intra-domínios**, associando objetos de um mesmo domínio ou **inter-domínios**, que associam objetos de domínios distintos, sempre segundo a hierarquia de domínios estabelecida anteriormente.

Além destas, podem ser definidas **transformações de coordenação** ou meta-transformações, que irão implementar as heurísticas de resolução de problemas de uma área de aplicação. O con-

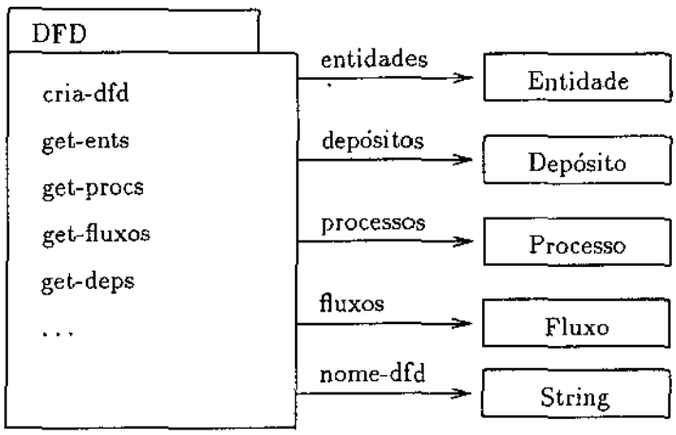


Figura 6.2: A classe DFD

junto inicial de transformações de coordenação pode ser modificado manualmente para refletir experiência ou constatações adquiridas através do uso do ambiente configurado.

Geração do Ambiente

A geração do ambiente configurado consiste na instanciação de um ambiente que suporte o desenvolvimento de aplicações que envolvam os domínios e a hierarquia definidos.

6.3.2 Operação

Após a configuração de um ambiente para apoio ao desenvolvimento de aplicações em uma área específica e, possivelmente segundo algum método de desenvolvimento, este poderá ser utilizado individualmente para o desenvolvimento de tarefas.

O objetivo desse processo, através da utilização do protótipo, é a derivação de uma solução aceitável no domínio final da hierarquia para um problema especificado no domínio inicial.

A construção da solução desejada prossegue mediante transformações que representam operações de projeto sobre os objetos do sistema.

6.4 Arquitetura do Ambiente

Considere a figura 6.3 que apresenta a arquitetura do protótipo implementado.

A implementação segue as linhas definidas pelo *framework* de *blackboards*, consistindo em:

1. uma memória compartilhada pelos agentes – o *blackboard* –, composta por **objetos do espaço de solução** de um problema, devidamente estruturado em domínios, e por **objetos do sistema**;

2. um **conjunto de agentes** que se comunicam apenas através da memória compartilhada e cooperam para a resolução de um problema:

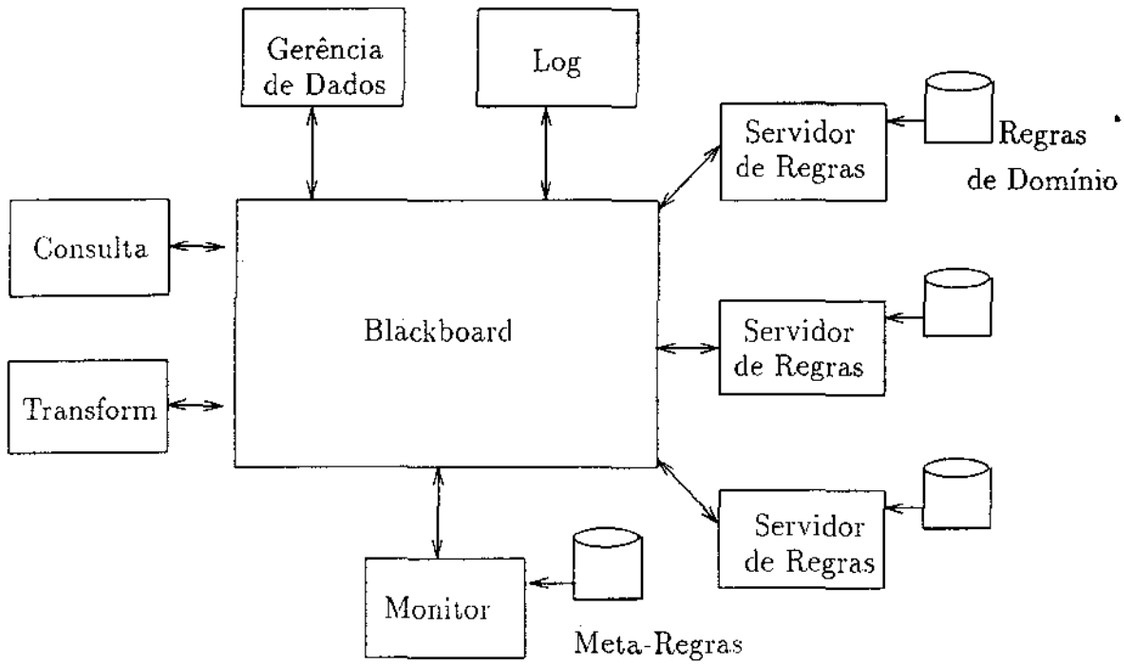


Figura 6.3: Arquitetura do Protótipo

3. um mecanismo de coordenação, o **monitor**, um tipo especial de agente que coordena as atividades realizadas por outros agentes e controla o estado dos objetos na memória compartilhada durante uma **sessão** de utilização do ambiente.

6.4.1 Mecanismo de Controle

O monitor detém o controle sobre a execução de tarefas **a nível de agentes** do sistema, executando o laço mostrado abaixo até que o usuário encerre a sessão.

- enquanto não for fim de sessão
  1. atualiza situação
  2. coleta ações possíveis
  3. mostra situação atual
     - objetos na memória
     - eventos pendentes
  4. seleciona ação a executar
     - seleciona sozinho, sempre que possível
     - senão, pede ao usuário para selecionar ação
  5. ativa algum agente para executar ação selecionada

A seleção de alternativas, quando é independente de interação com o usuário, requer a utilização de informações de coordenação contidas em meta-regras do ambiente.

As meta-regras encapsulam o conhecimento desejado sobre mecanismos de resolução para o problema, definem o comportamento e coordenam o funcionamento do ambiente. Elas manipulam dados de controle - predicados sobre objetos do sistema.

Por exemplo, algumas meta-regras de coordenação seriam:

1. **se** *domínio-selecionado*(d_i) **então** ativar servidor de regras para d_i
2. **se** *par-selecionado*(*regra*, *objeto*) **então** ativar mecanismo de transformação

Em paralelo à funcionalidade estabelecida através das meta-regras, a solução para um problema pode ser abordada de forma bi-direcional, através de **encadeamento progressivo e regressivo**. O encadeamento regressivo provê um mecanismo de coordenação que precede a adoção de qualquer outra estratégia de coordenação do ambiente.

- **Encadeamento Regressivo**

  O encadeamento regressivo faz uso das relações estabelecidas entre os objetos (através das transformações realizadas) e domínios envolvidos na resolução de um problema. Quando não há mais regras nem objetos interessantes de serem manipulados em um domínio, o sistema retrocede a um domínio que tem mapeamentos sobre o corrente (ou a objetos transformados em objetos do domínio corrente) e prossegue normalmente. Esse procedimento é detalhado na seção 6.6.

- **Encadeamento Progressivo**

  Dado um domínio e os objetos associados a ele, conceitualmente, todas as regras de transformação associadas ao domínio podem ser aplicadas em um dado instante, gerando um novo espaço de soluções.

  O processo de aplicação de regras pode prosseguir em outros domínios até que se chegue à solução desejada ou que todas as aplicações possíveis ou desejadas sejam efetuadas sem sucesso.

6.4.2 Objetos do Sistema

Os objetos do sistema são todos aqueles manipulados durante a utilização do ambiente. Os objetos que são manipulados por mais de um agente são mantidos na memória compartilhada. Os objetos do sistema são:

- domínios

  Os domínios definidos para a resolução de um problema não são mantidos na memória compartilhada, pois são manipulados apenas pelos servidores de regras e, só para consulta.

domínio	a "classe" domínio
classes	conjunto de classes que compõem o domínio

- relações

  As **relações** são objetos do sistema definidos sobre objetos, classes e domínios. Algumas relações *default* são:

  1. precede
  2. instância-de/tem-instância
  3. parte-de/tem-parte
  4. generalização-de/especialização-de
  5. evolução-de/involução-de
  6. domínio-selecionado
  7. domínio-validado
  8. objeto-criado

  Possivelmente, outras relações - específicas a cada domínio e encontradas em uma **tabela de símbolos** criada durante a fase de configuração do sistema - serão utilizadas pelo ambiente. Essas relações, conforme explicitado anteriormente, corresponderão a operações de projeto derivadas do núcleo básico definido.

- regras de transformação

  Regras de transformação mapeiam construções e definem relações entre elementos de um ou mais domínios. O formato de uma regra é mostrado abaixo:

nome	nome da regra
tipo	tipo da regra (evolução, decomposição, ...)
pré-condição	predicados sobre a cadeia de entrada
cadeia de entrada	lista de parâmetros/expressões
cadeia de saída	lista de parâmetros/expressões
pós-condição	predicados sobre a cadeia de saída
ação	procedimento que implementa as ações desejadas
contra-domínio	domínio para qual a regra está mapeada

- eventos

  Um **evento** é um acontecimento que requer um tratamento especial ou resposta imediata, fora do fluxo de controle corrente. Eventos são um mecanismo de sincronização entre os diversos agentes e o mecanismo de coordenação.

