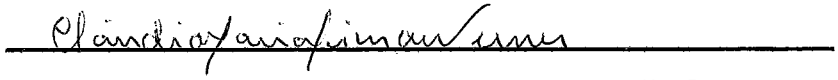
Uma Ferramenta de Reúso por Interconexão de Linguagens

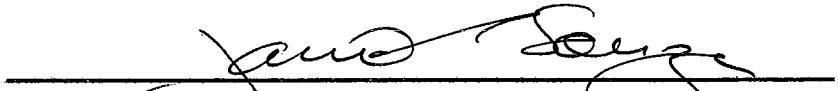
Flavio Araujo de Mattos

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

Aprovada por:



Profa. Cláudia Maria Lima Werner, D. Sc.



Prof. Jano Moreira de Souza, Ph.D.

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

RIO DE JANEIRO, RJ - BRASIL

ABRIL DE 1994

MATTOS, FLAVIO ARAUJO DE

Uma Ferramenta de Reúso por Interconexão de Linguagens [Rio de Janeiro] 1994.

ix,128 p. 29,7 cm (COPPE/UPRJ,M. Sc., Engenharia de Sistemas e Computação, 1994)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Reutilização de Software

I. COPPE/UFRJ II. Título (série)

À minha amada Ana

AGRADECIMENTOS

À minha orientadora Cláudia Werner pelo zelo, paciência, incentivo e orientação sem os quais este trabalho não se concretizaria.

Ao Jano pelo apoio, pelos conselhos e pelo conhecimento que nunca se negou a compartilhar.

Ao professor Júlio Leite, por integrar a banca examinadora.

Ao Guilherme pela força nos meus primeiros passos na orientação a objetos.

Aos funcionários do Programa de Sistemas e, em particular, à secretaria do programa, que deu apoio contínuo e indispensável.

Ao Jorge e ao Alexandre pelo apoio tático quando meu computador quebrou.

Aos amigos do BNDES pela solidariedade e companheirismo.

Ao Ruben, verdadeiro irmão, pela paciência e pelo ombro amigo nos momentos mais difíceis.

Aos meus pais pelo carinho e pelas orações, sempre presentes.

Ao meu Deus, por tudo isto, e pelo que mais há.

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

UMA FERRAMENTA DE REÚSO POR INTERCONEXÃO DE LINGUAGENS

Flavio Araujo de Mattos

ABRIL, 1995

Orientadora: Profa. Cláudia Maria Lima Werner, D. Sc.

Programa: Engenharia de Sistemas e Computação

Esta tese apresenta uma ferramenta para a reutilização de software orientada a domínios. A ferramenta permite organizar hierarquicamente domínios de aplicação na forma de sistemas lingüísticos. As linguagens especializadas em domínios são inter-conectadas através de técnicas compositivas da orientação a objetos

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

A TOOL FOR REUSE BY LANGUAGES INTERCONNECTION

Flavio Araujo de Mattos

APRIL, 1995

Thesis Supervisor: Prof. Cláudia Maria Lima Werner, D. Sc.

Department: Systems and Computation Engineering

This thesis presents a tool to provide domain oriented software reuse. This tool allows hierarchical organizations of domains by building linguistic systems. The domain specialized languages are interconnected using object-oriented composing techniques.

Sumário

1. Introdução

A reutilização de software, mais que uma nova linha de pesquisa dentro da engenharia de software, é uma necessidade principal dada a crescente demanda por sistemas maiores e mais complexos. Traduzida em termos de limitações de custo, tempo, qualificação e disponibilidade de recursos humanos, esta necessidade transcende as fronteiras da reutilização de código objeto e atinge o processo de desenvolvimento em sua totalidade. Tal dimensão impediu uma visão uniforme da reutilização, dada a intrínseca heterogeneidade do processo de desenvolvimento de software nas suas diversas abordagens.

O grande desafio das propostas da amplo emprego da reutilização é abranger a enorme diversidade de formas e tecnologias através das quais o software se manifesta. Mesmo se considerássemos apenas os produtos executáveis, há centenas de linguagens de programação trabalhando em universos conceituais diferentes. Além disto, a possibilidade de mudança de paradigma de desenvolvimento nas próximas décadas trás uma nova dimensão à complexidade do reúso. Uma solução de reúso a médio e longo prazo deve, ao mesmo tempo, dar apoio às principais linhas tecnológicas hoje adotadas e ser extensível para capturar as novas que virão. [Matt93]

Estas preocupações foram inicialmente levantadas no decorrer do desenvolvimento do protótipo do CAOS (Composing Application-Objects System) [Wern91][Wern92a][Wern92b][Wern93]. O CAOS surgiu com o intento de suportar o desenvolvimento no contexto científico, substituindo as atuais bibliotecas científicas por um sistema mais completo, capaz de dar apoio, não somente às atividades de manipulação de

biblioteca, mas também às demais atividades envolvidas na composição de aplicações (i.e., entendimento, modificação e composição de objetos de software). O ambiente CAOS utiliza conceitos e técnicas da orientação a objetos, explorando suas facilidades básicas para o desenvolvimento de aplicações baseado na reutilização de componentes de software. O ambiente se vale ainda de um mecanismo de classificação facetada de componentes reutilizáveis para a seleção de candidatos à reutilização, em uma variante da proposta de Prieto-Díaz [Prie85][Prie89], em que a similaridade entre componentes e a atividade de compreensão para o reúso são suportadas através da tecnologia de hipertexto [Conk87].

A orientação a objetos, além de oferecer mecanismos específicos úteis ao reúso (como a herança e o polimorfismo), traz a seu favor o poder de conciliar tecnologias de diversas áreas a que o paradigma já está integrado e frutificando, tais como linguagens de programação, bancos de dados orientados a objetos, métodos de análise e projeto etc. Este caráter interdisciplinar é fundamental para a pesquisa em uma área de tão amplo espectro como a reutilização. [Bigg87] , contudo, atribui à orientação a objetos uma posição desfavorável em relação a outras tecnologias dentro de um quadro em que a generalidade e o poder de expressão concisa são avaliados subjetivamente. (figura 1.1). O amadurecimento das tecnologias orientadas a objetos proporcionou um aumento significativo de seu poder de expressão desde então. Hoje não podemos colocá-lo como inferior ao das bibliotecas.

No ambiente CAOS [Wern91] [Wern92a] [Wern92b] [Wern93], por exemplo, o módulo acoplador de bibliotecas FORTRAN transforma cada sub-rotina de uma biblioteca em um método de um objeto. Em um nível mais sofisticado de acoplamento, os conceitos do domínio atendido pela biblioteca são representados em um modelo de objetos,

que oculta completamente as chamadas ao modelo original. Desta forma, o usuário dispõe do mesmo poder de computação através de uma visão mais próxima de sua realidade de trabalho. Este modelo de objetos pode ainda ser modificado através da linguagem de programação CAOL, que permite a inclusão de funcionalidade complementar, de modo a facilitar a reutilização de suas partes.

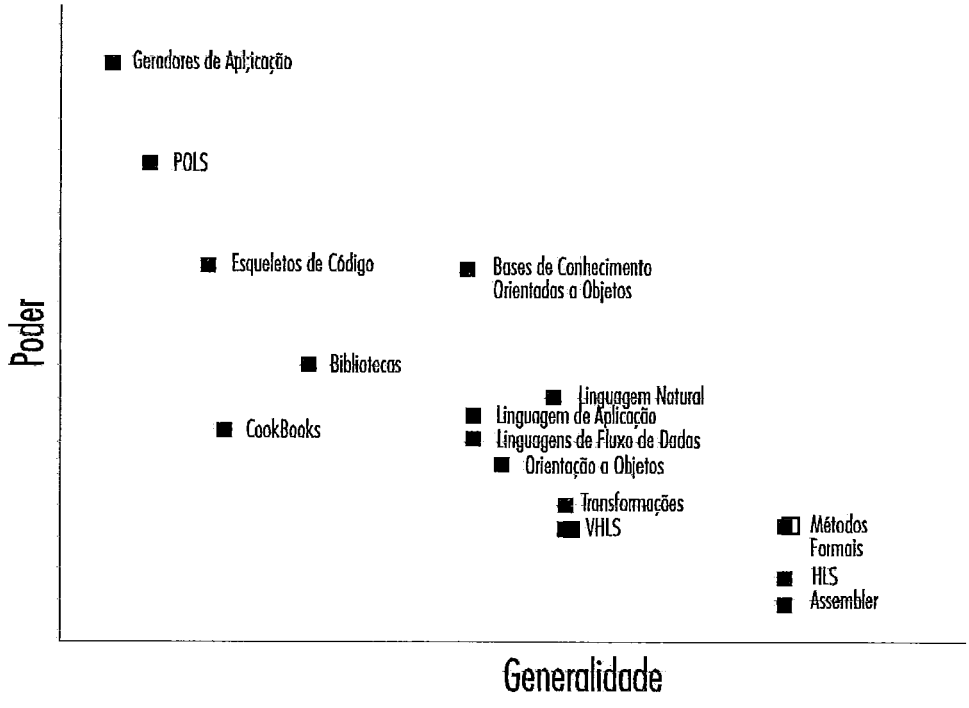


Figura 1.1 - *Caracterização das tecnologias para reúso [Bigg87]*

Se o poder de representação da orientação a objetos supera o das bibliotecas, ele, no mínimo, se equivale ao dos esqueletos de código. Sistemas baseados em esqueletos de código, como o Paris [Katz87], provêm o reúso através de código incompleto, isto é, código onde trechos específicos podem ser livremente substituídos para atender a uma necessidade particular de reutilização. A funcionalidade do software construído desta forma é uma integração da funcionalidade do esqueleto à das partes

substituídas. A orientação a objetos oferece, através do mecanismo de troca de mensagens e do polimorfismo, a possibilidade de construir trechos de código genéricos em um sentido muito similar ao dos esqueletos. O código, especificado em termos de envio de mensagens a objetos, não detém a definição completa da funcionalidade, que dependerá da implementação particular dos objetos com os quais será usado. A figura 1.2 ilustra um fragmento de código de uma linguagem de programação que pode ordenar todo tipo de coleção de objetos que se conforme a um padrão de interface.

Esta evolução de poder de expressividade (que é o responsável pela sua generalidade) não se deu por mudanças significativas no paradigma da orientação a objetos. Foi uma processo de exploração dos potenciais da abordagem através de sua aplicação sistemática a diversas áreas e de acúmulo de conhecimento, em um processo que ainda não cessou.

A investigação da integração das tecnologias orientadas a objetos e de tecnologias de geração é interessante porque, na escala de Biggerstaff, nas tecnologias baseadas em geração, encontramos o maior poder de expressividade.

Em [Wern92a], Werner propôs a continuidade de sua pesquisa em reutilização pela elaboração de um sistema híbrido, que combinasse fortemente aspectos gerativos e compositivos. Tal abordagem híbrida permitiria unir o alto nível de abstração e facilidade de uso das tecnologias de geração com a flexibilidade e manutenibilidade da estrutura de tecnologias de composição. As tecnologias de composição cumpririam, nesta solução, o papel de intermediar o refinamento da especificação de alto nível até um sistema derivado dela, especificando partes reutilizadas no processo de geração. Ao usar a

orientação a objetos neste papel intermediário, ganhamos dela a capacidade de construção de estruturas bastante complexas (objetos complexos) subordinadas a arquiteturas bem definidas.

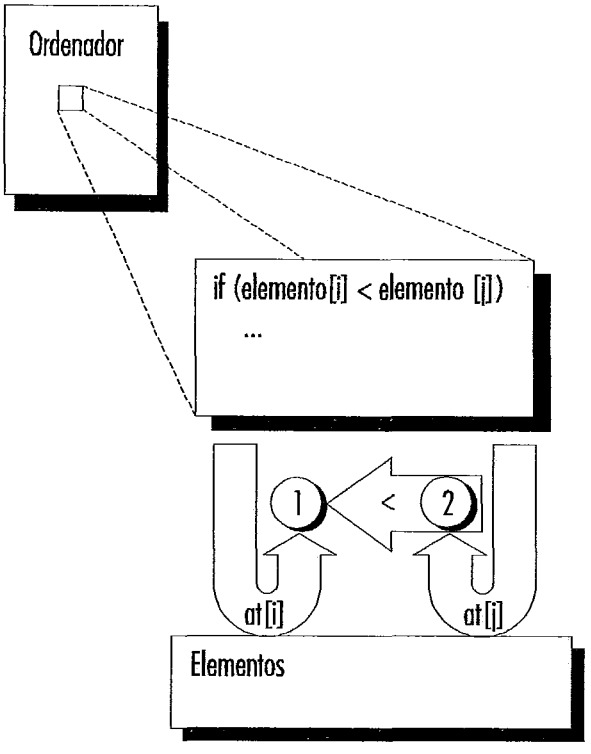


Figura 1.2 - Padrões de Interface - *fragmento de código de ordenação, escrito em linguagem Actor [Whit91], envia duas mensagens "At" ao objeto que contém os elementos a ordenar. A mensagem "<" é enviada ao objeto resposta "1" para que seja comparado com a resposta "2". O resultado da comparação é usado na ordenação. Em nenhum momento a natureza dos objetos envolvidos é considerada.*

O objetivo desta tese é desenvolver uma abordagem de reutilização híbrida que permita a construção de linguagens de alto nível de abstração especificadas através de

técnicas oriundas da orientação a objetos. O ponto central da abordagem é a nova leitura do processo de desenvolvimento de software como uma evolução de conceitos abstratos dirigida por linguagem, em analogia aos processos intelectuais em que a linguagem exerce o papel de ordenador do pensamento.

A tese está dividida em seis capítulos. No primeiro deles definimos o objetivo do presente trabalho, sua motivação e organização.

No capítulo 2, introduzimos a questão da reutilização de software. Fazemos uma revisão da literatura sobre os aspectos fundamentais do tema e debatemos as linhas tecnológicas de *reúso através de geração* e *reúso através de composição*, bem como o papel da linguagem no reúso. No capítulo 3, apresentamos os conceitos fundamentais da orientação a objetos e o seu papel no reúso.

Os capítulos 4 e 5, respectivamente, apresentam uma proposta para um ambiente de reúso baseado em linguagens voltadas para domínios de aplicação e a arquitetura do ambiente Gremlin, construído com base nesta proposta.

Finalmente, no capítulo 6, apresentamos as principais contribuições do trabalho indicando suas perspectivas de continuidade.

2. Reutilização

2.1. O porquê da palavra

> ***Reutilização*** *: Procedimento em que material que já fora anteriormente processado se insere, após o tratamento conveniente, numa corrente de processo. [Aure94]*

Não importa de que maneira seja revestido, o software é em última instância o fruto da imaginação criadora de um grupo de pessoas [Bigg89]. Um fruto, todavia, que possui valor comercial, que é componente industrial e é largamente usado em nossa sociedade, materializado através de linguagens de programação em seqüências de instruções para equipamentos programáveis específicos. Neste limiar entre a abstração intelectual e a realização material, está a raiz do problema de ausência de reúso.

O desenvolvimento de software é muito mais que a elaboração de programas. Antes e depois desta etapa há um grande esforço intelectual despendido em atividades organizadoras distribuídas ao longo do ciclo de vida de um software [Ghez91]. Durante estas atividades, o engenheiro de software constrói novas abstrações e trabalha com as de que já dispõe por experiência ou educação. A ausência de reúso é percebida quando é observado que, enquanto a experiência prévia de um indivíduo é integralmente reaplicada em problemas similares, os frutos desta experiência não o são. A título de ilustração, um programador que alguma vez tenha desenvolvido uma tabela em memória organizada em

árvore-B, aplicará sua experiência para desenvolver uma árvore-B+ em um sistema de arquivos. Os programas desenvolvidos para uma e outra situação podem, no entanto, assumir formas bastante diversas entre si. Cedo percebemos que, com a tecnologia adequada, os programas poderiam ser muito similares, de forma a minimizar o esforço do segundo desenvolvimento.

Diante da crise de software, a perspectiva de aumento de produtividade pela capacidade de aproveitar produtos de esforços anteriores delineia um horizonte onde a produção de software assuma feições de produção industrial, com benefícios de qualidade e produtividade. Enquanto o desenvolvimento se dá em pequena escala, o âmbito do problema é meramente tecnológico. Quando a experiência a reutilizar transcende a de uma pessoa ou equipe fixa, e pretendemos exercer a reutilização de modo intensivo e por um período arbitrariamente grande de tempo, temos caracterizado o reúso em larga escala. Neste caso, somente a aplicação de técnicas e disciplinas específicas permite avaliar com segurança se algum trabalho pode ser reaproveitado com esforço menor ao do desenvolvimento a partir do zero. A viabilidade da solução de reúso em larga escala envolve mudanças no modo atualmente empregado para a construção de software. A busca das soluções que viabilizem o reúso é o alvo da pesquisa que Freeman denomina Engenharia de Software Reutilizável (ESR) [Free87a].

2.2. O que é Reúso de Software

> ***Reúso de Software:*** *"É o processo de criar software a partir de software existente ao invés de construir sistemas de software do nada." [Krue92]*

Ao contrário do que ocorre com o vocabulário da orientação a objetos (vide capítulo 3), os sinônimos reúso e reutilização são amplamente usados com sentidos muito próximos, sempre com a finalidade de reaproveitar software existente no desenvolvimento de novos. As pequenas variações encontradas estão quase que exclusivamente restritas à abrangência do que é reutilizado e ao grau de formalismo necessário para a sua prática. Peter Freeman define reúso como "qualquer procedimento que produza (ou ajude a produzir) um sistema pela reutilização de algo (uso de algo novamente) de um desenvolvimento prévio" [Free87a]. Esta definição engloba o processo de produção de software em sua totalidade, e força-nos a reconhecer que, ao longo da história da ciência da computação, sempre houve algum nível de reúso [Trac88b], mesmo que nisto não houvesse intenção explícita. Estes mecanismos de reúso circunstancial se contrapõem ao reúso fruto de um esforço intencional, fundamentado em bases formais, com vistas à melhoria do processo de desenvolvimento de software [Bigg87]. O objetivo da ESR é o desenvolvimento e difusão das técnicas de reúso formal.