  Alguns eventos definidos e usados no ambiente são:

  1. INIT
  2. TRANSFORMACAO-OK
  3. TRANSFORMACAO-NOK
  4. DOMINIO-SELECIONADO
  5. DOMINIO-VALIDADO
  6. PAR-SELECIONADO
  7. OBJETO-CRIADO
  8. RETROCESSO
  9. TIMEOUT
  10. FIM-CONSULTA
  11. ERRO
  12. FIM-SESSAO

6.4.3 Mecanismo de Gerência de Dados

É o responsável pela interface com um repositório de dados. Ao iniciar uma nova sessão, o ambiente restaura o contexto da última sessão, recuperando objetos e relações de uma base de dados do ambiente. Da mesma forma, ao encerrar a sessão, o ambiente atualiza a base de dados com os dados existentes na memória compartilhada.

6.4.4 Servidor(es) de Regras

Um **servidor de regra** é o responsável pela seleção de todas as regras de transformação que ainda podem ser aplicadas em um determinado instante aos objetos de um domínio específico.

Um servidor retorna essa informação, gravando na memória compartilhada pares compostos por $< regra.local >$, que poderão ser lidos por outros agentes - em especial, o monitor, o mecanismo de transformação e o de documentação.

Cada servidor de regras tem conhecimento a respeito das regras de transformação de apenas um domínio **d**. Após esperar pela inicialização completa do ambiente, cada servidor executa o laço abaixo, até o final de uma sessão.

- *Loop*:
  1. seleciona pares (regra, local) ainda não selecionados anteriormente
  2. se houver ao menos um par (regra, local) selecionado, prossegue até o passo 6
  3. avisa ao monitor que pode colaborar
  4. espera OK do monitor (dominio-selecionado(d)=TRUE)
  5. quando o monitor der OK, o servidor é selecionado e grava os pares selecionados na memória compartilhada e sinaliza que há par(es) selecionado(s)
  6. volta ao passo 2
  7. se não houver pares selecionados, sinaliza que o domínio está validado (dominio-validado(d) = TRUE).

6.4.5 Mecanismo de Transformação

É o responsável pela aplicação de regras de transformação aos objetos selecionados.

Dados uma regra de transformação e um local de aplicação,

1. faz-se a unificação do local com a cadeia de entrada da regra
2. a pré-condição (condição de ativação) da regra é então avaliada e, caso seja verdadeira, a regra pode ser aplicada
3. a aplicação de uma regra a um local consiste na execução de uma ação definida na regra e implica na "sinalização" da sua pós-condição e na alteração do estado da memória compartilhada.

6.4.6 Mecanismo de Documentação

É o responsável pelo registro permanente das atividades realizadas durante uma sessão de alguma aplicação, documentando os passos de desenvolvimento, as alterações na memória compartilhada, seleção de regras e locais, resultados de transformações, caminhos alternativos para a solução de problemas, contexto da sessão imediatamente antes do fim da sessão, etc.

6.5 Detalhamento da Implementação

A implementação do protótipo foi realizada sobre a plataforma UNIX em uma rede de estações de trabalho SPARC Sun. A implementação foi feita em C e utiliza as primitivas do DPSK [Card87].

Os agentes são implementados através dos módulos enumerados abaixo:

1. Módulo de Coordenação

2. Módulo de Gerência de Dados
3. Módulo(s) Servidor(es) de Regras
4. Módulo de Transformação
5. Módulo de Documentação
6. Módulo de Apresentação e Consulta

além de um módulo de inicialização do ambiente.

6.5.1 Módulo de Inicialização

- Carrega configuração do ambiente
  - lê número, nomes dos domínios e hierarquia entre donínios
  - carrega tabelas do sistema (símbolos, eventos)
  - ativa módulo de gerência de dados para carregar domínios e regras de transformação
  - define predicados (relações) do ambiente (primitivos e os encontrados na tabela de símbolos)
- Carrega *log*
  - restaura contexto da última sessão
  - ativa módulo de gerência de dados para carregar instâncias (objetos) na memória compartilhada.

6.5.2 Utilização do DPSK

A implementação da arquitetura do protótipo é feita através da utilização das primitivas do DPSK.

Memória Compartilhada e Comunicação

A memória compartilhada do protótipo é definida pelo DPSK e manipulada por suas primitivas de comunicação.

Por exemplo, um objeto de uma classe é representado na memória compartilhada da seguinte forma:

```
{objeto [nome STR 3 _t4] [super STR 8 dominio1] [status INT 1]}
```

Obs.: [nome STR 3 _t4] é um dos slots da classe-dpsk *objeto*. **nome** é do tipo STR (cadeia), tem comprimento 3 e valor *_t4*. A superclasse do objeto é a classe *dominio1* e seu **status** tem valor 1.

A criação de um objeto é feita através da seguinte transação (onde **obj** é da classe-dpsk *objeto* e já teve os seus slots devidamente compostos):

```
tr = BeginTrans(obj.class, WRITE);
if(ISACCEPTED(tr)) {
  OpTrans(tr, MAKEOBJ, &obj, NULL, NULL);
  if (EndTrans(tr) != 1)
    Erro("Rolled back.\n");
}
else
  Erro("Rolled back.\n");
```

Sincronização e Tratamento de Exceção

Os eventos do sistema são manipulados através das quatro primitivas do DPSK.

Quando um servidor grava na memória compartilhada um par $< regra, objeto >$ ele sinaliza da seguinte forma:

```
AffirmEvent("par_selecionado");
```

Por sua vez, o laço de controle está sempre verificando os eventos pendentes para decidir o que fazer em seguida.

```
act = CheckEvent(tab_evento[FIM_SESSAO].nome);
if (act) {
   fprintf(stdout, "Evento afirmado: %s\n",
tab_evento[FIM_SESSAO].nome);
   fflush(stdout);
   NegateEvent(tab_evento[FIM_SESSAO].nome);
   SigGroup(SIGHUP, "xpadt");
}
```

No trecho acima, testa-se se o evento *FIM-SESSAO* ocorreu, através da primitiva **CheckEvent**. Se o resultado do teste for verdadeiro, o evento é negado e a primitiva de tratamento de exceções **SigGroup** é usada para sinalizar a todos os agentes do grupo *xpadt* o sinal SIGHUP (*hangup*) - - fim de execução. Esse sinal também provocará a ativação de um procedimento de tratamento de exceções pré-definido para este grupo.

6.6 Encadeamento Regressivo

O mecanismo de encadeamento regressivo adotado neste experimento permite o retrocesso a algum estágio anterior de projeto, através da identificação e utilização de relações específicas entre:

- objetos
- objetos e classes
- classes

- classes e domínios
- domínios

em uma representação hierarquizada de informações sobre o projeto.

As relações entre os elementos acima, em geral associadas a operações de projeto (restritas aos tipos básicos descritos anteriormente), são definidas como **predicados** e incluídas em tabelas de símbolos do ambiente configurado.

Quando uma solução incorreta ou insatisfatória for obtida – segundo os critérios do projetista – o ambiente deverá retroceder a um estágio no qual decisões de projeto alternativas possam ser tomadas. O mecanismo de encadeamento regressivo realiza este retrocesso e precede a adoção de qualquer estratégia mais genérica de coordenação do ambiente.

O encadeamento regressivo pode ser ativado em três circunstâncias e formas básicas:

- automaticamente, quando não houver eventos pendentes a serem tratados;
- automaticamente, quando um domínio for validado;
- a critério do usuário.

Quaisquer que sejam as circunstâncias de ativação, o mecanismo sempre atuará sobre relações entre objetos, classes e domínios.

Inicialmente, o retrocesso a estágios anteriores é feito sobre **relações estruturais existentes entre os objetos** – instâncias de classes de domínios.

Consideremos o exemplo mostrado através da figura 6.4. Após a aplicação de algumas operações a objetos dos domínios que compõem uma aplicação, obtivemos um objeto pertencente ao domínio D_3. Imaginando que tal objeto não é interessante, sob algum ponto de vista, o retrocesso conduzirá a algum estágio anterior do processo de projeto, a partir desse objeto.

O mecanismo constata que existe uma relação R_5 do tipo **tem-evolução**, associando $t12$ ao objeto $t11$ (que também pertence ao domínio D_3) – na forma *tem-evolução*$(t11, t12)$ – obtida após a aplicação de uma operação do tipo **evolução** ao objeto $t11$. O retrocesso ao objeto $t11$ será completado com a identificação e o cancelamento da transformação que realizou a operação Op_5 que, aplicada ao objeto $t11$, gerou o objeto $t12$.

Da mesma forma, podemos continuar o processo de retrocesso a partir do objeto $t11$. O mecanismo encontra duas relações do tipo **parte-de** associando $t11$ aos objetos $t9$, e $t10$, obtidas após uma operação de **agregação** sobre esses objetos que gera $t11$. Qualquer que seja o caminho escolhido, o retrocesso a $t9$ ou $t10$ implicará na eliminação do outro caminho, em função do cancelamento da transformação que implementou a operação Op_4.

Assim, o retrocesso a um estágio prévio no processo de projeto implica na eliminação de todos os objetos obtidos direta ou indiretamente a partir do objeto desprezado e relações associadas e, consequentemente, no cancelamento das operações de projeto realizadas sobre eles. As versões desprezadas, bem como os caminhos que levaram às mesmas deverão ser registrados no *log* da sessão.

Se o retrocesso não puder ser realizado através das relações entre os objetos, o ambiente deverá utilizar as **relações definidas entre instâncias e classes, entre classes, entre classes e domínios** e, por fim, **entre domínios**.

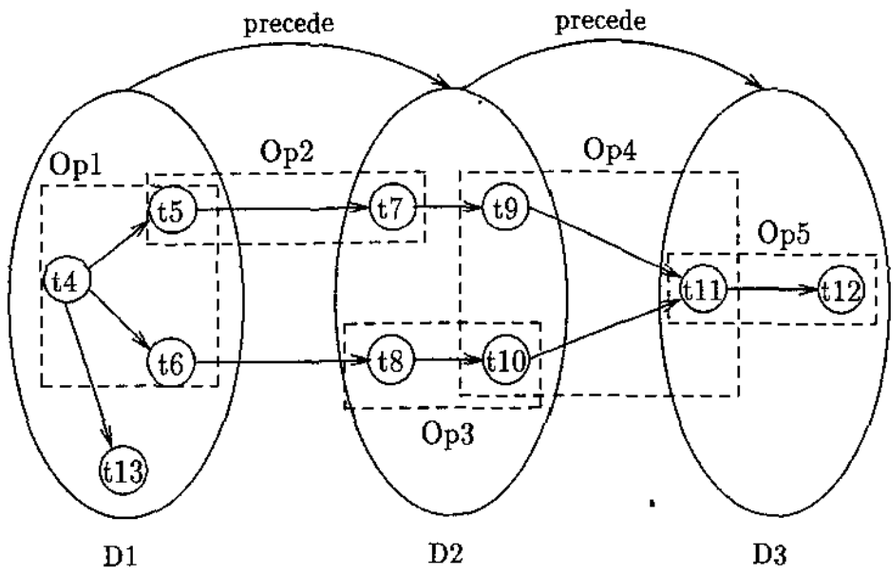


Figura 6.4: Domínios, Objetos e Relações

A principal relação entre instâncias e classes é a relação **tem-instância**; a principal relação entre classes é a relação **tem- generalização** e, a principal relação entre classes e domínios é a relação **parte-de**.

Tendo nos abstraído das instâncias e das relações entre elas, passamos a navegar por relações entre domínios. A única relação entre domínios considerada neste trabalho é a relação **precede**, que estabelece a hierarquia entre domínios. Através dela podemos retroceder a um ou mais domínios na hierarquia e, a partir daí, realizar o processo inverso: utilizar as relações **parte-de**, **tem- generalização** e **tem-instância** para descer no nível de abstração e voltar a manipular objetos e suas relações.

O mecanismo de encadeamento descrito aqui é exposto através de seu algoritmo e de um exemplo de utilização do ambiente.

6.6.1 Algoritmo

Seja *dom* o último domínio da hierarquia – o domínio objeto. O retrocesso é aplicado a *dom* através do algoritmo seguinte:

1. Recuperar da memória compartilhada os objetos que pertencem ao domínio *dom* e guardá-los em *objs*.

2. Se não existirem objetos **então** ir para o passo 5. Se existir ao menos um objeto **então** identificar relações que envolvem os objetos e guardá-las em *r-objs*. Se não existirem relações **então** ir para o passo 4. Se existir ao menos uma relação **então** pedir ao usuário que selecione uma opção de retrocesso, colocando-a em *rel*. Se alguma opção for selecionada em *rel* **então**:

(a) Identificar *obj*, o ponto de partida para o retrocesso

(b) Identificar *origem*, o objeto relacionado a *obj*

(c) Identificar *tipo*, o tipo da relação entre *obj* e *origem*

(d) Identificar *transf*, a transformação de tipo *tipo* aplicada sobre *origem* e que gerou *obj*.

3. Se a transformação for identificada **então**: (cancelar objetos e relações)

   (a) cancelar transformação *transf*

   (b) apagar relações que envolvem *obj*

   (c) apagar relações associadas a *transf* que envolvem *origem*

   (d) Se *tipo* = AGREGACAO **então**:

      i. Identificar *outra-origem*, o outro objeto que, agregado a *origem* gerou *obj*

      ii. apagar relações associadas a *transf* que envolvem *outra-origem*

   (e) Se *tipo* = DECOMPOSICAO[1] **então**:

      i. Identificar *outro-obj*, o outro objeto que foi gerado a partir de *origem*

      ii. apagar relações que envolvem *outro-obj*

4. Se não existirem relações entre objetos **então** subir o nível de abstração:

   (a) Escolher um objeto *obj* a partir de *objs*

   (b) Identificar *classe*, a classe do objeto *obj* através da relação *tem-instância*(*classe*, *obj*);

   (c) Percorrer a relação *subclasse-de* a partir de *classe* até encontrar uma classe sem superclasse;

   (d) Encontrar *domínio*, o domínio procurado através da relação *parte-de*(*classe*,*domínio*).

   (e) Encontrar domínios *n-doms* que precedem *domínio*.

5. Se não existirem objetos **então** encontrar domínios *n-doms* que precedem *dom*.

6. Se não existirem domínios em *n- doms* **então** perguntar ao usuário o que fazer. Se existir apenas um domínio em *n-doms* **então** *dom* recebe o valor de tal domínio e o algoritmo é chamado recursivamente. Se existirem dois ou mais domínios que precedem *dom* **então**:

   (a) o usuário é convidado a optar por um deles

   (b) *dom* recebe o valor do domínio escolhido

   (c) o algoritmo é chamado recursivamente.

[1] Obviamente, os conceitos de outra-origem e outro-obj podem ser estendidos para *n* outras origens e objetos. A opção feita por dois objetos de entrada e dois de saída, no máximo, reflete uma restrição de implementação e não uma restrição conceitual.

6.7 Um Exemplo

O exemplo aqui apresentado não possui vinculação com qualquer contexto real: não é mais do que um exemplo simples para ilustrar o mecanismo de encadeamento regressivo.

Os resultados apresentados na seção 6.7.4 correspondem ao *log* de uma sessão após a aplicação de algumas transformações que implementam as operações / ações do ambiente (figura 6.4), e ao cancelamento das mesmas através do retrocesso realizado a partir do objeto $t12$.

6.7.1 Domínios

Os domínios usados no exemplo são três e adotam nomes arbitrários:

- dominio1
- dominio2
- dominio3

As classes homônimas aos domínios não possuem classes componentes. Após a inicialização do ambiente temos[2]:

```
dominio(dominio3)
dominio(dominio2)
dominio(dominio1)

precede(dominio2, dominio3)
precede(dominio1, dominio2)

parte_de(classe3, dominio3)
parte_de(classe2,dominio2)
parte_de(classe1,dominio1)

classe(classe3)
classe(classe2)
classe(classe1)
```

6.7.2 Objetos do Sistema

- Relações

```
relacao(acao=14, rank=2, nome=tem_evolucao3_1)
relacao(acao=13, rank=2, nome=tem_evolucao2_2)
relacao(acao=12, rank=1, nome=predicado3_1)
relacao(acao=12, rank=2, nome=parte_de2_1)
```

[2] a e b são nomes de *slots*.

```
relacao(acao=11, rank=1, nome=predicado1_3)
relacao(acao=11, rank=2, nome=tem_evolucao1_3)
relacao(acao=11, rank=1, nome=predicado1_2)
relacao(acao=10, rank=2, nome=tem_parte1_2)
relacao(acao=9, rank=2, nome=tem_evolucao1_1)
relacao(acao=9, rank=1, nome=predicado1_1)
relacao(acao=2, rank=1, nome=existe)
relacao(acao=0, rank=2, nome=objeto_criado)
relacao(acao=0, rank=1, nome=dominio_validado)
relacao(acao=0, rank=1, nome=dominio_selecionado)
relacao(acao=5, rank=2, nome=tem_generalizacao)
relacao(acao=3, rank=2, nome=parte_de)
relacao(acao=2, rank=2, nome=tem_instancia)
relacao(acao=0, rank=1, nome=dominio)
relacao(acao=0, rank=1, nome=classe)
relacao(acao=0, rank=2, nome=precede)
```

O *rank* indica o número exato de elementos relacionados, ou seja, a cardinalidade da relação.

- Objetos

Os objetos – instâncias – ficam registrados na memória compartilhada através de duas estruturas: *tem-instancia* e *objeto*.

```
tem_instancia(dominio1,_t4)

objeto(status=1, super= classe1, nome=_t4)
```

6.7.3 Regras de Transformação

- Domínio 1

```
[regra1_1 64 existe(x)
[x] [y]
predicado1_1(x)=true && tem_evolucao1_1(x,y)=true
acao1_1 comentario dominio1]

[regra1_2 8 existe(x)
[x] [y1,y2]
tem_parte1_2(x,y1)=true && tem_parte1_2(x,y2)=true
acao1_2 comentario dominio1]

[regra1_3 64 existe(x)
[x] [y]
```

```
predicado1_2(x)=true && tem_evolucao1_3(x,y)=true &&
predicado1_3(y)=true
acao1_3 comentario dominio2]
```

- Domínio 2

```
[regra2_1 4 predicado1_3(x1)&&predicado1_3(x2)
[x1,x2] [y]
parte_de2_1(x1,y)=true && parte_de2_1(x2,y)=true &&
predicado3_1(y)=true
acao2_1 comentario dominio3]

[regra2_2 64 predicado1_3(x)
[x] [y]
predicado1_3(y)=true && tem_evolucao2_2(x,y)=true .
acao2_2 comentario dominio2]
```

- Domínio 3

```
[regra3_1 64 predicado3_1(x) [x] [y]
tem_evolucao3_1(x,y)=true
acao3_1 comentario dominio3]
```