Em nosso trabalho adotamos uma visão mais ampla do que é software, incluindo em um sistema o conhecimento usado para desenvolvê-lo. Nesta visão, não há desenvolvimento sem reúso, mas há a distinção de reúso informal, que é totalmente subjetivo em qualidade e produtividade, e do reúso formal, que repousa sobre instrumentos de apoio e cuja aplicação é desejável em larga escala.

O reúso formal vem sendo amplamente descrito como uma área onde o sucesso é a exceção e não a regra [Bigg87],[Parn83],[Krue92]. As causas arroladas para este insucesso são descritas em três planos: o plano pessoal, o plano estratégico e o plano

técnico. Os fatores do plano pessoal, como a síndrome NIH ("Not Invented Here") [Trac88a], [Bigg87], tratam da relutância individual em fazer com que o reúso substitua a criação (ou antes, a recriação) de software. Os fatores estratégicos tratam dos obstáculos políticos, econômicos e gerenciais para a adoção de uma abordagem intensiva de reúso. No plano técnico, as causas arroladas são a inexistência de tecnologia adequada para o reúso em larga escala [Meye87] ou a interpretação incorreta do reúso enquanto área de estudo [Bigg89] [Free87a]. A percepção de que o reúso formal não é largamente empregado não suscita a interpretação de que esta técnica seja de algum modo indesejável ou inadaptável às necessidades. Antes, serve de estímulo a novas experiências e à produção de tecnologia para transformar o reúso informal em reúso formal.

O reúso formal caracteriza-se, primordialmente, pela intenção de reutilizar sistematicamente. Esta intenção pode ser traduzida no cuidado a cada passo da evolução de um sistema de software, aproveitando ao máximo as experiências prévias similares. Para que o aproveitamento seja máximo, a experiência prévia deve estar de algum modo codificada em diversos níveis de abstração para reúso futuro [Prie89]. Para reúso formal adotaremos uma definição extraída de [Trac90]:

Reúso formal: *Processo de reúso de software criado para ser reutilizado.*

2.3. Origens e Evolução do Reúso Formal

As origens do reúso formal estão intimamente ligadas à pesquisa de engenharia de software em busca do aumento da produtividade e da qualidade do processo de desenvolvimento. Coube a McIlroy o pioneirismo ao antever o potencial da reutilização de código no desenvolvimento em larga escala [McIl68]. Sua proposta consistia no desenvolvimento centrado em rotinas reutilizáveis de alta qualidade organizadas em catálogos, de modo a oferecer uma abordagem do reúso de software similar à encontrada no design de hardware. A ênfase de McIlroy na qualidade dos componentes e na sua aplicabilidade ao desenvolvimento de software em larga escala lançou dois alicerces do reúso tal como é visto hoje. Em primeiro lugar, a qualidade é um ponto essencial para o reúso. A deficiência de um componente será propagada em todas as ocasiões em que for reutilizado. De modo inverso, uma base de componentes de alta qualidade influi positivamente na qualidade do software construído a partir dela.

O segundo aspecto é o reconhecimento de que o reúso é de grande valia ao desenvolvimento em larga escala de diversas maneiras. O aumento da produtividade pela redução do retrabalho é apenas o argumento mais evidente desta contribuição. A disponibilidade de extensas bases de software reutilizável, por exemplo, permite a confrontação de alternativas de design em bases de custo/benefício completamente diversas. O uso de um componente complexo pode viabilizar uma solução que, diante das restrições operacionais do desenvolvimento de um sistema específico, seria economicamente inviável.

Àquela época McIlroy foi questionado quanto a viabilidade de sua abordagem em três linhas principais:

1. seria impossível contornar aspectos de hardware como a representação interna dos dados;
2. não existia nenhum conhecimento acumulado sobre classificação de software em catálogos que permitisse a alguém localizar uma rotina desejada;
3. seria impossível conseguir rotinas paramétricas, eficientes e robustas gerais o suficiente para tornar conveniente o reúso em todos os sistemas. [Horo84]

Enquanto o primeiro ponto foi sensivelmente reduzido por padrões, linguagens de programação e sistemas operacionais, os dois últimos deram origem a áreas ativas dentro da pesquisa de reúso.

O estudo de mecanismos de classificação e busca em grandes bases de reúso é objeto de diversas linhas de pesquisas [Prie85] [Sim90] [Xav90] [Bell92] [Wern92a] [Wern92b] .

O estudo da parametrização de programas encontrou suporte no âmbito das linguagens de programação. Os mecanismos de generalidade e de polimorfismo ampliaram os horizontes de construção de código paramétrico [Blai89]. A generalidade é uma característica encontrada em linguagens como ADA[Booc83] e C++ [Stro93], que permite que um trecho de código seja especificado, deixando alguns tipos indefinidos em

suas variáveis. Quando o código é empregado, estes tipos abstratos são oferecidos para permitir a conclusão da verificação dos tipos. O polimorfismo é uma característica típica da orientação a objetos, baseada na ligação dinâmica entre a chamada de uma rotina e sua implementação (ver seção 3.3.1).

A preocupação com a parametrização dos componentes é, todavia, a semente mais remota de uma vasta diversidade de pesquisa quanto ao reúso. A busca da parametrização é, na realidade, a busca por componentes de aplicação genérica, cujos empregos particulares sejam especificados tardiamente. Foi da percepção de que o software é menos um produto estático e mais o efeito de um processo iterativo, que levou tal preocupação para dentro do ciclo de vida do software. Informações de análise e design são, hoje, objeto principal da pesquisa em reutilização. As informações deste nível possuem a generalidade desejada. Hoje, importa como organizá-las de modo a facilitar seu reúso, tanto no nível conceitual quanto nos mecanismos de sua implementação.

Peter Freeman classifica as informações reutilizáveis em cinco níveis, onde percebe-se uma distinção clara entre informações reutilizáveis internas e externas ao software [Free87a] .

Informações reutilizáveis internas são aquelas que possuem um representante computacional, a saber: fragmentos de código; estruturas lógicas (representadas por arquiteturas de software); arquiteturas funcionais (representadas por coleções funcionais e sistemas genéricos). Fragmentos de código são quaisquer trechos de programas passíveis de reúso. Arquitetura de software é uma organização de módulos e seus relacionamentos, de modo a formar o design interno de um sistema. Coleções funcionais são

conjuntos de funções agregadas em uma unidade independente que cobrem significativamente uma área do conhecimento. Os sistemas genéricos integram as funções de uma coleção em uma arquitetura tal, que o software pode ser empregado em um grande número de situações sem alteração.

Informações reutilizáveis externas ao software são diversas formas de conhecimento e dizem respeito ao processo de desenvolvimento, à área de aplicação, à difusão da tecnologia e ao conhecimento sobre o papel dos sistemas dentro da área de aplicação (estes dois últimos Freeman coloca em um nível ainda mais alto, chamado nível ambiental).

Hoje a pesquisa de reutilização está se deslocando do nível interno para o externo. O enfoque deixa, progressivamente, de estar na construção de bibliotecas de componentes em suas diversas formas se dirigindo para bases que armazenem conhecimento sobre o domínio de aplicação [Neig94]

2.4. Qual é a distinção real entre composição e geração?

Segundo Biggerstaff e Richter [Bigg87] as tecnologias para reutilização estão divididas segundo dois princípios fundamentais de reutilização: composição e geração. Eles ensejam, respectivamente, o que chamamos de reutilização compositiva e reutilização gerativa. O princípio da composição é caracterizado pela criação de software através do reúso de componentes "amplamente atômicos e idealmente imutáveis no curso de seu reúso"[Bigg87]. O princípio da geração baseia-se no reúso de seqüências de instruções e

seqüências de transformações embutidas em um gerador, sendo possível que o produto gerado seja bem diferente das seqüências que lhe deram origem.

São exemplos de reutilização compositiva as tecnologias baseadas em bibliotecas de sub-rotinas e orientação a objetos. Sob a reutilização gerativa estão as linguagens de altíssimo nível, os geradores de aplicação e os sistemas baseados em transformação.

O amadurecimento das tecnologias compositivas deixou evidentes dois novos aspectos que forçam uma redefinição desta categoria de solução:

- componentes não são atômicos;
- componentes são transformados durante a composição.

A atomicidade dos componentes, hoje, não se apresenta nem na sua organização estrutural, nem na sua caracterização conceitual. O reúso em larga escala exige a construção de novos componentes reutilizáveis para atender à demanda de novas áreas de aplicação. Uma porção significativa destes componentes é construída pela integração de componentes preexistentes. Mais que isto, sua evolução no tempo (manutenção) é afetada pela evolução destes componentes constituintes, estabelecendo um vínculo que não pode ser sempre abstraído em prol do conceito da atomicidade. Quanto à caracterização conceitual, um componente dificilmente é atômico. A compreensão necessária para o reúso [Prie89]

exige freqüentemente um estudo do contexto conceitual em que se situa. Um exemplo de sistema que dá apoio a este tipo de estudo é [Wern93].

A transformação é vista como uma característica importante para a reutilização. No mesmo artigo [Bigg87], Biggerstaff cita que a "transformação é a vida da reutilização" . A transformação de componentes também assume muitos aspectos diversos. A título de exemplo, [Deut83] propõe para o reúso em Smalltalk-80, a criação de "frameworks" de classes abstratas que representem protocolos de interação dentro de uma arquitetura de componentes. Classes abstratas são classes de objetos que não estão completamente definidas, de modo que são apenas parcialmente operacionais. O reúso nesta proposta se dá pela transformação destas classes através do mecanismo de herança, completando sua definição.

Um ponto aberto nesta discussão são as reais razões para a desejável estabilidade dos componentes de software. Operar uma transformação em um componente implica custo, e a estabilidade é sinônimo de economia. Os componentes revestidos de estabilidade e versatilidade são fortes candidatos ao reúso com qualidade. Por outro lado, esta estabilidade pode ser uma qualidade de natureza estritamente tecnológica, e não uma propriedade essencial do problema de reutilização. O ponto pode ser explicado por uma analogia. O reúso de componentes estáveis é análogo à construção de artefatos a partir de peças prontas, uma postura largamente usada em termos industriais. A perspectiva adequada ao reúso talvez seja outra. Quando estabelecemos analogias entre as componentes de software e peças reais, podemos levantar dúvidas quanto ao caráter essencial da estabilidade da peça no reúso. O problema do reúso de peças de um automóvel pode ser descrito em

"como aproveitar a experiência adquirida no projeto de uma peça para projetar outra peça". Neste sistema concreto a estabilidade das peças é propriedade desejável enquanto proporcione redução dos custos de produção ou preserve investimentos anteriormente feitos nas linhas de montagem. A estabilidade, todavia, não como propriedade essencial neste problema particular.

Como vimos, mesmo quando a estabilidade é desejada, um componente de software pode ser modificado para atender a uma composição específica. Além disto, a visão mais ampla do conceito de software estabelecida na seção 2.2 faz perceber que um componente é transformado pelas suas reutilizações, que afetam diretamente algumas de suas características, como o custo de manutenção, por exemplo. Em poucas palavras a reutilização compositiva transforma os componentes envolvidos pela alteração de sua finalidade, que deixa de ser um potencial geral, para cumprir um papel específico dentro do contexto em que é reutilizado.

Hoje as definições para reúso compositivo e gerativo devem ser muito mais próximas, ambas comportando transformações, diferindo apenas quanto à natureza destas e o enfoque que lhes é dado.

> ***Reutilização Gerativa:*** *Tecnologia de reutilização de software em que um sistema apoiado por computador transforma uma especificação em alto nível, textual ou não, baseada em um número finito e bem definido de*

abstrações, em uma especificação equivalente expressa em uma linguagem de nível mais baixo, textual ou não, obtida pela tradução das abstrações de uma para outra linguagem, bem como pela inserção do software necessário para explicitar quaisquer significados ou ações implícitos na linguagem de alto nível e não na de nível mais baixo. O software reutilizado é o que define a semântica implícita e o que dá suporte às abstrações de alto nível.

Reutilização Compositiva: *Tecnologia de reutilização de software em que o desenvolvimento se dá a partir da integração de entidades de software pré-projetadas, escolhidas dentro de um conjunto arbitrariamente grande e extensível. O processo de composição é uma transformação de software que pode ou não preservar a integridade original dos componentes. O software reutilizado é o conjunto de componentes. Idealmente as transformações devem ser geridas por mecanismos formais que mantenham sob controle a propagação dos efeitos das mudanças em componentes usados por muitas aplicações.*

2.5. A importância da linguagem na reutilização

Os produtos materiais do desenvolvimento de software (como programas, documentos e diagramas) são reflexos da atividade intelectual que os produziu. Pensar que o reúso pode ater-se a este material produzido é incorrer no erro de perder sua essência

intelectual. É dissociar forma e conteúdo. A dinâmica da reutilização em qualquer de suas formas sempre recai em, a partir da representação, apreender os conceitos abstratos que ela representa e dele derivar uma nova forma que nos interessa. Cabe à tecnologia facilitar esta transformação, de modo que o que foi matéria prima para a mente, também o seja para a representação.

Uma das mais importantes linhas de pesquisa para a reutilização é o desenvolvimento de linguagens de muito alto nível que propiciem uma forma de expressão intelectualmente natural dentro de um domínio específico do conhecimento. Esta abordagem possui forte respaldo na ciência da cognição, onde pensadores como Vigotsky apresentam o pensamento como uma processo fortemente guiado pela linguagem. "A formação de conceitos (...) é dirigida pelo uso de palavras como meio de centrar ativamente a atenção, abstrair determinados traços, sintetizá-los e simbolizá-los por meio de um signo" [Vigo87]. Estas linguagens de altíssimo nível expressam conhecimento como um padrão de atribuição de significados a suas sentenças.

2.5.1. DRACO, um sistema de reúso em larga escala baseado em linguagens

Segundo Neighbors [Neig84] [Neig89], as tecnologias transformacionais podem ser classificadas quanto à abrangência com que as linguagens de especificação cobrem o ciclo de vida do desenvolvimento. Em um extremo, as linguagens que contemplam apenas um aspecto ou atividade do desenvolvimento são chamadas linguagens de espectro estreito. No outro, aquelas que pretendem cobrir todas as fases do desenvolvimento de um sistema são chamadas de amplo espectro. Sua proposta DRACO criou um novo padrão para

sistemas transformacionais por incorporar, operacionalmente, o fato bem estabelecido de que o desenvolvimento de software apresenta características distintas em função da área a que se destina. Neighbors introduziu a atividade de **análise de domínio,** através da qual o conhecimento sobre a construção de software em uma determinada área é organizado. A abordagem DRACO prevê que cada domínio da aplicação é representado por uma linguagem específica que serve como organizador de objetos e operações modelados a partir do domínio. A linguagem de domínio passa a ser exercida no desenvolvimento de sistemas para aquela área, oferecendo o que Neighbors caracteriza como **reúso de análise**.

A linguagem de domínio é usada para a construção de um ambiente operacional chamado de **representação de domínio,** que implementa processos específicos de tradução desta linguagem e oferece uma série de mecanismos de refinamento que permitem que uma especificação seja conduzida passo a passo até o código executável.

Em [Neig84], a representação de um domínio de aplicação é caracterizada por:

1.**analisador sintático** que define a sintaxe externa da linguagem, originalmente apenas para linguagens LL(1), e implementa um mecanismo de tradução, cujo produto final é uma árvore sintática dotada de atributos passíveis de manipulação pela ferramenta DRACO;

2.**formatador** que faz o caminho inverso, sintetizando o aspecto externo de uma árvore sintática, oferecendo uma apresentação compreensível de sua estrutura;

3.**transformações** que são mecanismos através dos quais um código fonte, em uma linguagem, pode ser transformado em um outro na mesma linguagem que lhe seja equivalente;

4.**componentes** que são os representantes dos conceitos e operações do domínio de aplicação, além de consistirem o instrumento para o refinamento das especificações (como descrevemos adiante);

5.**procedimentos** que são instrumentos que permitem que o refinamento de uma especificação se dê através de um algoritmo, o que apenas é possível onde o conhecimento estabelecido seja suficiente para suprir todas as decisões de design automaticamente.

As transformações permitem criar uma especificação a partir de uma outra equivalente cuja forma seja mais adequada a um refinamento em particular. Um exemplo de transformação que poderia ser aplicado em linguagem **C** é a que substitui textos do tipo **ID*0** por **0**. As transformações são altamente dependentes da linguagem (a transformação anterior, por exemplo, não é válida para **C++**, onde o operador de multiplicação pode ser redefinido para cada tipo nativo ou definido pelo usuário) e, geralmente, são bastante numerosas. Neighbors propõe um mecanismo que auxilie o usuário do DRACO na escolha de transformações úteis em uma determinada situação [Neig84].

Os componentes são o elemento chave do processo de desenvolvimento através do DRACO, por representarem as diversas alternativas de design para a implementação de um objeto ou operação do domínio de aplicação. DRACO exige que tais

implementações sejam descritas em linguagens de outros domínios, de modo que os componentes definem um relacionamento hierárquico entre os domínios de aplicação.

Um passo de refinamento de uma especificação se dá pela escolha das alternativas de refinamento dos componentes que a integram. Este passo incorpora na especificação trechos de linguagens de outros domínios, que serão sucessivamente refinados até atingirem um nível final, onde procedimentos o conduzem a código executável.