6.7.4 Resultados

O *log* parcial de uma sessão, após a criação dos objetos $t4$ a $t12$ mostrados na figura 6.4 e do retrocesso até os objetos $t5$ e $t6$ no domínio 1 é mostrado abaixo:

```
n_objetos(12)

objeto(nome=_t12, super=classe3, status=0)
objeto(nome=_t11, super=classe3, status=0)
objeto(nome=_t1,  super=classe1, status=0)
objeto(nome=_t2,  super=classe2, status=0)
objeto(nome=_t3,  super=classe3, status=0)
objeto(nome=_t4,  super=classe1, status=1)
objeto(nome=_t5,  super=classe1, status=1)
objeto(nome=_t6,  super=classe1, status=1)
objeto(nome=_t7,  super=classe2, status=0)
objeto(nome=_t8,  super=classe2, status=0)
objeto(nome=_t9,  super=classe2, status=0)
objeto(nome=_t10, super=classe2, status=0)

tem_instancia(classe1,_t4)
```

```
tem_instancia(classe1,_t5)
tem_instancia(classe1,_t6)

existe(_t4)
existe(_t5)
existe(_t6)

dominio_selecionado(dominio1)
dominio_selecionado(dominio2)

selecionada(dominio=dominio1, regra=regra1_3, objeto1=_t4)
selecionada(dominio=dominio1, regra=regra1_1, objeto1=_t4)
...
selecionada(dominio=dominio1, regra=regra1_1, objeto1=_t5)

validada(objeto1=_t11, regra=regra3_1, dominio=dominio3)
...
validada(dominio=dominio2, regra=regra2_2, objeto1=_t7)

transformada(t_stamp=707127710.871942, saida1=_t12, acao=acao3_1,
tipo=64, regra=regra3_1, objeto1=_t11, dominio=dominio3)

transformada(t_stamp=707127914.053287, saida1=_t11, acao=acao2_1,
tipo=4, regra=regra2_1, objeto2=_t9, objeto1=_t10, dominio=dominio2)
...
transformada(t_stamp=707130118.271788, dominio=dominio2, objeto1=_t7,
regra=regra2_2, tipo=64, acao=acao2_2, saida1=_t9)

cancelada(t_stamp=707130363.453304, saida1=_t7, acao=acao1_3,
tipo=64, regra=regra1_3, objeto1=_t5, dominio=dominio1)
...
cancelada(t_stamp=707127710.286678, saida1=_t12, acao=acao3_11,
tipo=64, regra=regra3_1, objeto1=_t11, dominio=dominio3)

tem_parte1_2(_t4, _t5)
tem_parte1_2(_t4,_t6)
```

Capítulo 7

Considerações Finais

Neste trabalho, procuramos examinar **transformações** como aspecto central de ambientes de apoio a desenvolvimento de software, com ênfase em operações básicas de projeto e mecanismos de coordenação de projeto, e tendo em mente as perguntas básicas que o motivaram (expostas no capítulo 1).

No capítulo 2: a) apresentamos um resumo a respeito de estudos gerais sobre projeto, analisando a tarefa de **desenvolver software** como uma instância específica de projeto e discutindo dois aspectos críticos a considerar em ambientes de apoio a projeto, a saber: **operações básicas** e **mecanismos de coordenação** de atividades, e b) propusemos um conjunto mínimo de operações básicas de projeto.

No capítulo 3 descrevemos com maior detalhe o modelo PW de Lehman, com a preocupação básica de introduzir um **arcabouço conceitual** dentro do qual considerar **transformações**.

No capítulo 4 fizemos um apanhado geral sobre como o conceito de **transformações** tem sido proposto e tratado na literatura, incluindo um breve *survey* sobre sistemas transformacionais significativos na área.

No capítulo 5 propusemos uma **arquitetura de referência** para ADSs baseada no modelo de *blackboards* e apresentamos uma instância concreta centrada no uso de DPSK [Card87].

No capítulo 6 descrevemos um experimento concreto de implementação das idéias expostas nos capítulos anteriores – designado por AADT (sigla para **A**mbiente de **A**poio ao **D**esenvolvimento **T**ransformacional) – escrito em C. Um aspecto particularmente destacado é a implementação de um mecanismo de encadeamento regressivo que se vale das operações (i.e., relações) básicas descritas no capítulo 2 para navegar entre objetos e domínios.

o o o

Uma questão vital ligada a desenvolvimento de software diz respeito a ambientes de apoio. É provável que o cenário de utilização (no sentido mais pragmático que se possa imaginar) de ambientes de desenvolvimento não sofra alterações significativas nos próximos anos: alguns métodos coexistirão pacificamente, cada qual com seu jargão, notações, pontos fortes e fracos.

O **retoque** para este cenário parece estar calcado na evolução de ambientes genéricos porém configuráveis, onde a diversificação poderá ser mantida a nível de aplicações, métodos e ferramentas, mas assegurando, no mínimo, uma certa coerência em aspectos básicos, tais como: arquitetura, comunicação e coordenação a nível global, operações básicas e interface.

A arquitetura de referência proposta é condizente com o cenário acima descrito e permite a instanciação de ambientes para apoio a métodos específicos.

7.1 Contribuições deste Trabalho

As contribuições mais importantes deste trabalho são:

- a proposição de uma taxonomia genérica para transformações;
- a proposição de uma arquitetura de referência para ambientes de apoio a desenvolvimento transformacional;
- a proposição de um mecanismo de coordenação simples que atua a nível de relações estruturais entre objetos;
- a exploração inicial de um mecanismo de coordenação global *à la blackboards*, usando o DPSK.

Em paralelo, realizamos alguns experimentos interessantes sobre formalização e transformação entre domínios, apresentados nos apêndices A e B.

7.2 Trabalhos Futuros

O experimento concreto apresentado no capítulo anterior representa apenas uma exploração inicial de idéias que podem ser melhor desenvolvidas em trabalhos de pesquisa e experimentação futuros. Dentre as possíveis abordagens destacamos:

- implementação real do oportunismo preconizado pelo modelo de *blackboards*;
- estabelecimento de estratégias para a definição de linguagens de especificação para domínios que compõem ambientes personalizados;
- abordagens rigorosas para a definição de transformações e objetos manipulados;
- projeto da interface genérica do ambiente;
- personalização do ambiente genérico para o suporte a métodos conhecidos, através da definição de domínios que, em conjunto, implementam os mesmos, mantendo a coerência em torno da arquitetura proposta e operações básicas de projeto.

Apêndice A

QuickSort

A.1 Introdução

Para ilustrar algumas idéias aqui abordadas, utilizaremos um exemplo simples e conhecido: o de ordenar uma sequência de inteiros através do algoritmo de Quicksort de Hoare, considerado o melhor algoritmo de ordenação existente.

Um algoritmo de ordenação **genérico** é especificado da seguinte forma:

```
Sort(x) <= Ordered(Perm(x))
```

ou seja, ordenar uma sequência x corresponde a encontrar uma permutação ordenada (crescente ou decrescente) da sequência x.

Usando expressões-ZF (ou compreensões) e KRC, teríamos a seguinte especificação, considerando listas de inteiros:

```
perm [] = [[]]
perm x = {a:p; a <- x , p <- perm(x -- [a])}

ordered [] = T
ordered [a] = T
ordered (a:b:x) = a <= b and ordered (b:x)

sort x = ordered perm x
```

A derivação formal de um algoritmo iterativo a partir desta especificação aparece em [Darl78] e [Stei89].

A.2 Quicksort em Linguagem Natural

O algoritmo de Quicksort é essencialmente recursivo e baseia-se no conceito de partição, dividindo para conquistar. Nele, lidamos com três tipos de operação:

- decompor (particionar)

- ordenar
- compor (combinar)

Uma especificação deste problema em linguagem natural é apresentada abaixo:

1. Uma sequência de 0 ou 1 elemento já está ordenada.
2. Uma sequência com 2..n elementos é ordenada:
   (a) selecionando dentre os seus elementos um elemento de comparação;
   (b) dividindo-a em duas partes, com a primeira contendo elementos menores ou iguais do que o elemento de comparação e a segunda contendo elementos maiores;
   (c) ordenando as duas partes obtidas;
   (d) compondo a primeira parte com o elemento de comparação e a segunda parte.

Teoricamente, a divisão em duas partes iguais (divisão na metade) é necessária para se obter um desempenho ótimo, mas, na prática, uma aproximação da metade é boa.

A.3 Quicksort em KRC

```
quick []     = []
quick (a:x) = {b; b <- x; b <= a} ++ [a] ++ {b; b <- x; b > a}
```

A.4 Quicksort em Notação Funcional

```
(defun quick (lista)
   (cond
      ( (null lista) lista)
      ( T (append (quick (menor (car lista)
                                (cdr lista)))
                  (cons (car lista)
                        (quick (maior (car lista)
                                      (cdr lista))
                        )
                  )
          )
       )))

(defun menor (h t)
   (cond
      ( (null t) t)
      ( (<= (car t) h) (cons (car t)
                             (menor h (cdr t))
                       ))
```

```
      ( T (menor h (cdr t)))
   ))

(defun maior (h t)
   (cond
      ( (null t) t)
      ( (<= (car t) h) (maior h (cdr t)))
      ( T (cons (car t)
                (maior h (cdr t))))
   ))
```

A.5 Quicksort em Notação Lógica

```
domains
   intlist = integer*

predicates
   quick (intlist,intlist)
   particao (integer,intlist,intlist,intlist)
   append (intlist,intlist,intlist)

clauses
   quick ([],[]).
   quick ([H|T],R) :-
      particao (H,T,ME,MA),
      quick (ME,X),
      quick (MA,Y),
      append (X,[H|Y],R).

   particao (_,[],[],[]).
   particao (E,[H|T],[H|X],Y) :-
      H <= E,
      particao (E,T,X,Y).
   particao (E,[H|T],Y,[H|X]) :-
      H > E,
      particao (E,T,Y,X).

   append (X,[],X).
   append ([],X,X).
   append ([H|T],X,[H|Y]) :-
      append (T,X,Y).
```

A.6 Quicksort em Notação Procedural

```
quick (item,inicio,fim)
char *item;
int inicio,fim;
{
   int i,j;

   if (fim > inicio) {
      particao(item,&i,&j,inicio,fim);
      quick(item,inicio,j);
      quick(item,i,fim);
   }
}

particao (item,i,j,inicio,fim)
char *item;
int *i, *j;
int inicio, fim;
{
   char x, t;
   int ii, jj;

   ii = inicio;
   jj = fim;
   x = item [(inicio+fim)/2];

   do {
      while ((item[ii] < x) && (ii < fim)) ii++;
      while ((x < item[jj]) && (jj > inicio)) jj--;
      if (ii <= jj) {
      t = item[ii];
      item[ii] = item[jj];
      item[jj] = t;
      ii++;
      jj--;
      }
   } while (ii <= jj);
   *i = ii;
   *j = jj;
}
```

Apêndice B

De uma Visão Funcional a uma Visão de Objetos

Um problema não muito simples que tem merecido alguns estudos é o mapeamento do Modelo de Fluxo de Dados – representado através de DFDs – para um Modelo de Objetos, como mostra a figura B.1.

Com a tendência atual em Programação Orientada a Objetos, estudos têm sido feitos [Alab88], [Rote87] em relação ao problema de "migrar" da tradicional visão funcional preconizada pela decomposição funcional de um sistema, para uma visão de objetos correspondendo ao projeto do mesmo sistema segundo técnicas de projeto orientadas a objetos. "Migrar" nada mais é do que transformar um modelo em outro. Esta transformação será parcialmente automatizada e dependerá de certo grau de interação com o projetista, que avaliará critérios e tomará algumas decisões de projeto.

B.1 O Domínio DFD

Os DFDs são, inegavelmente, a parte mais bem sucedida da metodologia de Análise Estruturada. Utilizam notação gráfica simples para representar os seus elementos básicos (entidades, depósitos,

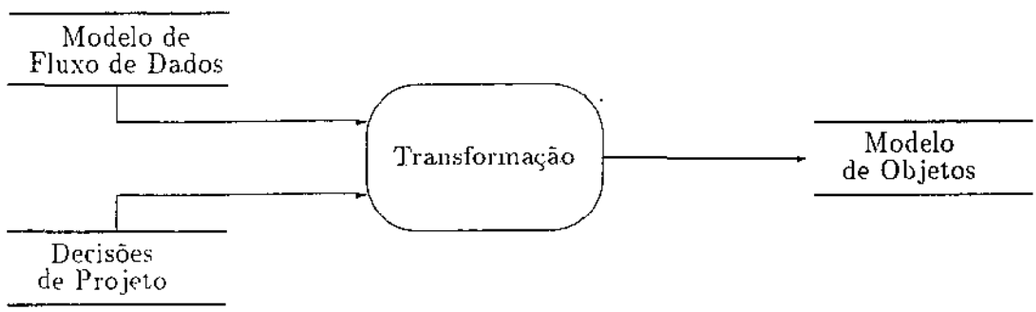


Figura B.1: Transformação de DFD para Objetos

processos e fluxos de dados - e de controle para as versões estendidas), que é considerada semi-formal e que facilita a comunicação com o usuário e entre projetistas.

B.1.1 Descrição em Linguagem Natural

- **Objetos**
  - Entidades Externas
  - Depósitos de dados
  - Fluxos de dados
  - Processos
- **Operações**
  - Definir objetos
  - Detalhar processsos
- **Restrições**

1. Deve existir ao menos um fluxo de entrada e um de saída para cada processo definido com, pelo menos, um fluxo de entrada (saída) que não seja fluxo de saída (entrada) para o mesmo processo;
2. Uma entidade (ou um depósito) deve possuir ao menos um fluxo de entrada ou de saída;
3. Os nomes dos processos devem ser únicos;
4. Idem para entidades. depósitos;
5. Todo fluxo de dados associado a uma entidade (depósito) deve também estar associado a um processo;
6. Todo fluxo de dados de entrada/saída de um processo "pai" deve ser representado no seu diagrama detalhado, podendo ser fluxo de entrada ou saída de um ou mais processos "filhos" (consistência estrutural);
7. A especificação deve ser feita de forma hierárquica e não recursiva (consistência global).

B.1.2 Formalização de DFDs

Com a utilização crescente de métodos formais no processo de desenvolvimento de software (ou, pelo menos, com o "militantismo" em torno do tema), e o preconceito relacionado à dificuldade em lidar diretamente com notações formais (os engenheiros de software e Cia. não têm que ser matemáticos), surgiu uma idéia de conciliatória e bastante razoável: prover uma notação formal para DFDs. de forma a satisfazer a "gregos" (matemáticos) e "troianos" (engenheiros e Cia.).

A formalização de DFDs permite que. a depender do formalismo utilizado. eles possam ser diretamente usados como especificação formal. executável (protótipo) ou não. verificados. etc.

Linha 1: Redes de Petri

Tse & Pong defendem em seu artigo "Towards a Formal Foundation for De Marco Data Flow Diagrams" [Tse89] o uso de Redes de Petri como notação formal para DFDs, pois:

- Redes de Petri podem ser representadas grafica e algebricamente (transições e lugares correspondem, respectivamente, a fluxos de dados e processos). O conceito de subrede é suportado, correspondendo à estrutura hierárquica de um sistema. Também suporta paralelismo.
- A representação algébrica provê uma base teórica para a análise de uma especificação. Os conceitos de "token" e marca (inexistentes em outros modelos) provêm um excelente meio de análise de propriedades comportamentais.
- Em função do seu formalismo rico, Redes de Petri podem ser usadas como ferramenta para o projeto e teste de sistemas. Entretanto, não se adequam para a fase de análise, pois são de difícil compreensão. Se o aspecto amigável dos DFDs for adicionado às Redes de Petri, a linguagem de especificação resultante possuirá as vantagens de ambos.

Seguindo tais argumentos, os autores propõem uma linguagem de especificação – Formal Data Flow Diagrams (FDFD) – definida em duas formas equivalentes: uma gráfica e outra simbólica. A representação gráfica retém as vantagens dos DFDs, em especial, o caráter amigável. A representação simbólica utiliza a fundamentação algébrica das Redes de Petri. FDFD possui uma sintaxe formal que permite o seu processamento automático. Há uma correspondência de um-para-um entre as duas representações, o que assegura a consistência e *traceability* entre elas.

Linha 2: Especificação Algébrica

Docker & France propõem uma abordagem para a parte *upper CASE* do desenvolvimento de software que integra uma técnica informal de prototipagem com outra formal [Dock89], [Fran89]. A técnica informal baseia-se em ferramentas e técnicas de Análise Estruturada e dá suporte a especificações executáveis. A técnica formal baseia-se em especificação algébrica e suporta a geração de especificações formais a partir das especificações obtidas através da Análise Estruturada. Esta abordagem ameniza os problemas decorrentes da falta de formalismo na Análise Estruturada, sem sacrificar a clareza das especificações obtidas.

A técnica formal consiste de duas partes: PL (Picture Level) e SL (Specification Level). PL é um sistema para investigação formal da estrutura sintática de DFDs. SL consiste de um conjunto de técnicas formais para a especificação de objetos e do comportamento de aplicações representadas através de DFDs estendidos.

PL e SL utilizam uma técnica especial de especificação algébrica que gera especificações relacionais "positivo-negativas" (extensões de especificações relacionais utilizadas por Astesiano [Aste86] em conjunto com a semântica observacional).