Cada uma das alternativas de refinamento possui condições que devem ser atendidas para sua escolha, e assertivas em que sua escolha implicam. Ao longo do processo de refinamento, assertivas e condições são verificadas para assegurar a validade do processo de refinamento.

A hierarquia de domínios do DRACO caracteriza um grafo orientado acíclico, de modo que o refinamento seja sempre orientado de domínios de níveis mais alto de abstração para os de nível inferior. Os domínios que ficam à base desta hierarquia são chamados **domínios executáveis**, porque descreve uma linguagem alvo. As especificações construídas neste domínio podem ser traduzidas por um compilador convencional e executadas. Os demais domínios dividem-se em **domínios de modelagem** e **domínios de aplicação**. Enquanto os domínios de aplicação são efetivamente especializados na construção de sistemas em uma área fim, os domínios de modelagem oferecem objetos e operações de infra-estrutura aos domínios de aplicação. Os domínios de modelagem são claramente identificados por dizerem respeito a conceitos da ciência da computação,

enquanto os domínios de aplicação representam conhecimento de áreas finais, não relacionadas com a ciência da computação[Leit94] .

Em [Neig89], a caracterização da representação de um domínio foi apresentada em uma outra forma. Transformações e processamentos foram integrados em uma nova categoria chamada otimizações, que passa a comportar também "scripts" estruturados de procedimentos e transformações. No trabalho de engenharia reversa DRACO-PUC [Leit92], procedimentos e componentes deram lugar uma nova categoria chamada refinamento, que agrega decisões de design que envolvem troca de nível lingüístico.

A necessidade de organização dos domínios de aplicação levou ao surgimento de novos agentes no processo de desenvolvimento de software. Os **analistas de domínio** organizam a informação de um domínio, estabelecendo em que bases conceituais e lingüísticas o reúso se dará. A partir destas especificações os **designers de domínios** organizam os componentes deste domínio, desenvolvendo as alternativas de refinamento válidas para cada um deles. O papel dos analistas de sistemas também sofre alteração. Na abordagem DRACO os domínios de aplicação são insumos fundamentais no exercício da análise e permitem que uma solução seja especificada em uma linguagem dentre as disponíveis. O trabalho do designer de sistema é o de estabelecer o roteiro de refinamentos que conduz esta especificação até o código executável.

2.6. Conclusão

A reutilização de software ganha importância na pesquisa em engenharia de software na medida em que, através dela, podemos aumentar os índices de produtividade e de qualidade do desenvolvimento de sistemas. Este ramo de pesquisa ocupa-se do estudo de técnicas e métodos para aproveitar o esforço despendido na elaboração de soluções de software em novos sistemas. As tecnologias desenvolvidas para aproximar-nos deste objetivo classificam-se em duas linhas principais: as tecnologias compositivas e as tecnologias gerativas. Estas não são categorias estanques, mas direções gerais ao longo das quais as soluções propostas se alinham. Na medida em que são elaboradas soluções que mesclam características de ambas as linhas, a fronteira entre o compositivo e o gerativo torna-se mais tênue e difusa.

Dentro da pesquisa em reutilização uma das linhas mais importantes é a investigação de mecanismos para a construção de linguagens especializadas em domínios de aplicação, que permitam elevado grau de abstração e alta produtividade. Dentro desta linha uma das mais importantes propostas foi a estratégia DRACO, que lançou as bases para a análise de domínio, e provê um suporte genérico à construção de linguagens para domínios de aplicação.

3. Orientação a Objetos

3.1. Introdução

Orientação :	*Ato ou arte de orientar(-se). Indicar o rumo a; dirigir, encaminhar, guiar. Direção, guia, regra.*
Para :	*disposição, determinação, intento, tendência (podendo, não raro, aqui, ser substituído por A)*
Objeto :	*Tudo que é manipulável e/ou manufaturável. Mira, fim, propósito, intento; objetivo.*

O paradigma da orientação a objetos tem suas raízes plantadas na área de linguagens de programação. O primeiro uso dos termos **objeto** e **classe** ocorreu na linguagem SIMULA [Birt73], com a intenção de representar os objetos envolvidos em uma simulação como objetos de software. Estes conceitos foram estendidos ao desenvolvimento de protótipos e de sistemas, sendo marco desta evolução o ambiente Smalltalk-80 [Gold83].

O interesse despertado por este novo paradigma foi explosivo. Em pouco tempo surgiram várias ramificações da proposta original, novas linguagens e revisões de linguagens já existentes, muitas delas trazendo conjuntos próprios de definições e interpretações. Áreas de pesquisa tradicionais, como Bancos de Dados e Engenharia de Software, encontraram no paradigma uma forma de organizar informação e conhecimento.

Hoje, a orientação a objeto congrega métodos, técnicas e tecnologias que permitem a construção e evolução de especificações [Coad92] [Booc91], bases de dados [Khos94], arquiteturas de software e programas. A despeito de diferenças particulares de cada abordagem, todas as soluções orientadas a objetos seguem uma filosofia de

organização que compreende os princípios de encapsulação, modularidade e hierarquia [Booc91] .

Dentre as contribuições do paradigma para a construção de software, destacamos duas como principais:

- Redução da distância semântica entre o software [Jaco92]
- Uniformidade conceitual ao longo do desenvolvimento [Coad91] [Coad92]

A distância semântica entre um modelo e a realidade é inerente à construção de um modelo e decorre dos processos intelectuais de abstração, que suprimem detalhes considerados irrelevantes para gerir a complexidade [Booc91]. Estes detalhes ignorados são discrepâncias entre a representação e o aspecto da realidade representado. A orientação a objetos possibilita a construção de arquiteturas mais próximas da realidade percebida. Como conseqüência desta aproximação, o software passa a ser mais compreensível [Jaco92] [Meye88]. Yourdon defende que a aplicação destas técnicas desde a análise permite a construção de um núcleo estável de software que concentra as representações desta realidade, chamada por ele de "responsabilidades do sistema dentro do domínio de aplicação" [Coad92]. O papel preponderante da encapsulação no paradigma aumenta ainda mais esta estabilidade. Nas palavras de Jacobson, "as modificações no modelo tendem a ser locais, porque elas freqüentemente resultam de um único item, que é representado por um único objeto" [Jaco92].

A uniformidade conceitual é decorrente da aplicabilidade da orientação a objetos em todas as fases de desenvolvimento. Desta vez, um único paradigma conceitual pode ser adotado desde a análise até a organização dos dados e a codificação, ao contrário do que ocorria com as técnicas estruturadas. O refinamento semântico inerente ao desenvolvimento envolve apenas modelos da mesma natureza. Isto torna mais simples a transição entre as fases do ciclo de vida, facilitando o rastreamento permanente dos desdobramentos que levam das especificações ao código.

A despeito da existência de uma base conceitual comum dentro da orientação a objetos, diferentes sistemas e tecnologias assumem encarnações próprias dos conceitos. A origem destas divergências está tanto na história do amadurecimento da área [Nier89] [Unga87] [Wegn87] [Wegn90] quanto na diversidade de campos em que a orientação a objetos é aplicada.

Neste capítulo, apresentamos as definições dos conceitos fundamentais da orientação a objetos, segundo as definições adotadas no restante da tese, e discutimos sua relação com a reutilização de software.

3.1.1. Duas Palavras Sobre Finalismo

Todo trabalho que expõe definições de conceitos em linguagem natural requer um esforço para que estas sejam precisas. Mesmo que vez por outra encontremos uma definição vaga como *"A Responsabilidade do Sistema é uma organização de elementos relacionados de modo a constituir um todo"* [Coad92] ou *"um objeto consiste*

em alguma memória privada e um conjunto de operações" [Gold83], percebemos que estas são exceção e não regra.

Neste capítulo, esta intenção de precisão passou pela adoção de uma postura teleológica, em que os conceitos são baseados na finalidade de seu emprego. Defendemos que a abordagem finalista (ou teleológica) tem aplicação direta na caracterização dos ambientes computacionais, onde não há espaço para o supérfluo ou o inútil. A existência de cada conceito ou componente de sistema é causada pela necessidade de solução de um problema.

3.2. Objetos

Ponto central em todas as abordagens orientadas a objetos, o conceito de **objeto** é o que mais dispõe de definições alternativas. Eis a definição encontrada no Smalltalk-80 - principal difusor da orientação a objetos [Nier89] : "objetos são componentes de software (...) e consistem de alguma memória própria e um conjunto de operações. (...) A propriedade crucial de um objeto é que sua memória privativa apenas pode ser manipulada por suas operações" [Gold83]. Já Wegner define objetos como "um conjunto de funções que compartilham um estado" [Wegn87]. Estas definições são bastante representativas, tanto na caracterização que é dada ao conceito como na sua generalidade, talvez excessiva. Na primeira definição, por exemplo, poderíamos enquadrar as bibliotecas de rotinas, como a IMSL [IMSL79]. Na segunda, o conjunto de funções de um programa compartilha um estado representado pelas variáveis globais. Um e outro exemplos fogem

daquilo que usualmente chamamos de objetos. Em 1990 Wegner, restringe sua definição de objetos aos encapsulados, como no Smalltalk-80 [Wegn90].

Um objeto é um componente de software com características particulares que o tornam adequado ao desenvolvimento de software altamente modular, com ênfase na encapsulação e no aumento da independência entre as partes. As propriedades principais que um objeto apresenta são:

- identidade
- funcionalidade
- estado

A *identidade* de um objeto é a propriedade que o distingue como um módulo de software independente, dentro de um sistema onde muitos objetos cooperam. Conhecida a identidade de um objeto sua *funcionalidade* pode ser invocada pelo envio de mensagens. O objeto representa, portanto, um potencial de processamento expresso por um conjunto de operações. Estas são organizadas de modo que cada mensagem solicite a execução de exatamente uma destas operações. Por fim, o *estado* é constituído de informação encapsulada que pode ser usada e alterada pela execução de uma de suas operações. O estado de um objeto compreende as identidades de outros objetos com os quais pode cooperar. A cooperação é obtida pelo envio de mensagens a estes objetos, e conhecimento de valores respondidos pela mesma. A figura 3.1 ilustra estes relacionamentos, onde um objeto é formado pela identidade **A** que o distingue dos outros dois objetos, pela sua funcionalidade, composta por três operações, e pelo seu estado

encapsulado, que contém referências a outros dois objetos, de identidades **B** e **C** . Este diagrama mostra que o objeto A pode enviar mensagens aos objetos B e C, e apenas a estes.

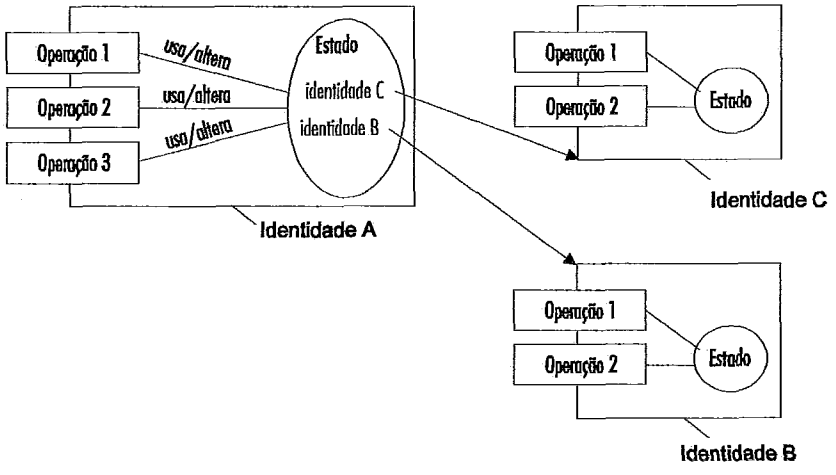


FIGURA 3.1 - Relacionamento entre identidade, funcionalidade e estado de um objeto.

Sintetizamos estas características na seguinte definição de objeto.

***Objeto** é uma representação computacional que existe por um período de tempo durante o qual é portadora de informação, de funcionalidade e ainda é passível de manipulação externa ou por software através de interface bem definida. Esta interface tem por finalidade abstrair quaisquer detalhes da realização do objeto (implementação de funções, estruturas de dados, etc.), constituindo o único instrumento através do qual este se torna útil. A qualquer objeto é vetada, por definição, a interferência direta sobre outras unidades portadoras de informação.*

3.3. Mensagens

Em um software, a independência entre os módulos é desejável [Page88]. A orientação a objetos oferece um elevado nível de independência entre objetos. O mecanismo de cooperação entre objetos é baseado em uma analogia com o processo de comunicação humana. Uma mensagem é enviada do emissor ao receptor, e cabe a este último a responsabilidade de decodificá-la. Uma mensagem enviada a um objeto é, ao mesmo tempo, uma solicitação de que execute uma operação de sua funcionalidade e um meio de transporte de informação. Nas palavras de Goldberg , "uma mensagem é uma solicitação para que um objeto leve a cabo uma de suas operações. Uma mensagem especifica qual a operação desejada, mas não como esta operação vai ser levada a cabo. (...) Uma propriedade crucial das mensagens é que elas são o único meio de invocar as operações de um objeto" [Gold83].

Muito embora a orientação a objetos não seja necessariamente procedimental [McCa92], o parentesco entre o envio de mensagem e a chamada de procedimento tem raízes históricas em Simula e é evidenciado em linguagens como C++ e mesmo em Smalltalk-80. Nesta visão, a mensagem pode portar um número arbitrário de argumentos, e resulta em um objeto respondido pelo receptor ao término da execução da operação. Vale a pena ressaltar que, em qualquer abordagem, é de responsabilidade exclusiva do receptor a determinação do modo como reagirá à mensagem. Deste modo, fica garantido que o emissor dependa exclusivamente da interface do receptor. As definições relevantes à troca de mensagens adotadas nesta tese são:

Mensagem: *Instrumento que intermedeia a cooperação entre objetos, significando a solicitação de operação a um objeto e, eventualmente , carregando dados para sua operação e respostas. Tais dados também são objetos com os quais o receptor tem liberdade de interagir.*

Emissor: *Objeto que envia a mensagem, isto é, objeto que solicita a realização da operação ao receptor.*

Receptor: *Objeto que recebe a mensagem, que corresponde a uma operação de sua interface. A execução da mesma operação é de sua responsabilidade e seus mecanismos ocultos do Emissor. É lícito a um objeto enviar uma mensagem para si mesmo. Neste caso, ele é tanto emissor quanto receptor da mensagem.*

Método: *Implementação de uma operação de um objeto. O receptor de uma mensagem determina qual método será executado.*

3.3.1. Polimorfismo

A palavra polimorfismo possui em botânica, genética e física, significados consagrados, que nada tem em comum com o sentido que lhe dá a literatura da orientação a objetos. Enquanto nas ciências naturais a palavra sempre é empregada com o sentido de formas externamente diversas para objetos ou seres de mesma natureza, o sentido na orientação a objetos é exatamente o oposto: o de caracterizar dois objetos intrinsecamente diferentes, mas que apresentam a mesma aparência externa. Dizemos que dois objetos apresentam **polimorfismo** quando reagem com funções diversas a uma mesma mensagem.

Retornando à figura 1.2, vemos que a mensagem "<" é enviada a dois elementos de uma lista, com a finalidade de ordená-los. Graças à intermediação através de mensagens, o mesmo trecho de código pode ordenar qualquer conjunto de objetos em que o polimorfismo em relação à mensagem "<" caracterize uma boa ordem.

O poder do polimorfismo é plenamente exercido através do mecanismo de ligação dinâmica ("late binding"). A ligação tradicional vincula de modo estático a chamada de um procedimento ao código que o implementa. Na ligação dinâmica, o vínculo entre o envio de uma mensagem e o código dela derivado se dá a tempo de execução, em função do receptor da mensagem. Este mecanismo geralmente consome mais tempo que a chamada direta de procedimentos [Rine91]. Este "overhead" pode ser significativamente reduzido no processo de tradução de programas orientados a objetos em código de máquina. Na linguagem C++, por exemplo, a eficiência das mensagens polimórficas aproxima-se muito da chamada de procedimentos [Germ92].

3.4. Classes

Os conceitos de orientação a objetos discutidos até agora mostram o aspecto operacional, extremamente relevante para a programação orientada a objetos e mesmo para os bancos de dados orientados a objetos. O papel destes conceitos nas primeiras etapas da construção de sistemas é, contudo, secundário. A engenharia de software, em geral, e a reutilização, em particular, preocupam-se principalmente com a descrição destes objetos e sua organização em arquiteturas de níveis mais altos.

Na orientação a objetos, encontramos como bloco fundamental na construções de modelos descritivos o conceito de "**classe**", que oferece o que Blair et al. chamaram de **abstração baseada em conjunto** [Blai89], onde o número de elementos de um conjunto torna-se irrelevante à luz das propriedades apresentadas uniformemente por eles. Portanto, adotamos a seguinte definição:

> ***Classe** é uma descrição comum a um número arbitrário de objetos que são ditos instâncias desta classe. Uma classe descreve tanto a interface quanto a realização de suas instâncias. Deste modo, as instâncias de uma mesma classe diferem apenas pela sua identidade e seu estado.*

As classes tem um papel destacado na orientação a objetos porque os programas, as bases de dados e as especificações são, em sua maioria absoluta, definidas em termos de classes.

A descrição de classes freqüentemente é enriquecida com informações sobre o relacionamento de suas instâncias com as instâncias de outras classes [Stro93] [Meye89] [Meye92].

3.4.1. Classes abstratas

A expressão "número arbitrário" adotada na definição de classe abre a possibilidade da existência de classes com um única instância, e mesmo de classes sem nenhuma instância.

Enquanto as classes com uma única instância destinam-se a objetos com alguma singularidade importante, as classes sem instâncias servem tão somente para emprestar sua descrição a outras classes através dos mecanismos de herança (seção 3.5). Neste caso, estão muitas vezes incompletas em sua definição, representando apenas os aspectos comuns às classes que dela herdam.

Classes abstratas são muito importantes no estabelecimento de arquiteturas de classes reutilizáveis. Seu uso permite descrever os padrões de comportamento e interação de componentes de uma arquitetura. Estes componentes serão desenvolvidos como classes especializadas das classes abstratas, de modo a reutilizar estes padrões através de herança.

3.5. Herança

Um grande vilão na manutenção de software é a alteração de um código já existente, pois ao editar o fonte de um módulo algum aspecto da finalidade do módulo pode

inadvertidamente escapar ao programador e causar um efeito colateral indesejável. Se de todo não podemos impedir a alteração de código já existente, podemos restringi-la aos pontos em que seja estritamente necessária. A orientação a objetos permite que o tratamento de novas situações interfira ao mínimo com o sistema já existente, através da evolução incremental dos componentes de software. A chave do processo é o poder de definir um componente como uma modificação de algum outro já existente. Não apenas mantemos o original inalterado, mas ganhamos o esforço de análise, projeto e programação das partes comuns entre o novo e o já aplicado. O esforço se concentra em detectar e tratar as diferenças entre os componentes, aproveitando todo o restante comum. A este aproveitamento chamamos herança.

O mecanismo de herança é de suma importância na orientação a objetos. Wegner assume que apenas os sistemas dotados de herança podem ser verdadeiramente chamados de orientados a objetos [Wegn87] [Wegn90]. Neste trabalho adotamos a seguinte definição:

> ***Herança:*** *Relacionamento entre duas classes que dele participam em papéis distintos (superclasse e subclasse) em que, durante o tempo em que as classes e o relacionamento* ***estiverem*** *definidos, todas as definições pertinentes à interface ou realização da superclasse aplicam-se implicitamente à subclasse, à menos de variações diferenciais. A descrição da subclasse passa a ser tão somente a descrições destas*

variações. As definições aproveitadas sem modificações são ditas herdadas.

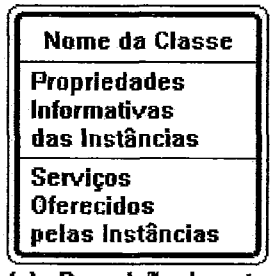
O aspecto temporal da herança é essencial para compreender seu papel na evolução de software orientado a objetos, porque todas as transformações aplicadas sobre a superclasse (refinamentos, evoluções, manutenções etc) afetam diretamente as subclasses.

A figura 3.2 ilustra um exemplo de herança entre classes usando a notação descrita em [Coad91] [Coad92]. Nela podemos ver a herança entre classes sendo usada para revelar níveis de abstração diferentes entre objetos. Um funcionário é, em um nível de abstração mais alto, uma pessoa, e por isto compartilha de todas as definições de pessoa.

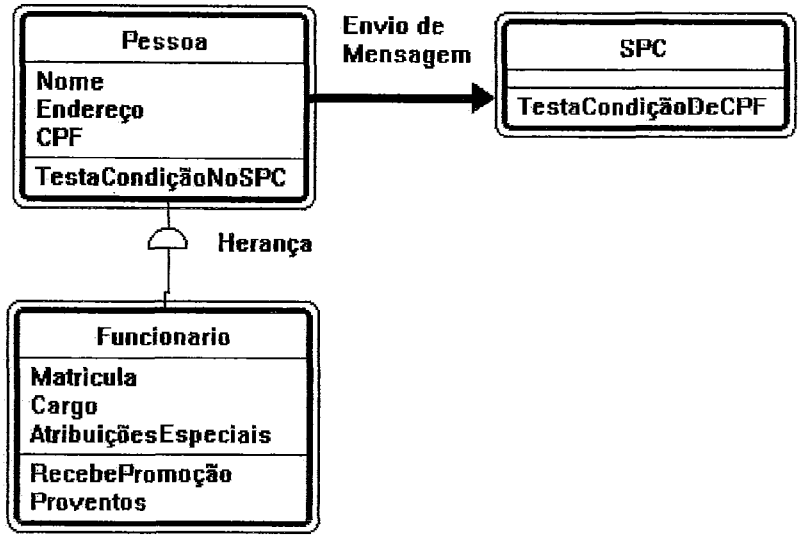
À organização da herança entre classes de um modelo chamamos **hierarquia de classes.**

3.6. Composição

Como mencionamos na seção 3.2 em que tratávamos de objetos, um objeto pode ter visibilidade de outros objetos através de seu estado, e é apenas a estes objetos que pode enviar mensagens. Em alguns casos este tipo de ligação é efêmera, existindo apenas na execução de uma operação. Na maioria das vezes, no entanto, um objeto mantém vínculos estáveis com outros, vínculos estes que portam significado.



(a) - Descrição da notação de classe



(b) - Exemplo de herança

Figura 3.2 - Exemplo de Herança. *A parte (a) da figura mostra a notação que Yourdon dá para classes, separando em três áreas da representação: o nome da classe, a lista de propriedades informativas das instâncias e o conjunto de operações que caracterizam a interface comum a estas mesmas instâncias. A parte (b) traz um exemplo com três classes de objetos (SPC - Serviço de Proteção ao Crédito, Pessoa e Funcionário). Este diagrama mostra que a classe Funcionário herda a definição de pessoa. Sendo assim, cada objeto da classe Funcionário porta informações de matrícula, cargo e atribuições especiais, bem como as de nome, endereço e CPF, definidas em sua superclasse. Semelhantemente, sua interface incorpora a operação TestaCondiçãoNoSPC.*

O uso de vínculos estáveis entre objetos é um instrumento para a concepção de objetos complexos. Tais objetos possuem em sua realização vínculos estáveis com objetos mais simples e sua funcionalidade é responsável por coordenar as funcionalidades dos objetos mais simples. A figura 3.3 ilustra um exemplo de composição.

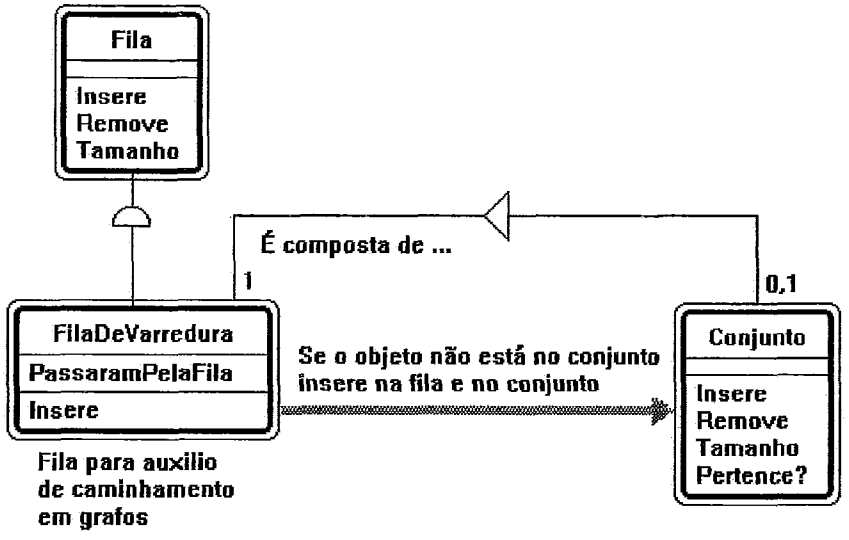


Figura 3.3 - Exemplo de Composição. *A classe FilaDeVarredura é uma variação de uma fila comum, especializada para servir ao caminhamento em grafos. Sua particularidade está em que apenas a primeira inserção do objeto na fila é válida. Construído como subclasse de fila - uma fila comum - FilaDeVarredura é composta por um conjunto, chamado PassaramPelaFila, que contém todos os objetos que estão ou em algum momento estiveram na fila. A composição compreende a inclusão do atributo PassaramPelaFila e pela construção do código específico que integra a funcionalidade do componente Conjunto ao do objeto composto FilaDeVarredura. Neste exemplo o código está na redefinição da mensagem Insere, que define a diferença de comportamento entre uma Fila e uma FilaDeVarredura.*

3.7. Orientação a Objetos e Reuso

A orientação a objetos não promove o reuso per si, mas quando a reutilização de software é uma meta, ela oferece significativo aumento de produtividade sobre o desenvolvimento estruturado tradicional [Lewi91].

Estas conclusões de Lewis originam-se em uma experiência para avaliar o aumento de produtividade obtido pela reutilização com e sem o uso da orientação a objetos. O experimento foi elaborado em duas etapas. Na primeira, um grupo de engenheiros de software estudou um problema e produziu duas bases de software reutilizável, uma destinada a uma linguagem estruturada (PASCAL) e outra destinada a uma linguagem orientada a objetos (C++). Na segunda etapa, um outro grupo de engenheiros de software foi dividido em quatro grupos homogêneos a que foi destinado um problema relacionado com a área estudada na primeira fase. Enquanto dois grupos trabalharam para solucionar o problema fazendo uso da reutilização (um em PASCAL e outro em C++), os dois grupos restantes eram usados como grupos de controle, proibidos de exercer reutilização.

O trabalho de cada grupo foi avaliado em número de execuções, número de erros a tempo de execução, tempo para corrigir os erros, número de edições e números de erros de sintaxe. Os resultados da experiência indicaram que os grupos de desenvolvimento sem reutilização apresentavam produtividade muito próxima um do outro. Os grupos exercendo reutilização progrediram melhor, com uma vantagem significativa para o grupo com reuso e orientação a objetos.

A vantagem do grupo de reutilização com a orientação a objetos é proporcionada pelos mecanismos específicos da orientação a objetos que apoiam o reuso intencional. Dentre elas, merecem destaque a encapsulação obrigatória, a composição, o polimorfismo e os mecanismos de herança.

A construção de software baseada em classes de objetos encapsulados é um princípio universal da orientação a objetos. Ao desenvolver software reutilizável, este princípio permite a criação de módulos bem definidos. Os mecanismos de integração de classes (composição e herança) transformam-se no mecanismo de integração de módulos reutilizáveis. Deste modo, a orientação a objetos adere bem à estratégia de reutilização pelo desenvolvimento de módulos de boa qualidade para o reuso. Além disto, a independência entre a interface de um objeto e sua implementação promovem uma redução do impacto de manutenção, já que a implementação de um objeto pode ser substituída sem que seja necessária a alteração de sua interface externa. Esta independência também permite que algoritmos, programas e mesmo sistemas sejam construídos com base nos padrões de interface genéricos, posteriormente resolvidos (figura 2.3).

Outra conseqüência da encapsulação é a localidade das manutenções. Um objeto interage apenas com os objetos que são seus vizinhos e, ainda assim, usa mensagens de acesso a sua interface pública. Uma manutenção que envolva a interface de um objeto afeta apenas aos objetos que usam esta interface. É possível que algum destes objetos indiretamente afetados sofra uma manutenção de sua interface em decorrência da primeira alteração. Esta propagação, todavia, acaba sendo amortecida por objetos cuja interface não será afetada (figura 3.4).

A composição de objetos também é útil à reutilização. A implementação de um objeto composto pode ser usada para integrar a funcionalidade de componentes reutilizáveis. Esta integração, além de promover a cooperação dos componentes, cria um novo módulo, com interface própria. Esta interface pode ser desenvolvida para tornar o objeto composto especializado na aplicação a que se destina, ou para caracterizar um novo componente reutilizável. Nesta última alternativa, a reutilização deste objeto composto implica na reutilização dos seus componentes e do código de integração.

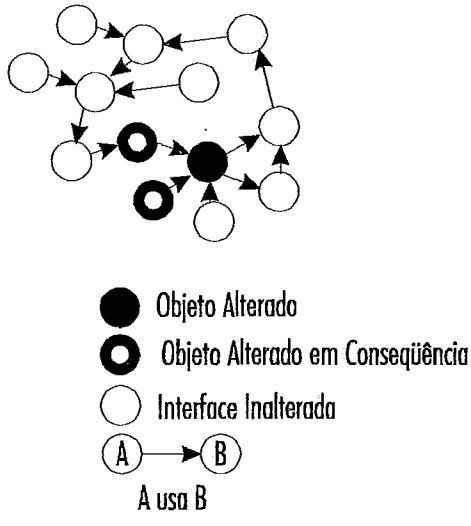


Figura 3.4 - Localidade de Manutenção - *Quando a definição de um objeto tem que ser atualizada (círculo negro), eventualmente alguns de seus usuários devem ser igualmente mantidos (anéis negros). Esta modificação em grande número de casos é absorvida sem exigir alteração nas interfaces dos objetos, o que esconde a manutenção do restante do sistema (círculos brancos).*

O polimorfismo é um mecanismo extremamente poderoso para o desenvolvimento de módulos reutilizáveis. Graças a este mecanismo, tudo que um objeto precisa saber a respeito de outros com que interage enviando mensagens é que eles atendem a protocolos específicos. A relação entre uma classe reutilizável e os protocolos de interação que ela exige definem padrões de interação, passíveis de reutilização.

A herança dá suporte à reutilização de diversas formas. A mais rudimentar delas é a criação de uma subclasse para implementar um componente que apresente uma diferença sutil em relação à classe original. O poder organizador da hierarquia de classes é, contudo, um recurso muito mais valioso. Classes e subclasses representam níveis de abstração diferentes para objetos. Superclasses podem concentrar os aspectos (estruturas de dados e código) comuns a diversas subclasses, que deles fazem reuso, complementando-os onde for necessário.

Finalmente, o paradigma de desenvolvimento orientado a objetos influi positivamente o comportamento dos engenheiros de software. O desenvolvimento dentro de um paradigma de orientação a objetos gera uma postura mais receptiva ao aproveitamento de classes prontas [John88].

A orientação a objetos, como qualquer tecnologia, traz consigo alguns problemas. O principal é o potencial de explosão do número de classes dentro de um ambiente de desenvolvimento, dificultando a localização daquela que melhor se adapte à reutilização (este problema é um caso particular do problema de classificação e busca de partes de software [Neig89]). Outro aspecto importante é a necessidade de compreender o funcionamento da classe para estabelecer se ela é adequada ou não às necessidades, se ela

requer alguma modificação (ou especialização) para adaptar-se ao reuso, e como esta adaptação deve ser conduzida. A solução para estes obstáculos está em munir o ambiente de ferramentas de organização e apoio ao reuso [Prie85] [Sim90] [Xav90] [Bell92] [Wern92a] [Wern92b]. Há ainda a necessidade de controle da qualidade na construção de classes reutilizáveis. Critérios de qualidade permitem escolher a melhor alternativa de representação de um aspecto da realidade, de modo a tornar o software mais manutenível e reutilizável. Alguns desses critérios podem ser encontrados em [John88] e [Coad91].

3.8. Conclusão

A orientação a objetos é um novo paradigma de organização de software que atingiu grande popularidade na indústria e na comunidade científica da computação. Seu potencial como tecnologia para reutilização já foi comprovado empiricamente, sendo justificado pelos mecanismos de encapsulação, composição, herança e polimorfismo, que permitem a especificação e construção de software em níveis de abstração maiores dos possíveis nas tecnologias estruturadas.

4. A proposta Gremlin

Este capítulo descreve nossa experiência no estudo de técnicas de representação de software em uma forma adequada para o reúso, que convergiu para a proposta que chamamos Gremlin. O nome Gremlin foi escolhido para lembrar que nosso objetivo é **g**erar **re**utilização por **m**ecanismos baseados em **lin**guagens. Este nome é usado nesta tese para caracterizar tanto a ferramenta descrita no capítulo 5, como a abordagem de desenvolvimento em que se insere - esta objeto do corrente capítulo. Neste capítulo, sempre que fizermos referência à ferramenta, o faremos como "Ferramenta Gremlin".

4.1. Introdução

Conforme dito na seção 2.5.1, a abordagem DRACO [Neig84] [Free87a] [Neig89] foi extremamente importante dentro da pesquisa em reutilização. Através dela Neighbors criou o conceito de análise de domínio. Atualmente, trabalhos de aperfeiçoamento desta abordagem vem sendo realizados trazendo novas perspectivas [Leit92][Leit94].

Gremlin é uma abordagem que se alinha com a estratégia DRACO, no sentido de enfatizar o papel da linguagem de alto nível orientada ao domínio de aplicação na reutilização de software. Gremlin, à semelhança do DRACO, oferece níveis de estratificação de conceitos que mapeiam domínios de aplicação, para formar espaços de trabalho dentro dos quais a solução abstrata para um problema é especificada. Tanto a construção destes espaços de trabalho, como os mecanismos de tradução das especificações de alto nível em

especificações progressivamente mais próximas da máquina foram desenvolvidos para explorar as vantagens do paradigma da orientação a objetos. Para isto foi desenvolvido um modelo de objetos particular, que acrescenta às características tradicionais um suporte mais rico a transformações. Este modelo estendido foi integrado a uma abordagem de reutilização orientada a domínios de aplicação, servindo como ferramental básico da caracterização destes domínios.

A reutilização na abordagem Gremlin se dá em dois níveis. No primeiro nível, componentes de software orientados a objetos são reutilizados na representação de novos domínios de aplicação, e na evolução deste domínio no tempo. A orientação a objetos é empregada para representar conceitos de análise e alternativas de projeto (design) disponíveis para a solução de problemas neste domínio de aplicação. O modelo estendido é empregado, ainda, na caracterização das linguagens de alto nível especializadas no domínio. O segundo nível de reutilização é obtido pelo uso das linguagens de domínio para a construção de soluções de problemas. Neste nível as bases conceituais do domínio de aplicação são reutilizadas em níveis mais altos de abstração. Neighbors caracteriza este tipo de reutilização como "reutilização de análise" e "reutilização de design" [Neig89].