A especificação completa do domínio DFD utilizando-se especificação algébrica é dada abaixo:

```
PLflow = Flowname +
```

```
    Signature
     sorts plflow
     constructors
         mkflow: flowname -> plflow
     ok_predicate
         okflow: plflow
    Laws f:flowname
 F1. okflow(mkflow(f))


 PLentity = Set(PLflow) + Entnames +
    Signature
     sorts plentity
     constructors
         mkplentity: entname, set(plflow), set(plflow) ->
plentity
    observation functions
         geteinputs, geteoutputs: plentity -> set(plflow)
         getename: plentity -> entname
    ok_predicate
         okent: plentity
    Laws in, out: set(plflow); n: entname
 E1. isempty(in-int-out), ~isempty(in), ~isempty(out) <->
     okent(mkplentity(n,in,out))
 E2. geteinputs(mkplentity(n,in,out)) = in
 E3. geteoutputs(mkplentity(n,in,out)) = out
 E4. getename(mkplentity(n,in,out)) = n


PLstore = Set(PLflow) + Storenames +
   Signature
     sorts plstore
     constructors
         mkplstore: storename, set(plflow), set(plflow) ->
plstore
    observation functions
         getsinputs, getsoutputs: plstore -> set(plflow)
         getsname: plstore -> storename
    ok_predicate
         okstore: plstore
    Laws in, out: set(plflow); n: storename
E1. isempty(in-int-out), ~isempty(in), ~isempty(out) <->
     okstore(mkplstore(n,in,out))
E2. getsinputs(mkplstore(n,in,out)) = in
E3. getsoutputs(mkplstore(n,in,out)) = out
```

```
E4. getsname(mkplstore(n,in,out)) = n


PLprocess = Set(PLflow) + Procnames +
    Signature
     sorts plprocess
     constructors
         mkplprocess: procname, set(plflow), set(plflow) ->
plprocess
    observation functions
         getpinputs, getpoutputs: plprocess -> set(plflow)
         getpname: plprocess -> procname
    ok_predicate
         okproc: plprocess
    Laws in, out: set(plflow); n: procname
PR1. isempty(in-int-out), ~isempty(in), ~isempty(out) <->
     okproc(mkplprocess(n,in,out))
PR2. getpinputs(mkplprocess(n,in,out)) = in
PR3. getpoutputs(mkplprocess(n,in,out)) = out
PR4. getpname(mkplprocess(n,in,out)) = n


Procstruct = Set(PLprocess) + Set(PLflows) + Pstnames
    Signature
     sorts procstruct
     constructors
            makeleaf: plprocess -> procstruct
         mkprocstruct: plprocess, struct -> procstruct
     observation functions
         getpstinputs, getpstoutputs: procstruct ->
set(plflow)
         getpstprocs: procstruct -> set(plprocess)
            getpstname: procstruct -> pstname
     ok_predicate
         okprocstruct: boolean
    Laws p: plprocess; st: struct
PST1.  okprocstruct(makeleaf(p))
PST2.  getpinputs(p) = getininterface(st),
       getpoutputs(p) = getoutinterface(st) ->
              okprocstruct(mkprocstruct(p,st))
PST3.  getpstinputs(makeleaf(p)) = getpinputs(p)
PST4.  getpstinputs(mkprocstruct(p,st)) = getininterface(st)
PST5.  getpstoutputs(makeleaf(p)) = getpoutputs(p)
PST6.  getpstoutputs(mkprocstruct(p,st)) =
getoutinterface(st)
```

```
PST7.  getpstprocs(makeleaf(p)) = insort(p,empty)
PST8.  getpstprocs(mkprocstruct(p,st)) = getprocs(st)
PST9.  getpstname(makeleaf(p)) = getpname(p)
PST10. getpstname(mkprocstruct(p,st)) = getpname(p)


Struct = PLset(Procstruct) + PLset(PLentity) +
PLset(PLstore) +
    Signature
     sorts struct
     constructors
         initstruct: procstruct -> struct
         mkstruct1: procstruct, struct -> struct
         mkstruct2: plstore, struct -> struct
     observation functions
         getinflows, getoutflows,
         getpinflows, getpoutflows,
         getininterface, getoutinterface: struct ->
set(plflow)
         getprocs: struct -> set(procstruct)
         getstores: struct -> set(plstore)
     ok_predicate
         okstruct: struct -> boolean
    Laws p, p1: procstruct; st, st1: struct; ds: plstore
ST1.  okstruct(initstruct(p))
ST2.  ~isin(getpstname(p),getnames(getprocs(st))),
       isempty(getpstoutputs(p)-int-getoutflows(st)) ->
                              okstruct(mkstruct1(p,st))
ST3.  ~isin(getsname(ds),getnames(getstores(st))),
       isempty(getsoutputs(ds)-int-getoutflows(st)),
       issubset(getsinputs(ds),getpoutflows(st)),
       issubset(getsoutputs(ds),getpinflows(st)) ->
                         okstruct(mkstruct2(ds,st))
ST4.  getprocs(initstruct(p)) = insort(p,empty)
ST5.  getstores(initstruct(p)) = empty
ST6.  getprocs(mkstruct1(p1,st)) = p1 + getprocs(st)
ST7.  getprocs(mkstruct2(ds,st)) = getprocs(st)
ST8.  getstores(mkstruct1(p1,st)) = getstores(st)
ST9.  getstores(mkstruct2(ds,st)) = ds + getstores(st)
ST10. getoutflows(mkstruct1(p1,st)) = getpstoutputs(p1) +
getoutflows(st)
ST11. getoutflows(mkstruct2(ds,st)) = getsoutputs(ds) +
getoutflows(st)
ST12. getoutflows(initstruct(p)) = getpstoutputs(p)
```

```
ST13. getinflows(mkstruct1(p1,st)) = getpstinputs(p1) +
getinflows(st)
ST14. getinflows(mkstruct2(ds,st)) = getsinputs(ds) +
getinflows(st)
ST15. getinflows(initstruct(p)) = getpstinputs(p)
ST16. getpoutflows(mkstruct1(p1,st)) = getpstoutputs(p1) +
getpoutflows(st)
ST17. getpoutflows(mkstruct2(ds,st)) = getpoutflows(st)
ST18. getpoutflows(initstruct(p)) = getpstoutputs(p)
ST19. getpinflows(mkstruct1(p1,st)) = getpstinputs(p1) +
getpinflows(st)
ST20. getpinflows(mkstruct2(ds,st)) = getpinflows(st)
ST21. getipnflows(initstruct(p)) = getpstinputs(p)
ST22. getininterface(st) = getinflows(st) - getoutflows(st)
ST23. getoutinterface(st) = getoutflows(st) - getinflows(st)
ST24. getprocs(st1) = getprocs(st),
      getstores(st1) = getstores(st) <-> st1 = st


SimpleApplic = Struct +
    Signature
     sorts plapplic
     constructors
         mkapplic: struct, set(plentity) -> plapplic
            mkapplic2: struct, set(plentity) -> struct
     ok_predicate
         okapplic: plapplic -> boolean
    Laws se: set(plentity); st:struct
A1. ~isempty(se),
     isempty(getallinputs(se)-int-getalloutputs(se)),
     getallinputs(se) = getoutinterface(st),
     getalloutputs(se) = getininterface(st) <->
okapplic(mkapplic(st,se))
A2. isempty(se) <-> okapplic(mkapplic2(st,se))
A3. ~isempty(se),
     isempty(getallinputs(se)-int-getalloutputs(se)),
     issubset(getallinputs(se),getpoutflows(st)),
     issubset(getalloutputs(se),getpinflows(st)) <->
                          okapplic(mkapplic(st,se))
```

Exemplo:

```
--------------------------- Nivel 2 ----------------------

P5 = mkplprocess('conversor',
                 {mkflow('materia digitada')},
                 {mkflow('materia convertida')})

P4 = mkplprocess('programa-paginador',
                 {mkflow('lay-out'), mkflow('materia
convertida')},
                 {mkflow('materia paginada')})

PS5 = makeleaf(P5)
PS4 = makeleaf(P4)

S3 = mkstruct1(PS5, initstruct(PS4))

--------------------------- Nivel 1 ----------------------

E4 = mkplentity('copy-desk',
                {mkflow('materia p/ser revisada')},
                {mkflow('materia revisada')})

P3 = mkplprocess('estacao-de-trabalho',
                 {mkflow('materia'), mkflow('materia
revisada')},
                 {mkflow('materia p/ser revisada'),
                  mkflow('materia digitada')})

P2 = mkplprocess('paginacao',
                 {mkflow('lay-out'), mkflow('materia
digitada')},
                 {mkflow('materia paginada')})

PS3 = makeleaf(P3)
PS2 = mkprocstruct(P2,S3)

S2 = mkstruct1(PS3, initstruct(PS2))
SE2 = mkapplic2(S2,{E4})
```

```
--------------------------- Nivel 0 ----------------------

E3 = mkplentity('editor-da-pagina',
                {},
                mkflow('lay-out'))

E2 = mkplentity('jornalista',
                {},
                mkflow('materia'))

E1 = mkplentity('Fotocomposicao',
                mkflow('materia-paginada),
                {})

P1 = mkplprocess('sistema-de-editoracao',
                 {mkflow('lay-out'), mkflow('materia')},
                 {mkflow('materia-paginada')})

PS1 = mkprocstruct(P1,SE2)

S1 = initstruct(PS1)

DFD = mkapplic(S1,{E1,E2,E3})
```

Linha 3: Especificação Funcional com StandardML

```
(* ENTITY *)

abstype entity = makentity of string * flow set * flow set
  with
    exception Error

    fun okentity(ename:string,inflows:flow set,outflows:flow
set) =
      (isempty(intersect(inflows,outflows)) andalso
      (not (isempty(inflows)) orelse
       not (isempty(outflows))))

    fun mkentity(ename,inflows,outflows) =
      if okentity(ename,inflows,outflows)
      then makentity(ename,inflows,outflows)
      else raise Error

    fun eqe(makentity(e1,i1,o1),makentity(e2,i2,o2)) =
```

```
        (e1 = e2)  andalso
        equ(i1,i2) andalso
        equ(o1,o2)

    (* funcoes de observacao *)

    fun geteinflows(makentity(ename,inflows,outflows)) =
inflows

    fun geteoutflows(makentity(ename,inflows,outflows)) =
outflows

    fun getename(makentity(ename,inflows,outflows)) = ename
  end;
```

Linha 4: Linguagem de Domínio

Uma linguagem de domínio é uma linguagem de especificação na qual as construções sintáticas e os aspectos semânticos correspondem exatamente aos conceitos (objetos e operações) de um domínio de aplicação.

O sistema DRACO adota esta linha para definir e formalizar domínios. A sintaxe de uma nova linguagem de domínio é definida através de uma notação do tipo BNF; a semântica é definida através de mapeamentos adequados entre os conceitos da linguagem de domínio em questão e os conceitos de outra(s) linguagem(ns) de domínio conhecidas por DRACO (ou seja, previamente definidas). O usuário pode, então, formular um problema específico de algum domínio de aplicação utilizando a sua própria linguagem.

Uma linguagem de domínio para DFDs deverá representar de forma natural os conceitos encontrados nesse domínio e permitir que as restrições sejam facilmente verificadas.

B.2 O Domínio de Objetos

B.2.1 Modelo de Objetos

"Modelo de Objetos" corresponde a um modelo do sistema de programação que usa o estilo de programação orientada a objetos, ou, genericamente falando, um modelo para o processo de desenvolvimento de um sistema de programação que suporta a programação orientada a objetos.

Os principais conceitos subjacentes ao paradigma de objetos estão descritos no capítulo 1 e deveriam ser representados por um modelo de objetos genérico.

Abaixo apresentamos duas propostas encontradas em [Rote87] e [Alab88].

Modelo de Booch-Rothemberg

Propõe-se aqui uma linguagem gráfica de representação que expressa um modelo de objetos contendo os conceitos básicos mostrados na figura B.2

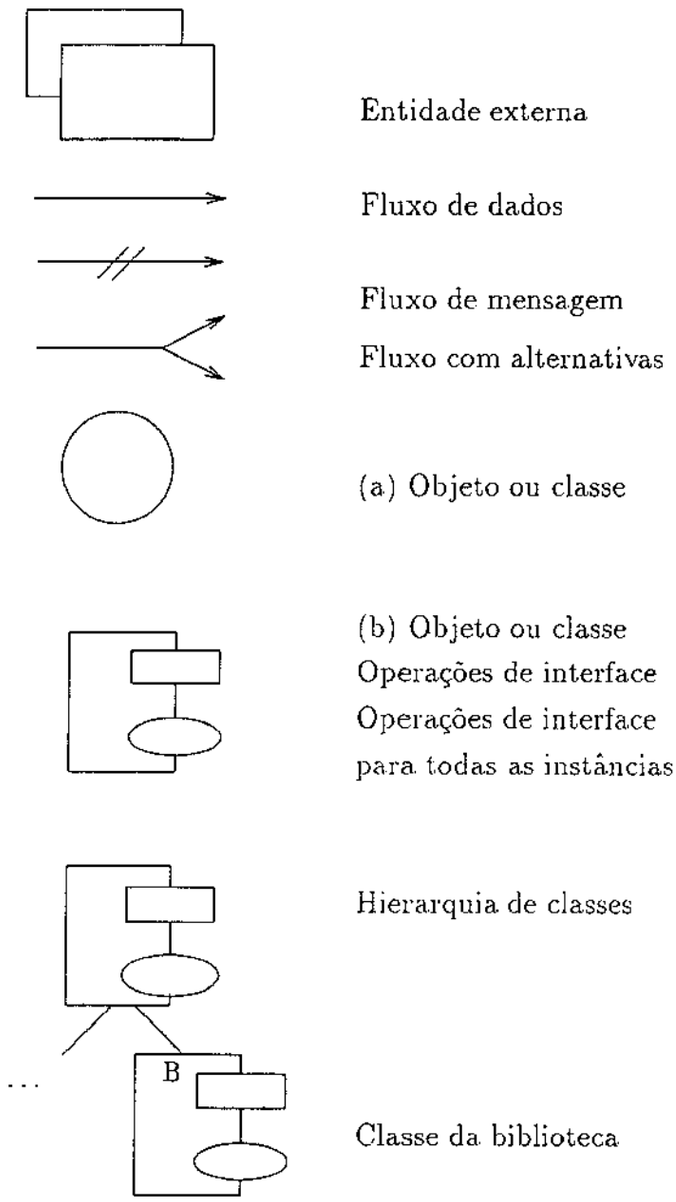


Figura B.2: Uma Linguagem para Representação do Modelo de Objetos

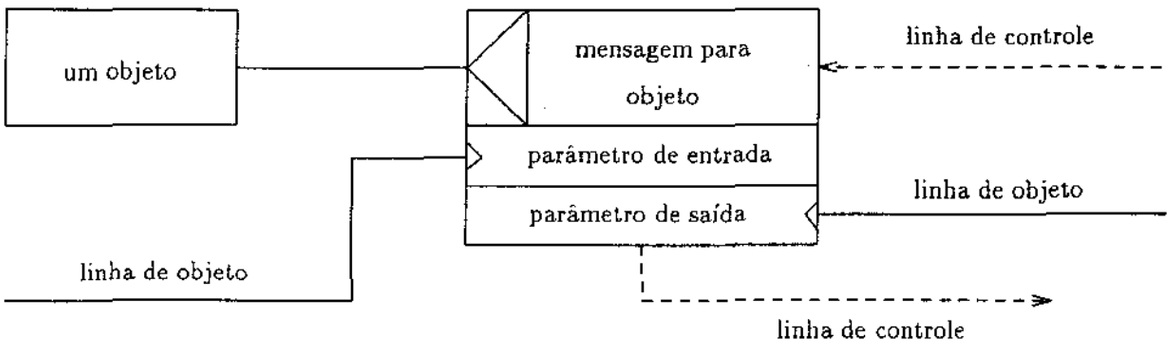


Figura B.3: Convenções de um FDC

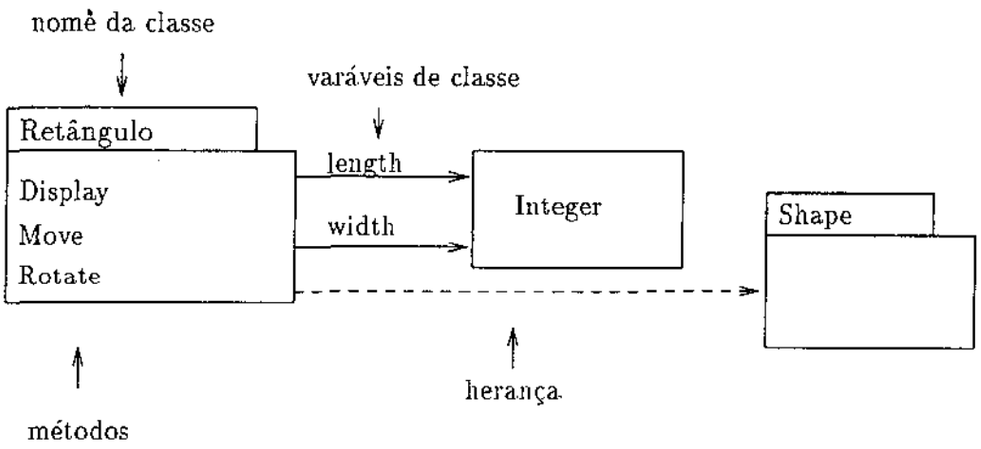


Figura B.4: Convenções de um OSC

Modelo de Alabiso

A descrição física do Modelo de Objetos é feita através de:

1. Functional Design Charts (FDCs)
   São usados para expressar o comportamento funcional de objetos, seguindo as convenções mostradas na figura B.3.

   Em um FDC, linhas de controle correspondem a ações (envio de mensagens) e linhas de objeto representam depósitos de objetos (*receiver*).

2. Object Structure Chart (OSCs)
   São usados para expressar e detalhar a estrutura de dados dos objetos, como mostra a figura B.4.

B.3 Métodos de Transformação DFD-MO

B.3.1 Método de Booch-Rothemberg

Cada uma das entidades do DFD terá uma correspondente no Modelo de Objetos. A transformação é baseada na idéia de que os objetos do modelo são originados dos depósitos de dados em conjunto com os processos correspondentes. A análise de correspondência de processos em relação a objetos deve ser feita no nível mais detalhado de cada processo.

Método de Transformação

1. Identificar objetos
   "Os depósitos de dados que aparecem no DFD são candidatos a darem origem a objetos". Convém também identificar os seus atributos.

   Os depósitos de dados transitórios possivelmente não originarão objetos.

2. Identificar as operações de cada objeto
   "Analisar os processos em seu nível mais detalhado, identificando a que objetos correspondem".

   - Se um processo em seu nível mais detalhado corresponder a mais de um objeto, deve-se tentar decompô-lo em processos menores;
   - Objetos podem ser criados apenas para acomodar processos;
   - Objetos com características passivas geralmente ficam sem operações de consulta.

3. Representar objetos, suas operações e as entidades externas na linguagem proposta.

4. Completar a representação com os fluxos de dados.

5. Definir as interfaces dos objetos
   "Operações que não recebem fluxos de dados ou só os recebem do objeto a que pertencem são candidatas a serem transformadas em operações privativas do objeto".

6. Avaliação final
   Controle de Coesão.

B.3.2 Método de Alabiso

Alabiso propõe um método para transformar Análise de Fluxo de Dados em Projeto Orientado a Objetos. Esta transformação é realizada, em termos gerais:

1. extraindo informação do modelo de fluxo de dados;

2. enriquecendo-a com decisões de projeto;

3. produzindo um modelo de projeto orientado a objetos.

Método de Transformação

1. Interpretar dados, processos, depósitos e entidades em termos de conceitos orientados a objetos;

2. Interpretar a hierarquia do DFD em termos de decomposição de projeto;

Este método trabalha com um modelo de fluxo de dados estendido com processos e fluxos de controle, o que não será considerado nesta descrição.

Interpretação de Conceitos

- Dados são objetos

- Processos são métodos
  A ativação de um processo corresponde ao envio de uma mensagem.

- Entidades são objetos
  Os atos de receber e enviar dados são associados à invocação de métodos.

- Depósitos são objetos
  Métodos podem ser adicionados para extrair/colocar dados de/em depósitos de dados.

- DFDs serão convertidos em FDCs (Functional Design Charts)

- A transformação dos dados (contidos no Dicionário de Dados) em objetos resultará em OSCs (Object Structure Charts).

  - Se uma entrada especifica uma sequência $A = a_1 + \cdots + a_n$, então a classe A deverá ter $a_1, \ldots, a_n$ como variáveis.
  - Se uma entrada especifica uma repetição, A é composta de 1 a n a_x's, então A deverá conter uma coleção (Set, List, Array, etc.) da classe a_x.
  - Se uma entrada especifica seleção, A é composto de a_1 ou a_2 ou ... ou a_n, então há uma transformação para um esquema com herança. $a_1, a_2, \ldots, a_n$ podem ser subclasses de um ancestral comum: a classe a_c.

Apêndice C

Sintaxe para Condições

```
prog
  *: regras

regras
   : regra
   | regras regra

regra
   : ACOL ID CTE_NUM pre_condicao ent_saida ent_saida
     pos_condicao ID ID ID FCOL

pre_condicao
   : pre_condicao AND expr_logica
   | pre_condicao OR  expr_logica
   | expr_logica

ent_saida
   : ACOL lista_de_parms FCOL

pos_condicao
   : pos_condicao AND pos_condicao
   | predicado ATRIB constante_logica

expr_logica
   : constante_logica
   | predicado
   | NOT expr_logica
   | APAR pre_condicao FPAR
   | expr_aritm op_relacional expr_aritm

constante_logica
```

```
  : TTRUE
  | TFALSE

predicado
  : PREDID APAR lista_de_parms FPAR

lista_de_parms
   : ID VIRG ID
   | ID

expr_aritm
  : expr_aritm operador fator
  | fator

fator
  : constante
  | variavel
  | funcao
  | APAR expr_aritm FPAR

funcao
  : FUNCID APAR lista_de_parms FPAR

op_relacional
  : IGUAL
  | DIFERENTE
  | MAIOR
  | MENOR
  | MAIOR_IGUAL
  | MENOR_IGUAL

operador
  : MAIS
  | MENOS
  | VEZES
  | DIV

constante
  : CTE_NUM
  | HORA
  | DATA

variavel
  : ID
```

Bibliografia

[Alab88] Alabiso, B.,
Transformation of Data Flow Analysis Models to Object Oriented Design, OOPSLA88 Conf. Proc., pp. 335-353, 1988

[Alen89] Alencar, P.S.C. & Lucena, C.J.P.,
Métodos Formais para o Desenvolvimento de Programas, IV Escuela Brasileño-Argentina de Informática, Kapelusz, 1989