As demais seções deste capítulo descrevem as extensões de um modelo de objetos que melhoram sua integração com abordagens de reúso gerativo, e a abordagem Gremlin para o desenvolvimento orientado a domínios de aplicação.

4.2. O Modelo Orientado a Objetos Estendido

4.2.1. Fundamentos

Um modelo de objetos é a sistematização dos princípios da orientação a objetos para fins específicos, geralmente associado a uma linguagem de programação, banco de dados ou ferramenta de software que dá suporte operacional a ele. Cada modelo de objetos possui sua definição particular para os conceitos da orientação a objetos. A título de exemplo, o modelo de objetos da linguagem C++ e o da linguagem Eiffel definem classes como tipos abstratos de dados [Stro93] [Meye92]. Já o ambiente Smalltalk-80 define classe como um objeto responsável por representar o protocolo comum e a implementação de outros objetos, chamados suas instâncias, sem introduzir o conceito de tipo abstrato de dados [Gold83]. Os objetos de Smalltalk-80 e Eiffel não precisam ser explicitamente destruídos, enquanto em C++ sim.

Estas divergências tornam as linguagens orientadas a objetos diferenciadas, em um modo similar ao das linguagens de programação convencionais. As diferenças entre estes modelos privilegiam a construção de representações de uma determinada classe de problemas em detrimento a outras. O que varia é a naturalidade com que as soluções de uma área são construídas em cada classe de problemas.

4.2.2. Requisitos de um modelo de objetos estendido para transformações

Os modelos orientados a objetos convencionais não dão suporte específico a transformações de programas ou de especificações. Quando a reutilização baseada em

geração é promovida em ambientes baseados em tais modelos coexistem dois universos conceitualmente distintos. Em um deles está a orientação a objetos, cujas facilidades gerais são usadas para a implementação da funcionalidade do ambiente e da representação interna das especificações. Em outro está o modelo de transformações, uma camada conceitual erguida sobre a orientação a objetos subjacente. O grau de integração entre estes dois universos conceituais é, hoje, fraco. Limita-se ao uso da orientação a objetos como base de implementação do paradigma de transformação.

Um modelo de objetos estendido para dar suporte específico a transformações promove a integração destes dois universos, permitindo a elaboração de ambientes de reutilização que explorem o melhor de cada abordagem. Os requisitos para tal mecanismo são:

1.**Mecanismo geral de transformação** - O modelo estendido deve ser dotado de um mecanismo genérico de transformação, bem como de instrumentos para especializar estes mecanismos. Deste modo mecanismos específicos de reutilização gerativa podem ser criados.

2.**Homogeneidade conceitual** - As extensões do modelo devem estar homogeneamente integradas às demais características da orientação a objetos, evitando a distinção conceitual do que seja ou não gerativo.

3.**Fidelidade ao paradigma da orientação a objetos** - As características de herança, encapsulação, cooperação através de mensagens e polimorfismo devem ser

preservadas. Isto garante que não sejam perdidos o poder de expressão e a generalidade da orientação a objetos.

4.2.3. Um modelo de objetos estendido com mensagens lingüísticas

Um modelo de objetos estendido que atende aos requisitos da seção 4.2.2 pode ser criado a partir de uma revisão do mecanismo de troca de mensagens entre objetos.

Conforme já descrito no capítulo 3, a orientação a objetos promove a cooperação de objetos através de trocas de mensagens. Este mecanismo pode ser visto como uma simplificação extrema do processo humano de comunicação. Podemos destacar pelo menos quatro pontos de semelhança evidente.

1.Existe um emissor, responsável por codificar a mensagem.

2.A mensagem é instrumento portador de informação.

3.Existe um receptor a quem a mensagem se destina.

4.A decodificação desta mensagem é de responsabilidade do receptor.

De um modo geral, os diferentes modelos de objetos são muito similares no modo de implementação da troca de mensagens, segundo o padrão definido na figura 4.1. Inicialmente o emissor codifica a mensagem como um seletor, usualmente um nome ou um número que a identifica. Opcionalmente objetos complementares podem ser agregados a ela na forma de argumentos. Quando o receptor recebe a mensagem, ele usa uma função de decodificação para determinar o código que será ativado. Ao fim da execução, um valor é retornado ao emissor como resposta.

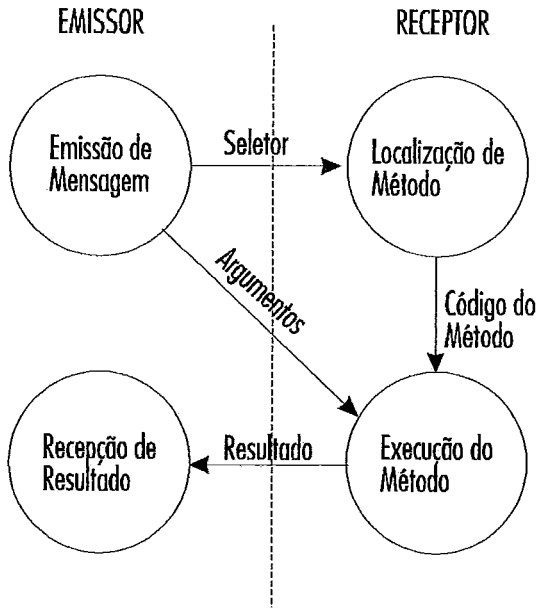


Figura 4.1 - Mecanismo tradicional que rege a troca de mensagens entre objetos

Esta abordagem distancia-se da comunicação humana em alguns aspectos significativos. A estrutura das mensagens da comunicação humana são de organização extremamente complexa. Através desta complexidade são estabelecidos os relacionamentos entre os significantes, de modo que o significado final da comunicação possa ser construído pelo receptor da mensagem. Apenas em poucos casos a interpretação de uma mensagem é imediata como na orientação a objetos. Na maioria das vezes a compreensão do significado de uma mensagem é obtida por processos intelectuais superiores, intimamente ligados ao processo de pensamento e à linguagem humana [Vigo87].

Os requisitos para um modelo de objetos estendido enumerados na seção 4.2.2 são obtidos quando aproximamos ainda mais o mecanismo de troca de mensagens e a comunicação humana. Tal aproximação envolve:

1.a adoção de mensagens de estrutura complexa;

2.um mecanismo mais poderoso de interpretação das mensagens;

3.a encapsulação deste mecanismo em uma linguagem que define a organização das mensagens que um objeto pode receber.

O emprego de mensagens de estrutura complexa permite a especificação de relacionamentos de significado entre os argumentos de uma mensagem. Isto permite representar mais fielmente o comportamento que se espera do objeto receptor, ou a resposta que desejamos que ele produza.

O uso de uma linguagem para descrever as regras de formação destas mensagens organiza a complexidade das mensagens em uma estrutura bem definida que, ao mesmo tempo, encapsula a implementação interna do processo de interpretação e reação à mensagem, e representa, de modo mais claro, a visão que este objeto tem do contexto em que é utilizado ou reutilizado.

Finalmente, a interpretação de mensagens complexas exige a integração de três fatores: um mecanismo genérico de interpretação, a especificação da linguagem particular e a especificação das ações semânticas envolvidas na interpretação.

Este tripé caracteriza um mecanismo genérico de transformação tal como especificado na seção 4.2.2, em que mensagens são transformadas em ações ou representações.

A figura 4.2 dá um exemplo de um modelo de troca de mensagens estendido como descrito.

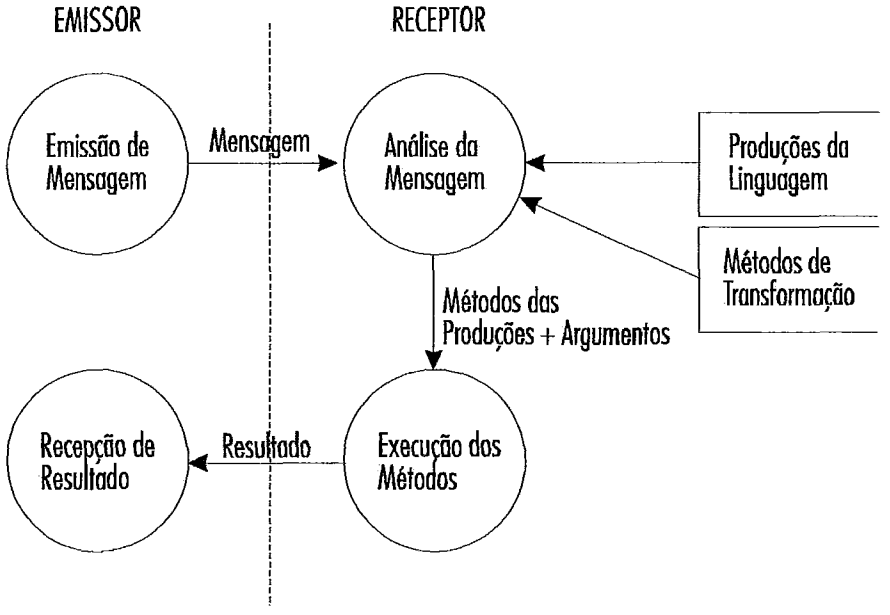


Figura 4.2 - Exemplo de extensão do mecanismo de troca de mensagens - *Neste exemplo mostramos um mecanismo de troca de mensagens em que as mensagens de um objeto são descritas por uma linguagem livre de contexto, cujo mecanismo de transformação é regido por procedimentos associados à aplicação das produções. O paradigma de envio de mensagens permanece muito próximo da orientação a objetos tradicional. A interpretação da mensagem é um método de compilação tradicional, que passa por sucessivas transformações da mensagem original até que um resultado seja completamente definido.*

4.3. Domínio de Aplicação no enfoque Gremlin

4.3.1. Fundamentos

Domínio é dicionarizado como *"Âmbito de uma arte ou de uma ciência; pertença."* [Aure94]

Para desenvolver software de qualquer tipo usamos, mesmo que informalmente, a especificação de requisitos do que ele fará. A necessária interpretação desta especificação requer sempre informações sobre o contexto em que o software está inserido. Hoje, reconhecemos que este conhecimento sobre o contexto pode ser organizado segundo a área do conhecimento ou da tecnologia a que cada software se presta. Este conhecimento, organizado e sistematizado, dá origem a um grande potencial de reutilização que pode ser exercido de diversos modos. A própria organização do conhecimento consiste em elemento passível de reúso. O maior ganho, porém, advém da construção de software reutilizável que dê suporte operacional a esta área.

Chamamos ***domínio de aplicação*** *à área do conhecimento que se serve ou se servirá de software, e que pretendemos organizar para a reutilização.*

Nesta definição, caracterizamos um ponto de vista de que os domínios de aplicação não preexistem ao reúso. O domínio de aplicação apenas nasce quando há a percepção do seu potencial de reúso.

As técnicas de organização do conhecimento em domínios de aplicação são alvo da pesquisa em Análise de Domínios, uma área muito nova de pesquisa para a qual

não existe ainda uma definição de consenso [Trac92]. Nesta tese, adotemos a seguinte definição encontrada em [Prie90]:

> ***Análise de Domínio*** *- é o processo onde informações (de um domínio) usadas no desenvolvimento de software são identificadas, estruturadas e organizadas para reúso posterior.*

Hoje, há diversas propostas de métodos para análise de domínios. Uma boa análise comparativa destes métodos se encontra em [Aran94].

Um ambiente para o reúso em larga escala deve prover uma base conceitual em que os diferentes métodos de análise de domínio possam se encaixar. A solução que propomos é uma leitura do processo de nascimento e evolução de um domínio à luz da epistemologia de Kuhn [Kuhn77].

A próxima seção é uma breve exposição do ponto de vista kuhniano sobre a evolução do conhecimento. Segue-se a ela uma nova interpretação dos fundamentos da análise de domínios através deste ponto de vista. Por fim, discute-se o conceito de **patamar de reúso** como o representante computacional de um domínio de aplicação.

4.3.2. A visão epistemológica de Kuhn.

O modelo de ciência proposto por Thomas Kuhn [Kunh90] é hoje amplamente discutido por diversos profissionais de distintas áreas do conhecimento [Pepe93].

Kuhn enfoca o papel da tradição da comunidade científica que dita uma linha de investigação chamada **investigação normal**. Nesta linha o cientista não é um inovador, mas um solucionador de enigmas propostos e resolvidos dentro da tradição científica da época. Kuhn usou a palavra **paradigma** para descrever esta estrutura conceitual, social e ferramental desta tradição.

A evolução da ciência se dá pela troca de paradigma, que se inicia pela evidência de **anomalias,** que são falhas da natureza em conformar-se com a expectativa. Estas anomalias inicialmente não são reconhecidas como tais, até que haja uma suficiente maturidade teórica e ferramental. Neste transcurso de tempo, são tratadas como novos enigmas para os quais se busca soluções dentro do paradigma atual. Um exemplo para ilustrar este comportamento: um astrônomo que não encontra um planeta na posição de sua órbita onde os cálculos indicavam, vai supor uma falha de cálculo ou o desvio do planeta por um outro astro desconhecido. Ele não vai supor uma exceção às leis da gravitação universal sem a evidência empírica ou teórica prévia. Reconhecidas as anomalias, a tradição científica converge para um novo paradigma que age retrospectivamente sobre o conhecimento científico, levando à transformação das técnicas estabelecidas da prática científica. Um exemplo de troca de paradigma na história da física foi o surgimento da teoria da relatividade que suplantou o paradigma da física newtoniana.

4.3.3. A visão kuhniana da análise de domínio

O estabelecimento de um paralelo entre o pensamento de Kuhn e o desenvolvimento de software baseia-se em que ambas, a ciência e a análise de domínio, buscam a sistematização do conhecimento. Ambas igualmente se valem de aparatos técnicos

e conceituais, e ambas são fortemente determinadas pela tradição prévia. O ponto central de divergência é que o software, como produto do intelecto humano, não pode ser contraposto com uma realidade única e natural. Sendo assim, no desenvolvimento de sistemas, coexistem diversos paradigmas que conduzem a resultados válidos em suas áreas de atuação.

Nós defendemos que cada domínio de aplicação compreende um **paradigma de investigação próprio**, que pode se valer de ferramentas (metodologias de análise de domínio, técnicas de representação do conhecimento, etc) já existentes ou demandar instrumentos próprios. A análise de um domínio passa a ser a aplicação contínua de seu paradigma para a construção de software reutilizável.

Traduzindo em termos da abordagem DRACO, o paradigma de um domínio de aplicação, é o conjunto de táticas adotadas para adquirir representações de conceitos ou operações do domínio e mapeá-las em componentes.

A validade de um paradigma científico depende da sua capacidade para equacionar os problemas dentro de uma realidade científica. A validade do paradigma de um domínio de aplicação depende do grau de esforço necessário para reutilizar seus conceitos.

Podemos definir anomalia em um domínio de aplicação como a dificuldade em aproveitar a base conceitual do domínio na elaboração de um sistema. O conceito de anomalia ajusta-se bastante bem a este caso, porque tal dificuldade é inicialmente considerada um caso particular daquele problema. Apenas o amadurecimento de soluções semelhantes suscitará a descoberta de um novo domínio de aplicação que trate daquele conjunto específico de problemas.

Associar anomalias de domínio com o grau de esforço necessário para o reúso permitirá o estabelecimento de medidas de adequação do domínio. O estudo destas medidas no tempo possui potencial para antever o surgimento de novos domínios de aplicação.

4.3.4. Patamar de Reúso, o representante Gremlin

O ambiente Gremlin agrupa as informações reutilizáveis em ambientes de trabalho chamados **patamares de reúso**. Para cada domínio de aplicação identificado há na ferramenta um único patamar de reúso. Um patamar de reúso concentra todas as informações consideradas relevantes em um domínio de aplicação, incluindo documentação, uma base de conceitos do paradigma, e componentes de software que especificam operacionalmente a visão atual que se tem deste domínio. O patamar de reúso cumpre a função de organizar esta diversidade de informação em um ambiente de desenvolvimento coerente.

Um patamar de reúso pretende ser um ambiente suficiente para o desenvolvimento de soluções de software em um domínio de aplicação específico. Para conquistar este objetivo é necessário que dê suporte básico a várias atividades dentro do ciclo de vida do desenvolvimento. A fim de atender a diversos ciclos de vida, estabelecemos como requisitos fundamentais que um patamar de reúso esteja apto a:

- Apoiar o processo de compreensão de conceitos do domínio, bem como dos instrumentos através dos quais são reutilizados.

- Representar modelos de conceitos do domínio de aplicação, incluindo modelos operacionais, que possam ser empregados na construção de software.

- Representar linguagens de desenvolvimento orientadas ao domínio de aplicação.

As tomadas de decisão envolvidas em um processo de reúso exigem que o engenheiro de software conheça os conceitos do domínio de aplicação envolvidos no processo. Por isto, o suporte à sua compreensão é fundamental em um ambiente de reutilização. Para dar apoio ao estudo dos conceitos o oferece a capacidade de registrar documentos em formato livre como nós de uma rede de hipertexto. Navegando por esta rede, o usuário adquire o conhecimento sobre o domínio de aplicação, capacitando-se para o processo de reúso.

Dentre as possíveis representações de um domínio de aplicação, a construção de modelos operacionais é aquela que oferece resultados à reutilização em prazos mais curtos. Um modelo operacional é uma arquitetura de classes de objetos que registra aspectos estruturais e comportamentais do domínio de aplicação, de modo que pode ser empregado em protótipos e em avaliações de qualidade. Como o Gremlin usa a orientação a objetos para representar modelos operacionais, um modelo operacional pode representar tanto arquiteturas do domínio de aplicação como padrões de interação entre objetos do domínio. A maturidade de um modelo operacional orientado a objetos é obtida quando este é formado por componentes reutilizáveis, que podem ser empregados na construção de soluções de software dentro do domínio de aplicação.

O papel das linguagens de alto nível é o de oferecer um mecanismo para descrever soluções de software, abstraindo-se dos detalhes irrelevantes à concepção da solução. Estes detalhes são supridos durante o processo de tradução da linguagem, seguindo padrões de complexidade variável. Retirando estes detalhes da atenção do engenheiro de software, a linguagem de alto nível dirige sua atenção para as características relevantes do domínio de aplicação. O processo de tradução de tais linguagens acrescenta detalhes ás abstrações originais do domínio de aplicação, aumentando sua produtividade e reduzindo as áreas onde pode haver a incidência de erro humano. Ao direcionar a atenção também influi positivamente a capacidade de criar soluções [Vigo87].

Para atender aos requisitos já enumerados, o patamar de reúso é composto por quatro subsistemas:

- **Subsistema de documentação** - responsável pela armazenagem, organização, classificação e acesso a documentos em formato livre.

- **Subsistema de modelagem** - responsável pela construção e implementação de modelos de software destinados a representar conceitos de domínios de aplicação como arquiteturas de software reutilizável.

- **Subsistema lingüístico** - responsável por construir linguagens de alto nível orientadas a domínios, para a especificação de soluções de software, e por traduzir estas especificações em termos de reúso de arquiteturas já definidas.

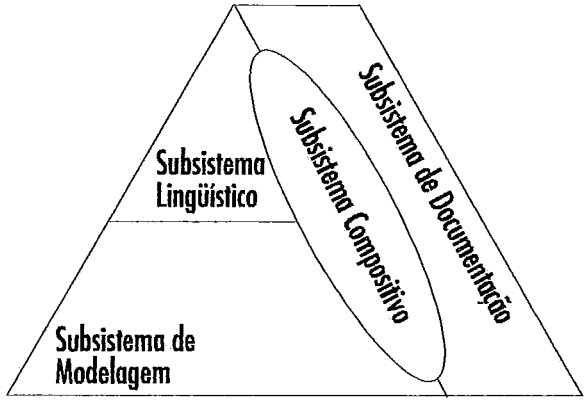


Figura 4.3 - Relacionamento entre os subsistemas de um patamar de reúso - *O subsistema compositivo integra os demais subsistemas na construção de organizações reutilizáveis..*

Cada um destes subsistemas serve para organizar diferentes tipos de informação de modo a auxiliar a reutilização intencional. A qualidade da organização destas informações é fundamental para o sucesso da reutilização [Horo84]. Como discutimos no início do capítulo, a orientação a objetos é usada para que haja reutilização durante a construção desta organização. Este nível de reúso é sustentado pelo último subsistema de um patamar de reúso:

- **Subsistema Compositivo** - responsável por classificar e organizar os diversos tipos de informação (documentos, modelos e linguagens) de modo útil à reutilização.

A figura 4.3 ilustra o relacionamento dos quatro subsistemas de um patamar de reúso.

4.3.4.1.Subsistema de Documentação

O exercício do reúso exige que o engenheiro de software domine um sistema de conceitos, através do qual o conhecimento é reutilizado. Em ambientes compositivos, este sistema de conceitos compreende a definição dos componentes reutilizáveis. Em ambientes de reúso gerativo, o sistema de conceitos compreende os mecanismos usados para construir a especificação oferecida ao processo de geração. Este sistema de conceitos caracteriza um tipo de conhecimento produzido pelo engenheiro de software reutilizável (responsável pela construção e organização do software reutilizável) e que se destina ao engenheiro de software aplicado (responsável por construir soluções de software específicas com reutilização intensiva). O sucesso da reutilização depende da eficiência com que este conhecimento é transferido entre estes profissionais.

O subsistema de documentação provê mecanismos para registrar e tornar disponíveis informações pertinentes ao exercício da reutilização dentro de um patamar de reúso. A unidade de armazenamento de informação é chamada **documento**. Dentro de um patamar de reúso, um documento é qualquer objeto informativo, que pode ser examinado ou alterado através de uma interface adequada.

Os documentos integram-se aos demais objetos dentro do patamar de reúso em uma rede. O engenheiro de software interessado em reutilização navega por ela como em um hipertexto, de modo a atingir os documentos ou outros objetos que deseje.

O subsistema de documentação está intimamente ligado ao subsistema compositivo, oferecendo através dele facilidades de construção de nomes para documentos selecionados, classificação segundo critérios pertinentes ao domínio de aplicação, e seleção de documentos por estes mesmos critérios.

4.3.4.2.Subsistema de Modelagem

O subsistema de modelagem do Gremlin é responsável pela construção, armazenamento e disponibilização de modelos de objetos pertinentes a domínios de aplicação.

Uma vez que os modelos de objetos são a forma de representação mais importante para a reutilização dentro de um patamar de reúso, sua construção deve avançar em conjunto com o processo de análise de domínio. Por isto, o subsistema de modelagem precisa dar suporte a ciclos iterativos de desenvolvimento de modelos. Nestes ciclos, a construção de um modelo surge pelo refinamento sucessivo a partir de um modelo abstrato e incompleto. A cada refinamento, o modelo de objetos incorpora mais informações colhidas da análise do domínio de aplicação. É inevitável que, neste processo de refinamento, algumas decisões de representação sejam tomadas. Para reduzir o impacto destas decisões na generalidade da representação do domínio sendo considerado, é importante considerar as alternativas disponíveis.

O subsistema de modelagem individualiza cada patamar de reúso na forma de um objeto complexo composto de classes e objetos inter-relacionados. O processo de refinamento é exercido sobre estas entidades, e pode envolver a construção de versões alternativas para a avaliação das decisões de representação consideradas.

Um modelo é dito **prototipável** quando a definição das classes é suficientemente rica para a construção de protótipos de soluções. O papel dos protótipos é levantar aspectos específicos do desenvolvimento de aplicações no domínio, baseado nos conceitos representados pelo modelo do domínio de aplicação.

Um modelo é dito **maduro** quando suas classes constituem componentes de software cuja qualidade seja julgada adequada para a reutilização dentro de um domínio de aplicação. Cada patamar de reúso disponível para o desenvolvimento de software está associado a um único modelo de objetos, necessariamente maduro. Este modelo representa todos os aspectos do domínio de aplicação disponíveis para reúso.

Os componentes de software reusável encontrados em um modelo maduro de um domínio de aplicação são chamados **componentes semânticos**. Estes componentes dividem-se em duas grandes categorias. A mais importante engloba os **componentes de domínio**, que descrevem aspectos relevantes do domínio. A segunda categoria de componentes semânticos refere-se aos **componentes auxiliares**, que se prestam a tarefas secundárias, mais relevantes às implementações de um modelo, à verificações de consistência, etc.

Cada patamar de reúso possui um conjunto de componentes peculiar, chamados de **componentes fundamentais**. O que caracteriza um componente fundamental não é o seu caráter semântico ou auxiliar, e sim o fato de não possuir definição dentro do patamar de reúso em que está inserido. A definição de um componente fundamental está em

outro patamar de reutilização (e, portanto, seu significado reside em um domínio de aplicação diverso).

O adjetivo **fundamental** vem enfatizar que todos os componentes do patamar de reúso são expressos em termos destes componentes fundamentais. Estes componentes servem de ponte entre este domínio e domínios de nível mais baixo.

Os componentes fundamentais, do mesmo modo como os componentes do DRACO, estabelecem uma ordem parcial entre os domínios, estabelecendo assim níveis diferentes de abstração.

Três pontos principais motivaram a restrição da interface entre domínios diferentes:

1. O conjunto de componentes fundamentais define uma interface através da qual um patamar de reúso de nível mais alto solicita recursos a um patamar de reúso de nível mais baixo. Este controle de interação entre os patamares é importante dado que os domínios de aplicação são, em geral, dinâmicos e seu paradigma de investigação exercido continuamente, causando, por certo, efeitos secundários entre os patamares de reúso. Interfaces bem definidas permitem o controle da propagação destes efeitos secundários.

2. Ao confinar o desenvolvimento à expressão em termos de componentes fundamentais, a abordagem Gremlin passa a contemplar a possibilidade de que opções de design sejam realizadas pelo mapeamento entre

componentes. Estes mapeamentos determinam os processos de transformação que conduzem o refinamento de especificações dentro de um domínio. A análise quantitativa e qualitativa destas transformações podem denunciar que a representação do domínio de aplicação comporta um tipo de problema. A percepção deste desvio leva à evolução do domínio de aplicação, ou à identificação de anomalias. O número de transformações desta natureza constituem uma medida rudimentar da validade do domínio de aplicação tal como discutimos em 4.2.3.

3. A obrigatoriedade da representação das soluções com base nos componentes fundamentais incentiva a reutilização destes componentes.

4.3.4.3.Subsistema Lingüístico

O subsistema lingüístico é responsável por oferecer o potencial de reutilização do modelo de objetos do domínio de aplicação através de linguagens de programação especializadas. O analista de sistemas aplicados vê um patamar de reúso como um ambiente de desenvolvimento, dotado de ferramentas de apoio à construção de sistemas. Através deste ambiente ele tem acesso a uma ou mais de uma linguagem de desenvolvimento voltada àquele domínio de aplicação. Estas linguagens e ferramentas, todavia, compartilham a mesma base conceitual que representa, com a maior fidelidade possível, o domínio de aplicação.

A existência de mais de uma linguagem de desenvolvimento em um mesmo domínio é uma divergência significativa da abordagem DRACO, que vincula diretamente uma linguagem para cada domínio. Na abordagem DRACO, contudo, a linguagem cumpre ao mesmo tempo o papel de elemento organizador do domínio de aplicação, e o papel de interface de uso deste mesmo domínio. Na abordagem Gremlin a linguagem cumpre apenas esta última função, cabendo aos modelos de classes o papel de representação dos conceitos dos domínios de aplicação. Esta restrição do papel da linguagem, permite refletir mais de perto os domínios de aplicação para os quais há mais de uma linguagem. Outra vantagem desta divisão é isolar da base conceitual do domínio de aplicação as particularidades tecnológicas inerentes à construção e evolução de linguagens. Desta forma a linguagem apresenta-se como um elemento de interface para uso dos conceitos do domínio de aplicação representados na forma modelos operacionais. Gremlin oferece esta independência relativa, abrindo a possibilidade de que coexistam mais de uma linguagem de especificação dentro de um mesmo domínio. O relacionamento entre duas linguagens quaisquer dentro de um patamar de reúso pode ser classificado em um dos seguintes casos:

1. **alternativas reais** - neste caso, o domínio de aplicação comporta ambas as linguagens em caráter permanente, de modo que especificações possam ser construídas tanto em uma quanto em outra;

2. **substituição** - neste caso, o domínio está em fase de substituição de uma linguagem mais antiga por uma linguagem mais nova, que se adapte melhor à situação conceitual do domínio de aplicação;

3. **versão** - neste caso, coexistem versões diferentes de mesma linguagem básica. Cada versão existe para atender a circunstâncias específicas de aplicação.

4. **anomalia** - neste caso a segunda linguagem se faz necessária para representar algum aspecto do domínio de aplicação que não está corretamente modelado no domínio de aplicação, ou simplesmente não faz parte dele.

A classificação destes relacionamentos não é imediata, e requer um profundo conhecimento da estrutura do domínio e da sua evolução.

Para evitar que uma possível proliferação de linguagens dentro de um domínio venha a descaracterizá-lo, definimos alguns princípios que devem ser observadas em linguagens de mesmo domínio:

1. **totalidade** - Cada linguagem de desenvolvimento deve atender à totalidade dos conceitos representados de um domínio de aplicação;

2. **uniformidade semântica** - A tradução de uma linguagem acarreta uma série de transformações em objetos que estão descritos neste patamar de reúso ou não. As representações equivalentes em uma e outra linguagem devem ser traduzidas do mesmo modo, ou convergir para representações equivalentes.

3. **formatação universal** - o produto de uma tradução em qualquer linguagem deve poder ser revertido no fonte de qualquer das linguagens disponíveis no domínio.

Devemos chamar a atenção para que, durante a manutenção do domínio ou das linguagens, podem existir lapsos de tempo em que estes princípios sejam violados. Outro aspecto relevante é que a propriedade da formatação universal, quando aplicada em domínios com apenas uma só linguagem, corresponde muito fielmente ao subsistema de formatação ("prettyprinter") do DRACO.

A unidade de organização do subsistema lingüístico é a **gramática gremlin**. Conceitualmente, uma *gramática gremlin* é uma descrição da sintaxe de uma linguagem do patamar de reúso, bem como dos mecanismos de transformação de sentenças desta linguagem em objetos do modelo de objetos do domínio. Deste ponto em diante, usaremos a palavra **gramática** e a expressão **gramática gremlin** indistintamente.

Operacionalmente, gramáticas são objetos do modelo orientado a objetos estendido (discutido na seção 4.2) cuja interface reconhece sentenças desta linguagem. Cada gramática tem a capacidade de sintetizar objetos que representam a sentença da gramática recebida, através de transformações encapsuladas em sua implementação.

Para cada linguagem de um patamar de reúso, há pelo menos uma gramática que reconheça sua sintaxe. Quando duas ou mais gramáticas reconhecem a mesma linguagem, elas são ditas **gramáticas polimórficas** e representam alternativas semanticamente distintas para tradução de uma linguagem.

O nome "gramática polimórfica" vem do modelo de objetos subjacente, que trata a tradução de sentenças como envio de mensagens. No sentido rigorosamente orientado a objetos, dois objetos que atendem de modo diferentes à mesma mensagem são ditos polimórficos.

Uma exemplo trivial de gramáticas polimórficas são gramáticas que traduzem uma especificação de alto nível em diferentes linguagens de baixo nível.

Componentes gramaticais ou gramáticas abstratas, são fragmentos de linguagens que são usados para construir novas gramáticas, ou para criar novas versões de gramáticas já existentes, de modo a alterar sua sintaxe ou sua semântica de tradução.

As gramáticas, abstratas ou não, são objetos da mesma natureza, com a diferença que a definição das gramáticas abstratas é incompleta, impedindo que ajam como elementos transformadores de sentenças.

4.3.4.4.Subsistema Compositivo

Conforme indicado na figura 4.3, o subsistema compositivo está fortemente integrado com os demais subsistemas de um patamar de reúso. Cada um destes subsistemas apresenta organizações complexas, elaboradas a partir de componentes elementares. A tabela 4.1 mostra a relação entre organização e componente dentro de cada subsistema.

Subsistema	Organização complexa	Componente
Documentação	Hiper-Documento	Documento
Modelagem	Modelo	Classes
Lingüístico	Linguagem	Gramática

Tabela 4.1 - Organizações complexas dentro de um patamar

A construção destas estruturas determina de que modos a reutilização dentro do patamar de reúso é exercida. A complexidade de construção destas estruturas acompanha a complexidade do domínio de aplicação representado pelo patamar. A aplicação de técnicas de reutilização compositiva propicia a redução de custos e aumento da qualidade na construção destas estruturas.

O subsistema compositivo tem por finalidade prover o instrumental básico para a obtenção de reutilização compositiva na construção das estruturas complexas que constituem um patamar de reúso, proporcionando o reúso da experiência prévia de estruturação de patamares de reúso. Este subsistema é responsável por organizar grandes conjuntos de componentes de software, sejam eles documentos, classes ou gramáticas. Além disto, oferece mecanismos de busca e seleção de componentes, de modo que estes são tornados disponíveis ao usuário. O exercício específico da integração destes componentes em uma organização é realizado dentro de um dos demais subsistemas.

O processo de seleção de componentes é baseado em um esquema de classificação por facetas, eficiente quando aplicado a coleções específicas de componentes [Prie91]. A classificação facetada se distingue dos mecanismos de classificação enumerativa por apresentar uma visão multidimensional do conjunto sendo classificado. Enquanto a classificação enumerativa busca a divisão sucessiva deste conjunto em subconjuntos disjuntos que compartilham características progressivamente mais específicas, a classificação facetada reconhece que um domínio de componentes pode ser examinado sob diversos pontos de vista. Cada ponto de vista, chamado **faceta**, oferece uma caracterização básica do domínio sendo classificado, enumerando uma série de características que cada objeto pode apresentar segundo este ponto de vista. Cada característica é chamada **termo** da faceta. A classificação de um conjunto de objetos envolve a identificação de facetas deste domínio, e dos termos dentro destas facetas. Cada objeto individual é classificado por termos das diversas facetas do domínio.

A classificação serve para a exploração seletiva de conjuntos de componentes, de modo a identificar aqueles que contribuem ao processo de reúso.

4.3.5. O ciclo evolutivo de um domínio dentro da abordagem Gremlin

Nesta seção, estabelecemos as grandes etapas por que passa um domínio dentro do ambiente Gremlin, baseados em [Aran91].

1. **Identificação do Domínio**. Esta etapa compreende a percepção da existência de um novo domínio e determinação do seu escopo. A percepção de um domínio

pode ter causas totalmente externas ao Gremlin, fruto da percepção subjetiva da área, ou ser motivada pelo comportamento do desenvolvimento de software dentro do Gremlin. Na seção 4.3.3, discutimos como a percepção de anomalias pode indicar a necessidade de investigar a existência de um novo domínio e na seção 4.3.4 sugerimos uma métrica inicial para o número de anomalias em um domínio. Estas características dão suporte inicial a esta etapa, mas a caracterização do domínio, sua abrangência e sua aplicabilidade estão fora do escopo da ferramenta.

2.**Definição do paradigma de investigação.** Na seção 4.3.3, definimos o conceito de paradigma de investigação como a abordagem conceitual e instrumental para a aquisição do conhecimento. Esta etapa consiste fundamentalmente no estabelecimento da estratégia de aquisição de informação.