[Aran85] Arango. G., Baxter, I., Leite, J.C. & Pidgeon, C.,
The Domain-based Paradigm and the DRACO Tool: Open Research Problems and Suggested Improvements, 1985

[Aran88] Arango, G.,
Evaluation of DRACO - A Reuse-based Software Construction Technology, Proc. 2nd. IEEE/BCS Conf. on Software Eng., Liverpool - UK, July 1988

[Aste86] Astesiano, E. *et al.*,
Relational Specifications and Observational Semantics, Lecture Notes in Computer Science, vol. 233, pp. 209-217, Springer 1986

[Balz81] Balzer, R.,
Transformational Implementation: An Example, IEEE Transactions on Soft. Engineering, vol. SE-7, no. 1, pp. 3-14, Jan 1981

[Baue80] Bauer, F.L.,
A Trend for the Next 10 Years of Software Engineering, Software Engineering, pp. 1-24, 1980

[Baue85] Bauer, F.L., *et al.*,
The Munich Project CIP, vol. I: The Wide Spectrum Language CIP-L, Lecture Notes in Computer Science, vol. 183, Ed. G.Goos & J.Hartmanis, Springer 1985

[Baue87] Bauer, F.L., *et al.*,
The Munich Project CIP, vol. II: The Program Transformation System CIP-S, Lecture Notes in Computer Science, vol. 292, Ed. G.Goos & J.Hartmanis, Springer 1987

[Berg82] Berg, H.K., *et al.*,
Methods of Program Verification and Specification, Prentice-Hall, 1982

[Bigg84] Biggerstaff, T.J. & Perlis, A.J.,
Foreword, IEEE Transactions on Soft. Engineering, vol.10, no. 5, pp. 474-477, Sept 1984

[Bjor87] Bjorner, D.,
On the Use of Formal Methods in Software Development, ACM Trans. on Comp., pp. 17-29, 1987

[Bjor89] Bjorner, D.,
Towards a Meaning of 'M' in VDM, Lecture Notes in Computer Science, vol. 352, pp. 1-35

[Broy81] Broy, M. & Pepper, P.,
Program Development as a Formal Activity, IEEE Transactions on Soft. Engineering, vol. SE-7, no. 1, pp. 14-22, Jan 1981

[Brun86] Bruns, G.R. & Gerhart, S.L.,
Theories of Design – An Introduction to the Literature, MCC Tech. Rep. STP-068-86, MCC, Texas, 1986

[Burs77] Burstall, R.M. & Darlington, J.,
A Transformation System for Developing Recursive Programs, Journal of the ACM, vol. 24, pp. 44-67, 1977

[Card87] Cardozo, E.,
DPSK: A Kernel for Distributed Problem Solving, Phd Thesis, Carnegie Mellon University, 1987

[Clea88] Cleaveland, J.C.,
Building Application Generators, IEEE, 1988

[Darl78] Darlington, J.,
A Synthesis of Several Sorting Algorithms, Acta Informatica, 11, pp. 1-30, Springer 1978

[DeMa89] DeMarco, T.,
Análise Estruturada e Especificação de Sistemas, Campus, 1989

[Ders83] Dershowitz, N.,
The Evolution of Programs, Birkhauser, 1983

[Dill88] Diller, A.,
Compiling Functional Languages, John Wiley & Sons Ltd., 1988

[Dock89] Docker, T.W.G. & France, R.B.,
Flexibility and Rigour in Structured Analysis, IFIP 89, pp. 89-94, 1989

[Enge88] Engelmore, R.S. *et al.*,
Introduction in Blackboard Systems, Eds. R. Engelmore & T. Morgan, Addison-Wesley Publ. Company, 1988, pp. 1-22

[Leit84] Leite, J.C.,
An Introduction to the Use of DRACO through a Simple Example, July 1984

[Luce87] Lucena, C.J.P.,
Inteligência Artificial e Engenharia de Software, Jorge Zahar Editor Ltda., Rio de Janeiro, 1987

[Naur85] Naur, P.,
Programming as Theory Building, Microprocessing & Microprogramming 15, North-Holland, 1985

[Neig80] Neighbors, J.M.,
Software Construction Using Components, Ph.D. Thesis and Tech. Report 160, University of California at Irvine, ICS Dept., 1980

[Neig84a] Neighbors, J.M., Arango, G. & Leite, J.C.,
DRACO 1.3 Users Manual, Tech. Rep. RTP003.3, University of California at Irvine, ICS Dept., 1984

[Neig84b] Neighbors, J.M.,
The DRACO Approach to Constructing Software from Reusable Components, IEEE Transactions on Soft. Engineering, vol.10, no. 5, pp. 564-574, Sept 1984

[Neig87] Neighbors, J.M.,
DRACO: A Method for Engineering Reusable Software Systems, 1987

[Nii86] Nii, H.P.,
Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, The AI Magazine, vol. 7, no. 2, pp. 38-53, Summer 1986

[Part83] Partsch, H. & Steinbruggen, R.,
Program Transformation Systems, ACM Comp. Surveys, vol. 15, no. 3, Sept 1983

[Peck88] Peckham, J. & Maryanski, F.,
Semantic Data Models, ACM Comp. Surveys, vol.20, no. 3, Sept 1988

[Pryw77] Prywes, N.,
Automatic Generation of Computer Programs, in Advances in Computers, vol. 16, M. Rubinoff and M. Yovits, Eds. New York: Academic, 1977

[Ritt73] Rittel, H.W.J. & Webbe, M.M.,
Dilemmas in a General Theory of Planning, Policy Sciences 4, 1973

[Rote87] Rotenberg, H.B.,
Programação Orientada a Objetos: Um Enfoque de Engenharia de Software, Tese de Mestrado, PUC, Dept. de Informática, Rio de Janeiro, 1987

[Sath85] Sathi, A., Fox, M.S. & Greenberg, M.,
Representation of Activity Knowledge for Project Management, IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. 7, pp. 531-552, Sept 1985

[Fero87] Feroldi, A.,
Evaluation of the DRACO tool, July 1987

[Firt87] Firth, R., *et al.*,
A Classification Scheme for Software Development Methods, Tech. Rep., Software Eng. Institute, Carnegie Mellon University, Nov 1987

[Fox79] Fox, M.,
Organization Structutring – Designing Large Complex Software, Tech. Rep. CMU-CS-79-155, CMU, Pittsburgh, 1979

[Fran89] France, R.B. & Docker, T.W.G,
Formal Specification using Structured Systems Analysis, Lecture Notes in Computer Science, vol. 387, pp. 293-310, Springer 1989

[Free80] Freeman, P.,
The Central Role of Design in Software Engineering: Implications for Research, Software Engineering, pp. 121-132, 1980

[Free87] Freeman, P.,
A Conceptual Analysis of the DRACO Approach to Constructing Software Systems, IEEE Transactions on Soft. Engineering, 1987

[Frie92] Friedman, L.W.,
From Babbage to Babel and Beyond: A Brief History of Programming Languages, Computer Lang., vol. 17, no. 1, 1-17, 1992

[Gane84] Gane, C. & Sarson, T.,
Análise Estruturada de Sistemas, LTC, 1984

[Gogu84] Goguen, J.A.,
Parameterized Programming. IEEE Transactions on Soft. Engineering, vol.10, no. 5, pp. 528-551. Sept 1984

[Haeb89] Haeberer, A.M., Veloso. P.A.S. & Baum, G.,
Formalizacion del Proceso de Desarrollo de Software, IV Escuela Brasileño-Argentina de Informática, Kapelusz, 1989

[Jack75] Jackson, M.,
Principles of Program Design, London, Academic Press, 1975

[Lehm84a] Lehman, M.M.,
A Further Model of Coherent Programming Processes, IEEE Proc. of Software Process Workshop, UK, Feb 1984, IEEE Comp. Soc., 1984

[Lehm84b] Lehman, M.M., Stenning, V. & Turski, W.M.,
Another Look at Software Design Methodology, ACM SIGSoft Soft. Eng. Notes, vol. 9, no. 2, pp. 38-53, Apr 1984

[Simo73] Simon, H.A.,
The Structure of Ill-structured Problems, AI 4, 1973

[SPW84] Potts C (ed),
Proceeding of the Software Process Workshop, Egham, Surrey, U.K., Feb. 1984, Publ. IEEE

[Stei89] Steier, D.M. & Anderson, A.P.,
Algorithm Synthesis: A Comparative Study, Springer- Verlag, 1989

[Smit77] Smith, J.M. & Smith, D.C.P.,
Database Abstractions - Aggregations and Generalizations, ACM Trans. on Database Systems, vol. 2, pp. 105-133, 1977

[SE80] *Software Engineering*,
Ed. H.Freeman & P.M. Lewis II, Academic Press, 1980

[Taka90] Takahashi, E.T.,
Basic Integration Mechanisms in the Architecture of an ETHOS environment - Directions for Research, Memorando Interno, Projeto ETHOS, 1990

[Taka92] Takahashi, E.T.,
Notas de Discussões Informais, 1992

[Tse89] Tse, T.H., Pong, L.,
Towards a Formal Foundation for DeMarco Data Flow Diagrams, The Comp. Journal, vol.32, no.1, 1989

[Turs87] Turski, W.M. & Maibaum, T.S.E.,
The Specification of Computer Programs, Addison-Wesley Publ. Company, 1987

[Webs87] Webster, D.E.,
Mapping the Design Representation Terrain - A Survey, Tech. Rep. STP-093-87 MCC, 1987

[Wing90] Wing, J.M.,
A Specifier's Introduction to Formal Methods, Computer, pp. 8-24, 1990

[Wino86] Winograd, T. & Flores, I.,
Understanding Computers and Cognition - A New Foundation for Design, 1986

[Wirt71] Wirth, N.,
Program Development by Stepwise Refinement, Comm. of the ACM, vol. 14, no. 4, pp. 221-227, Apr 1971

[Your79] Yourdon, E. & Constantine, L.L.,
Structured Design, Prentice-Hall, 1979