3.**Definição dos componentes fundamentais.** Na seção 4.3.4, definimos componentes fundamentais de um domínio como conceitos de domínios de nível mais baixo sobre os quais serão construídos os conceitos do domínio de aplicação.

4.**Estruturação do patamar de reúso.** Na seção 4.3.4, definimos patamar de reúso como o espaço de trabalho onde os conceitos de um domínio são representados e onde as ferramentas do paradigma do domínio tem seu espaço de atuação. A estruturação do patamar de reúso compreende as atividades de organização de seus componentes fundamentais e a de construção do modelo orientado a objetos do domínio de aplicação.

5.**Definição da base lingüística.** Na seção 4.3.4, apresentamos a estrutura lingüística de um patamar de reúso como o conjunto de linguagens que é oferecido

para a especificação de soluções de software dentro deste domínio. Estas linguagens são representadas por gramáticas, construídas a partir do zero ou através de composição de outras gramáticas já definidas no Gremlin.

6.**Evolução paradigmática**. Esta etapa compreende na aplicação contínua do paradigma do domínio de aplicação. Nesta etapa, as técnicas empregadas nas cinco etapas anteriores são continuamente aplicadas com o intuito de criar ou rever domínios, paradigmas, componentes e linguagens.

4.4. Interconexão de Linguagens

Para estabelecer uma baixa relação de custo/benefícios, um ambiente de reúso deve aumentar a generalidade e reduzir os custos de produção dos objetos da reutilização. Em ambientes de reutilização baseados em linguagens, o desenvolvimento da linguagem despende grande esforço de desenvolvimento, em um trabalho de sistematização de conceitos de domínios. Para que uma linguagem cumpra adequadamente o papel de força organizadora em um domínio de aplicação dinâmico, deve ter em si um forte caráter evolutivo.

Nesta seção, estamos preocupados com a reutilização da experiência de construção de uma linguagem de domínio, bem como em atender às suas necessidades de evolução a custos menores. A aplicação de técnicas da orientação a objetos, como a composição e a herança, para inter-conectar gramáticas e fragmentos de gramáticas traz vantagens em ambos os aspectos. A interconexão de linguagens constitui um ferramental simples e eficaz para produzir versões alternativas de linguagens, permitindo avaliação

comparativa, evolução incremental, prototipagem rápida. Além disto, possibilita a criação de componentes reutilizáveis que representam fragmentos de gramáticas que trazem em si sintaxe e semântica de tradução.

Nesta visão, a elaboração de uma linguagem de domínio passa a ser uma atividade fortemente interativa, baseada no paradigma de reúso compositivo.

4.4.1. Operações de Composição de Gramáticas

Para a definição das operações de composição, definiremos gramática como uma quádrupla **G = <s,i,P,S>**, onde **S** é o conjunto de símbolos que participam das produções da gramática, **P** é o conjunto das produções **{p1,p2,...}** da gramática, **i** $\notin$ **S** é o símbolo indefinido, e **s** $\in$ **S** $\cup$ **{i}** é o símbolo inicial da gramática. As gramáticas que possuem símbolo inicial indefinido (**s=i**) são chamadas gramáticas abstratas. Tais gramáticas podem participar de operações de composição mas não podem traduzir sentenças. As operações de composição são:

1.**Substituição de símbolo** - Esta operação substitui todas as ocorrências de um símbolo em uma gramática. Seja **G = <s,i,P,S>**, a troca de um símbolo **x** por um símbolo **k** em **G** definida como **G.Subs(x,k) = GS**, onde **GS = <z,i,P,S-{x}** $\cup$ **{k}>**, com **z = s** se **s**$\neq$**x** ou **z=k** se **s=x.**

2.**União** - Esta operação cria uma gramática como resultado da agregação de duas outras. Sejam $G_1 = <s_1,i,P_1,S_1>$, $G_2 = <s_2,i,P_2,S_2>$, a união $G = G_1 \cup G_2$ é definida como $G = <s,i, P_1 \cup P_2, S_1 \cup S_2>$, onde $s=s_1=s_2$ ou $s=i$.

3.**Interseção** - Esta operação cria uma gramática como resultado da interseção de duas outras. Sejam $G_1 = <s_1,i,P_1,S_1>$, $G_2 = <s_2,i,P_2,S_2>$, a interseção $G = G_1 \cap G_2$ é definida como $G = <s,i, P_1 \cap P_2, S_1 \cap S_2>$, onde $s=s_1=s_2$ ou $s=i$

4.**Diferença** - Esta operação cria uma gramática que retira de uma gramática as produções que ela tem em comum com outra. Sejam $G_1 = <s_1,i,P_1,S_1>$, $G_2 = <s_2,i,P_2,S_2>$, a diferença $G = G_1 - G_2$ é definida como $G = < s_1,i, P_1-P_2, S_1>$.

4.4.1.1. Um exemplo de composição de gramáticas

Vamos imaginar que disponhamos de uma gramática que aceite expressões matemáticas simples **G1** que desejamos alterar para receber parênteses. Neste caso, basta compor **G1** com **G2,** criando uma nova gramática **G3** como a união das duas.

```
G1:                G2:              G3 := G1 ∪ G2
    E:=E+T             P:=(E)           E:=E+T
      |T                                  |T
    T:=T*P                              T:=T*P
      |P                                  |P
    P:=ID                               P:=ID
                                          |(E)
```

4.4.2. Herança

A herança em gramáticas age como instrumento para vincular a evolução das gramáticas. Seja a especificação:

```
G1:
      E:=E+T
        |T
      T:=T*P
        |P
      P:=ID

G2 super G1:
      P:=(E)
```

Nesta definição **G2** possui todas as produções de **G1** e ainda `P:=(E)`.

A diferença entre a herança e a união está em que qualquer alteração em **G1** afetará igualmente **G2**. Sendo assim, unindo duas novas produções em **G1**

```
G1:
      E:=E+T
        |E-T
        |T
      T:=T*P
        |T/P
        |P
      P:=ID

G2 super G1:
      P:=(E)
```

Nenhuma operação adicional é necessária para que G2 aceite a expressão ID*(ID-ID).

No exemplo anterior (seção 4.4.1.1) seria necessário recalcular uma nova **G3.**

4.5. Conclusão

Neste capítulo apresentamos uma proposta para a reutilização de software orientada a domínios de aplicação. A proposta baseia-se no emprego de técnicas compositivas oriundas da orientação a objetos na construção de modelos genéricos sobre domínios de aplicação, cuja manipulação é regida por linguagens especializadas construídas para estes domínios. Esta proposta compreende a especificação de um modelo de objetos que dê suporte nativo a mecanismos transformacionais, e caracteriza um modelo de extensão do modelo baseado no enriquecimento do mecanismo de troca de mensagens. Sobre tal modelo híbrido elaboramos uma proposta de organização hierárquica de domínios. Cada domínio é dotado de uma base conceitual organizada em modelos de classes sob o paradigma da orientação a objetos. O modelo do domínio de aplicação pode ser manipulado através de linguagens especializadas que aproveitam o reúso compositivo em sua construção e o reuso gerativo em sua aplicação na criação de sistemas aplicativos.

5. A Ferramenta Gremlin

5.1. Propósito da Ferramenta

A ferramenta Gremlin foi desenvolvida para dar suporte a reutilização através do desenvolvimento de linguagens orientadas a domínios de aplicação. A ferramenta provê, ainda, suporte ao exercício da reutilização durante a construção da representação de cada domínio de aplicação, composta por modelos de reúso de domínio e por linguagens através das quais estes modelos são reutilizados.

Outro objetivo da ferramenta é a exploração do potencial da integração de tecnologias de reúso gerativo e compositivo para formar uma abordagem híbrida que concilia a vantagem de ambas. Na ferramenta Gremlin, o reúso compositivo está concentrado na construção dos patamares de reúso, representantes dos domínios de aplicação modelados. O reúso gerativo é exercido pelas linguagens definidas dentro do patamar de reúso.

A integração das duas formas de reúso é obtida por um modelo de objetos estendido que, em conformidade com os requisitos traçados na seção 4.2.2, amplia as características da orientação a objetos para dar suporte específico a transformações. Este capítulo discute aspectos de implementação deste modelo, e de que modo este se adequa à proposta apresentada no capítulo 4.

5.2. Organização da Ferramenta

A ferramenta Gremlin está dividida em sete subsistemas, que se relacionam como ilustrado na figura 5.1. Os subsistemas de documentação, de composição, lingüístico, e compositivo correspondem aos subsistemas especificados na seção 4.3.4. Os subsistemas específicos da implementação da ferramenta são o **modelo de objetos estendido,** o **gerente de objetos** e o **gerente de domínio**.

O modelo de objetos estendido implementa as adaptações ao paradigma da orientação a objetos particulares a esta ferramenta, tornando-o melhor adaptado ao uso de transformações. O gerente de objetos mantém em memória secundária os diversos tipos de objetos (documentos, classes e gramáticas) envolvidos no desenvolvimento e reutilização de software. Ele dá ainda, suporte básico aos mecanismos de classificação e localização de componentes exigidos pelo subsistema compositivo.

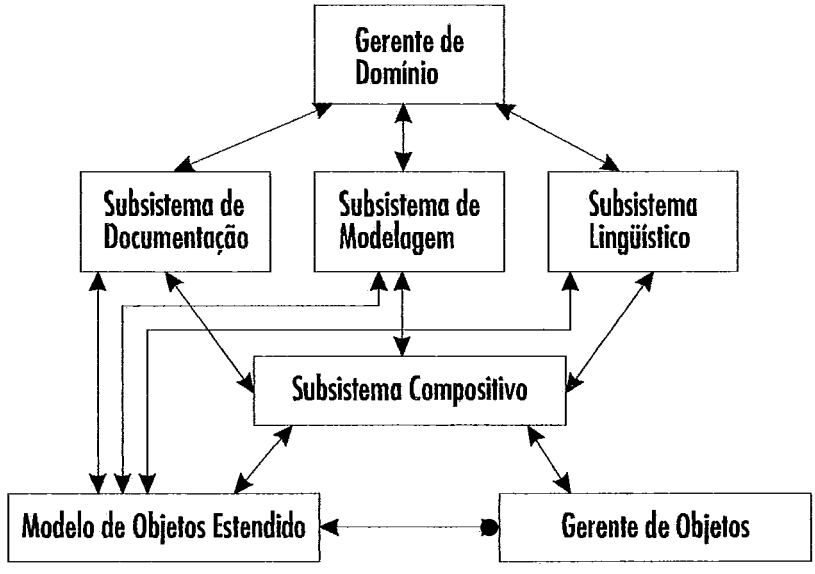


Figura 5.1 - Organização geral da Ferramenta Gremlin

Por fim, o gerente de domínio trabalha como agente organizador dos patamares de reúso, permitindo sua construção e uso.

5.3. Implementação da Ferramenta

5.3.1. Plataforma de desenvolvimento

A ferramenta foi desenvolvida na linguagem C++ [Stro93] sob o sistema operacional Windows 3.1[Micr94] em microcomputadores 486. A biblioteca de classes usada para a construção da interface foi a Microsoft Foundation Class 2.0 [Micr93].

A metodologia de desenvolvimento adotada foi a Análise Baseada em Objetos de Coad-Yourdon [Coad92] [Coad91].

5.3.2. Modelo de objetos estendido

5.3.2.1. Introdução

O modelo orientado a objetos estendido é a base de implementação de todos os subsistemas do Gremlin. Conforme descrito na seção 4.2.3, o modelo se baseia em alterações na arquitetura de troca de mensagens da orientação a objetos tradicional. Esta adaptação teve como objetivo aproximar a troca de mensagens dos processos de comunicação humanos, a fim de aumentar o poder de representação de cada mensagem.

O protocolo de um objeto define as mensagens que ele pode receber, e está definido em sua classe. No modelo de objetos do Gremlin, o protocolo das instâncias de uma classe é definida por uma **linguagem** que descreve a estrutura sintática das mensagens

que podem ser aceitas. Em nossa implementação, estas linguagens estão restritas àquelas geradas por gramáticas livres de contexto, LR(1). Mesmo com esta restrição, os protocolos das classes Gremlin são significativamente mais expressivos que os das classes de modelos orientados a objetos tradicionais, baseados em descrições enumerativas. Vale a pena ressaltar que todos os protocolos baseados na enumeração das mensagens podem ser descritos por linguagens LR(1). A recíproca, contudo, não é verdadeira.

A aceitação de mensagens baseadas em gramáticas transforma o receptor da mensagem em um tradutor, responsável por analisar a mensagem e, com base nesta análise, assumir um comportamento específico. Cada objeto Gremlin tem a capacidade de simular um autômato de pilha que reconhece a linguagem com base nas informações da classe a que este objeto pertence. Durante o processo de tradução, este autômato solicita à classe as produções que formam a gramática da linguagem reconhecida. A mensagem recebida é analisada a luz destas produções, de modo a determinar a ordem em que as produções são aplicadas para que a mensagem seja traduzida.

As produções de uma classe possuem um papel fundamental na caracterização do comportamento do objeto. Cada produção tem associado um script de instruções que são executadas na sua aplicação. O comportamento específico de um objeto assumido em decorrência da recepção de uma mensagem é a sinergia do comportamento de cada produção aplicada na interpretação da mensagem.

O modelo de objetos do Gremlin presta-se tanto a expressão do comportamento de objetos do domínio do problema quanto à definição de transformações. Como um super-conjunto dos modelos orientados a objetos tradicionais, o modelo do

Gremlin compartilha com eles o potencial de representação de objetos e arquiteturas de objetos dentro do domínio de aplicação [Cima94]. A representação de transformações vem através de scripts de produções que transformem os produtos intermediários do processo de tradução de uma mensagem.

A figura 5.2 mostra o modelo de classes do Gremlin.

5.3.2.2. Dois exemplos de classes do modelo estendido

O primeiro exemplo desta seção ilustra como classes do modelo estendido podem realizar implementações típicas da orientação a objetos. O segundo exemplo mostra um protocolo de mensagens que apenas pode ser implementado no modelo estendido.

Exemplo 1: Classes True e False

Todo ambiente de programação envolve alguma forma de manipulação de valores lógicos verdadeiro ou falso. O ambiente de desenvolvimento Smalltalk-80 implementa duas classes True e False para representar valores lógicos. Esta implementação permite que Smalltalk-80 não possua estruturas de controle de fluxo embutidas na linguagem. Estrutura como **if then else**, por exemplo são implementadas como funções polimórficas, com comportamento diferente em uma e outra classe.

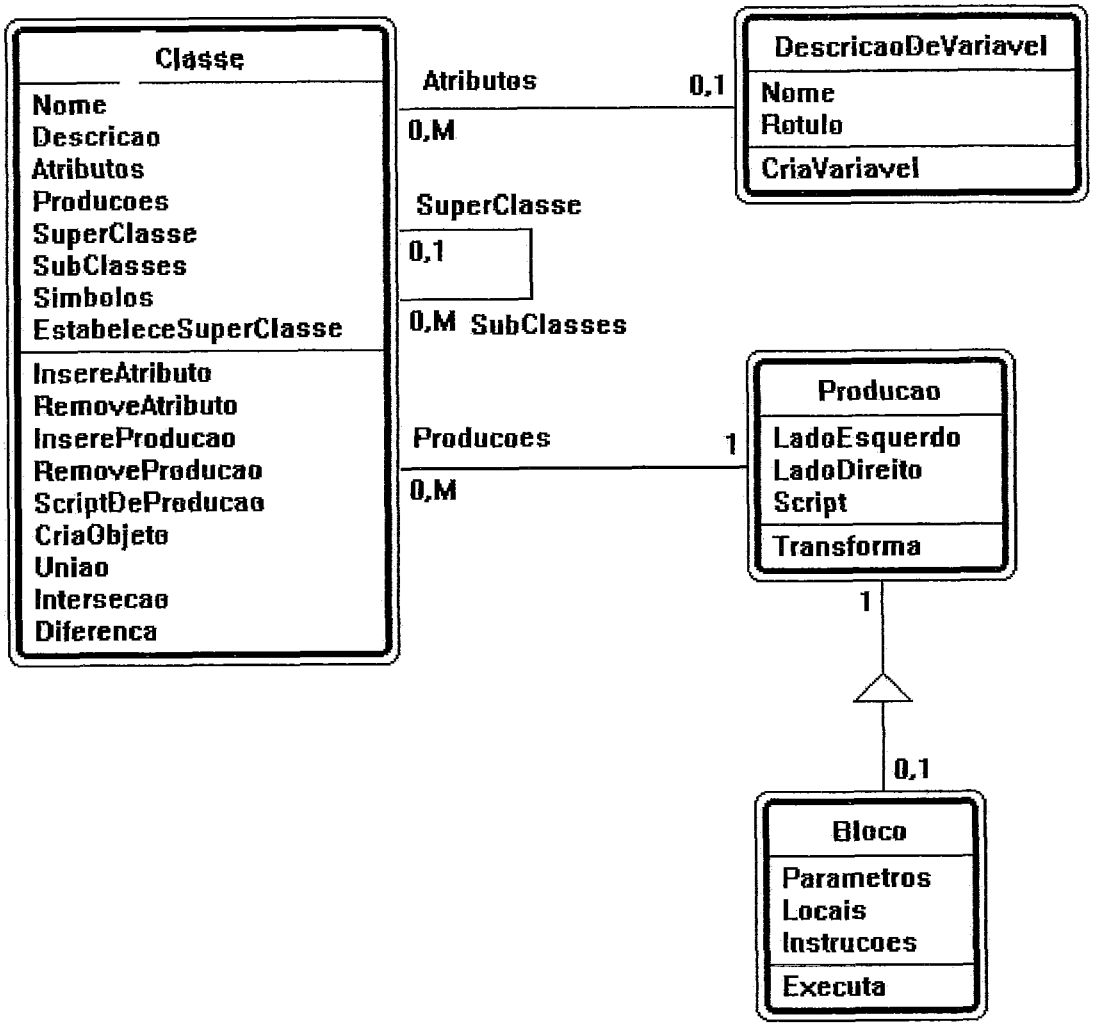


Figura 5.2 - Classes da Implementação do Modelo de Objetos do Gremlin - *Neste modelo o comportamento das classes é definido pelo conjunto de produções que descrevem seu protocolo. Cada produção possui um* ***script*** *que define o algoritmo ativado quando for aplicada. A funcionalidade* ***Transforma*** *é responsável por ativar este script.*

A listagem que se segue traz parte da implementação das classes **False** e **True** dentro do modelo de objetos estendido do Gremlin. A sintaxe da linguagem é derivada da linguagem CAOL [Wern93], de cuja implementação participamos. Um resumo da sintaxe pode ser encontrado no Anexo I

```
CLASS False
SUBCLASS OF Boolean
SYMBOLS_ARE
        If_True, If_False,And,Or,Not
INTERFACE_IS
        SELF -> If_True Block If_False Block;
        SELF -> If_True Block;
        SELF -> If_False Block;
        SELF -> And Boolean;
        SELF -> Or Boolean;
        SELF -> Not;
IMPLEMENTATION_IS
        SELF -> If_True Block If_False Block
        {
                Block[2] <- Execute;
        }
        SELF -> If_True Block
        {
        }
        SELF -> If_False Block
        {
                Block <- Execute;
        }
        SELF -> And Boolean
        {
                return SELF;
        }
        SELF -> Or Boolean
        {
                return Boolean;
        }
        SELF -> Not
        {
                LOCAL resp;
                resp := True new;
                return resp;
        }
```

```
END_CLASS

CLASS True
SUBCLASS_OF Boolean
SYMBOLS_ARE
        If_True, If_False,And,Or,Not
INTERFACE_IS
        SELF -> If_True Block If_False Block;
        SELF -> If_True Block;
        SELF -> If_False Block;
        SELF -> And Boolean;
        SELF -> Or Boolean;
        SELF -> Not;
IMPLEMENTATION_IS
        SELF -> If_True Block If_False Block
        {
                Block[1] <- Execute;
        }
        SELF -> If_True Block
        {
                Block <- Execute;
        }
        SELF -> If_False Block
        {
        }
        SELF -> And Boolean
        {
                return Boolean;
        }
        SELF -> Or Boolean
        {
                return SELF;
        }
        SELF -> Not
        {
                LOCAL resp;
                resp := False new;
                return resp;
        }
END_CLASS
```

As definição de cada classe é formada por 5 partes, que definem o nome da classe, os vínculos de herança, os símbolos reservados, a interface e a implementação da classe, respectivamente. O nome da classe a identifica dentro da hierarquia de classes.

A herança dentro do modelo estendido é em tudo análoga à encontrada em modelos de objetos tradicionais onde são herdadas estruturas de dados, interface e implementação. A interface de uma classes é constituída por suas produções e pelos símbolos reservados. O papel dos símbolos reservados é distinguir que símbolos das produções são palavras reservadas da linguagem.

A implementação das produções são procedimentos que recebem um parâmetro para cada símbolo da produção que não seja reservado. Assim, a produção *SELF -> If_True Block* recebe exatamente um parâmetro, de nome *Block*. *If_True* não é um parâmetro por que consta na lista de símbolos reservados da classe. Quando um símbolo ocorre mais de uma vez na produção, os parâmetros são qualificados por um número inteiro, como ocorre com a produção *SELF -> If_True Block If_False Block* que recebe dois parâmetros, chamados *Block[1]* e *Block[2]*.

As gramáticas destas classes são polimórficas, porque reconhecem a mesma linguagem com comportamentos diferentes (ver seção 4.3.4.3). Vejamos, por exemplo, o comportamento obtido pela expressão:

Teste If_True [a:=b;] If_False [a:=c;];

O receptor da mensagem é Teste, uma variável. Se esta variável contiver uma instância da classe True, a aplicação da produção SELF -> If_True Block If_False

Block da classe True será invocada, executando Block[1], no caso, [a:=b;]. Se, ao contrário, a variável contém o valor falso, a produção ativada será a da classe False, que executa o bloco [a:=c;].

As classes deste exemplo são muito semelhantes às suas equivalentes em Smalltalk-80. O exemplo que se segue mostra uma gramática com mensagens mais complexas.

Exemplo 2: Classe Calculadora

A fim de ilustrar as características particulares do modelo estendido, exploramos um exemplo clássico da literatura de compiladores [Holu90]. Os objetos da classe *Calculadora* recebem mensagens na forma de expressões aritméticas munidas das quatro operações e de parênteses. As operações multiplicativas * e / tem precedência sobre as aditivas + e -.

```
CLASS Calculadora
UNION_WITH Lex_Number
SYMBOLS_ARE
        +,-,*,/,(,)
INTERFACE_IS
        SELF  ->      E;
        E     ->      E + T;
        E     ->      E - T;
        E     ->      T;
        T     ->      T * P;
        T     ->      T / P;
        T     ->      P;
        P     ->      Lex_Number;
        P     ->      (E);
IMPLEMENTATION_IS
        SELF  ->      E
        {
                return E;
        }
```

```
        E       ->      E + T
        {
                LOCAL resp;
                resp := E + T;
                return resp;
        }
        E       ->      E - T
        {
                LOCAL resp;
                resp := E - T;
                return resp;
        }
        E       ->      T
        {
                return T
        }
        T       ->      T * P
        {
                LOCAL resp;
                resp := T * P;
                return resp;
        }
        T       ->      T / P
        {
                LOCAL resp;
                resp := T / P;
                return resp;
        }
        T       ->      P
        {
                return P;
        }
        P       ->      Lex_Number
        {
                LOCAL resp;
                resp := Lex_Number value;
                return resp;
        }
        P       ->      (E)
        {
                return E;
        }
END_CLASS
```

A tabela 5.1 mostra uma seqüência típica de ativação dos métodos da classe para a mensagem *3+4*. Nesta seqüência, vamos nos deter nos passos 4 e 6, por serem representativos dos efeitos da interconexão de linguagens e da transformação de objetos.

O passo 4 é uma cooperação de duas classes inter-conectadas, a saber, *Calculadora* e *Lex_Number*. A classe *Lex_Number* é responsável por interpretar os numerais da mensagem recebida, inserindo-os na pilha operacional do autômato que controla a interpretação da mensagem. O restante do processamento fica a cargo da classe Calculadora. Esta cooperação é especificada pela cláusula *UNION_WITH Lex_Number* na declaração da classe. A figura 5.3 ilustra a operação de interconexão expressa nesta linha e na implementação da produção *P->Lex_Number* .

Ordem de aplicação	Produção ativada	Pilha do autômato
1	P->Lex_Number	3
2	T->P	3
3	E->P	3
4	P->Lex_Number	3 + 4
5	T->P	3 + 4
6	E->E+T	7
7	SELF->E	7

Tabela 5.1 - Ordem de aplicação de produções para objeto da classe Calculadora

O passo 6 aplica uma transformação, ativando a produção *E->E+T*. O script da produção é ativado com os parâmetros *E* e *T* com *3* e *4*. O valor retornado à pilha operacional é a soma dos argumentos.

5.3.2.3.Implementação do mecanismo de tradução

Os mecanismos de tradução convencionais [Holu90] [Fish88] baseiam-se na tradução de especificações de linguagens em tabelas organizadas para orientar algoritmos especificamente responsáveis pela tradução.

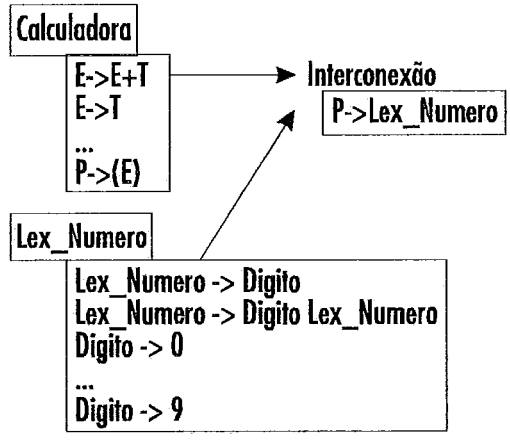


Figura 5.3 - Interconexão de classes e, conseqüentemente, de linguagens - *Neste exemplo a classe Calculadora é vista em sua forma original, que implementa apenas o padrão de precedência entre as operações. Esta forma genérica é conectada a uma classe específica que a supre com a natureza dos números que integram as mensagens.*

Para dar maior flexibilidade aos mecanismos de interconexão de classes, a estratégia de tradução adotada em Gremlin difere das tradicionais por não necessitar de um processo de transformação global da linguagem para que a tradução se torne operacional.

O mecanismo de tradução é dividido em duas partes, como indica a figura 5.4. A parte geral é fixa e está implementada em uma classe denominada **Gramática**. A gramática é responsável por todo o mecanismo de simulação do autômato de pilha que realiza a tradução. Sempre que este mecanismo precisa de qualquer informação sobre a sintaxe e semântica da linguagem, ele usa o conjunto de produções para determiná-las. Cada produção é um objeto independente que encapsula uma parte da sintaxe da linguagem e a semântica de tradução correspondente. Cada produção se conforma a um protocolo único, de modo que a gramática processa todas as produções de um mesmo modo.

Figura 5.4 - *O processo de tradução possui uma parte geral implementada no objeto gramática que usa o conjunto de produções através de um protocolo comum, de modo a depreender a sintática e semântica da linguagem.*

Uma vez que não há tabelas, o autômato precisa armazenar cada estado por que passa, na forma de um conjunto relativamente complexo de informações. Este conjunto é representado por objetos da classe **EstadoDeTradução**. O estado tem acesso à pilha operacional e às produções da gramática de modo que é autônomo para criar o objeto que representa o estado que o sucede na tradução, diante de um símbolo da sentença. O processo de tradução é, portanto, uma seqüência de criação e destruição de objetos que representam os estados pelos quais passa o autômato de pilha (figura 5.5). Durante o

processo de sucessão de estados, a pilha operacional é passada de um para o outro, e é a única forma de troca de informações entre um estado e seu sucessor.

A troca de estados se dá cada vez que a pilha operacional é alterada pela inclusão ou exclusão de símbolos em seu topo. O novo estado passa a representar o conjunto de produções que são compatíveis, ao mesmo tempo, com o conteúdo atual da pilha operacional e com o símbolo da sentença em tradução que está sendo processado.

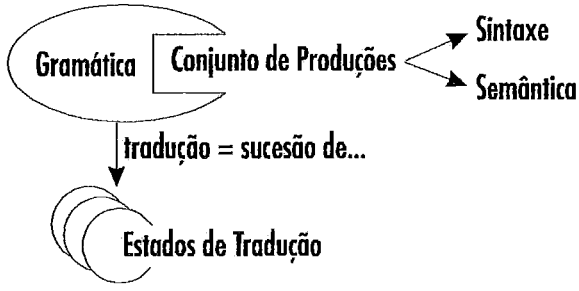


Figura 5.5 - *No processo de tradução, objetos da classe EstadoDeTradução se sucedem na caracterização do estado do autômato de pilha. Cada estado é um objeto atuante no processo de tradução, realizando a construção de novos estados e deixando a pilha operacional disponível para as produções à medida que elas são aplicadas.*

Um estado também é representado como um objeto complexo. Cada produção compatível com a pilha operacional e com o símbolo da entrada é incluída no estado na forma de uma instância da classe **Possibilidade.**

Uma possibilidade se refere diretamente à produção que descreve, e indica ainda quantos símbolos compatíveis com ela estão na pilha operacional (figura 5.6). Uma possibilidade representa também a expectativa que um símbolo (o que sucede a posição) seja inserido na pilha para que continue válida. Este símbolo é chamado *símbolo esperado por uma possibilidade.*

A ocorrência de uma troca de estados pode ocorrer em dois casos:

1.Um novo símbolo é empurrado para a pilha

2.Uma possibilidade indica que todos os símbolos necessários para a aplicação de uma produção estão na pilha.

No primeiro caso, o novo estado é construído a partir das possibilidades que continuem compatíveis com a nova pilha operacional. Tais possibilidades são "avançadas", isto é, sua posição é aumentada em um. Todas as produções da gramática que iniciem pelo símbolo esperado de qualquer possibilidade do estado também são inseridas no estado, em uma operação transitiva.

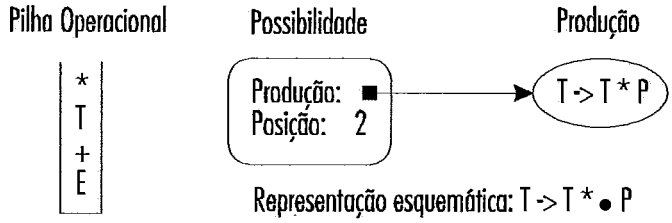


Figura 5.6 - *Uma Possibilidade é um objeto que integra um estado da tradução para indicar que uma produção poderá vir a ser aplicada, e o quanto de seu lado esquerdo já está na pilha operacional.*

No segundo caso de troca de estados, uma produção é aplicada. O estado localiza a possibilidade completa para, através dela, identificar a produção que será responsável pela tradução. Esta produção recebe a pilha operacional como o parâmetro de uma mensagem polimórfica **Transforma**. O polimorfismo permite que esta mensagem ative o seu mecanismo particular de tradução, de modo que a pilha operacional possa ser manipulada de acordo. A produção retira da pilha os objetos associados aos símbolos de seu lado direito e executa um algoritmo que sintetizará um representante único para a sua semântica. Este objeto é retornado à pilha operacional associado ao seu lado esquerdo. O estado em que o autômato volta é, neste caso, o estado em que a possibilidade foi inicialmente considerada.

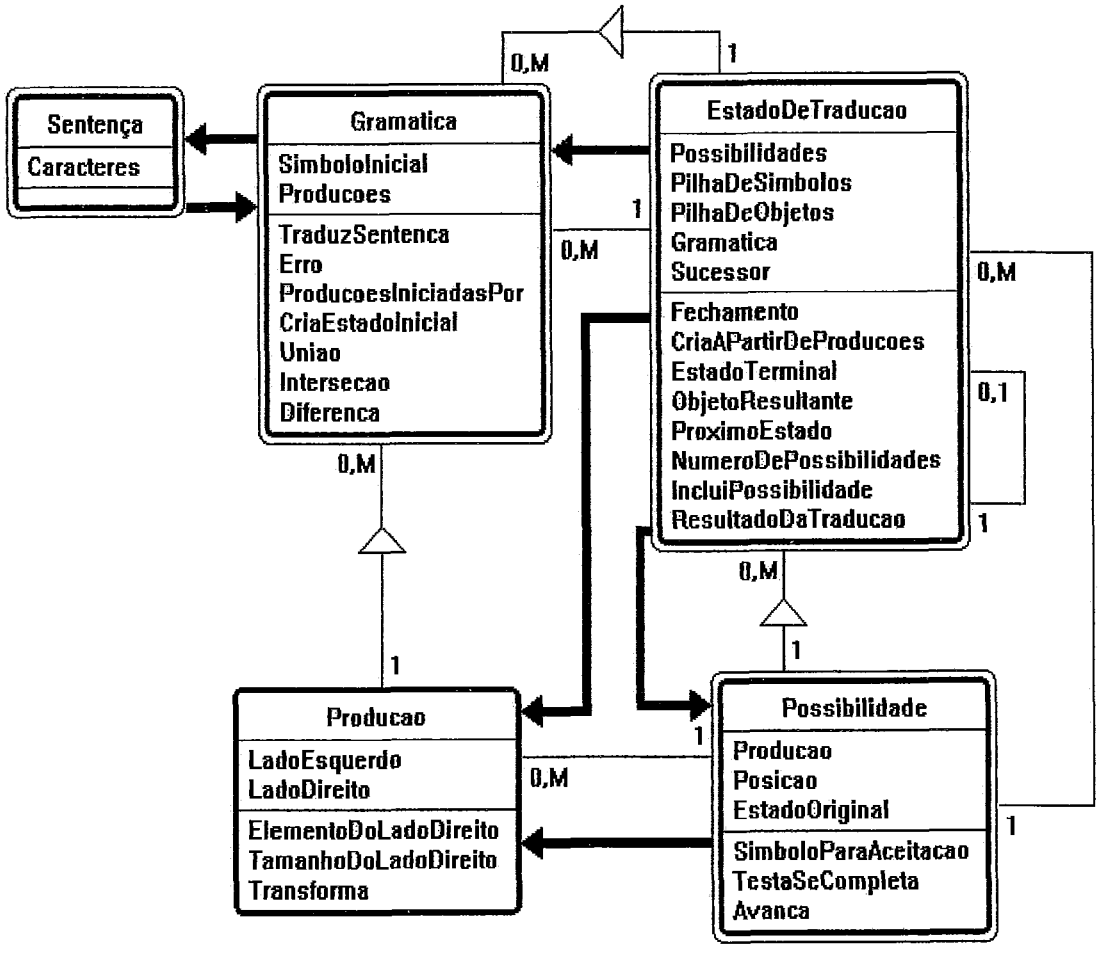


Figura 5.7 - *Diagrama de classes que participam do processo de tradução.*

Este mecanismo é essencialmente o mesmo que rege a tradução de linguagens LL(1), revisto em uma abordagem em que a orientação a objetos permite a fácil interpretação da gramática, ao invés de sua tradução. A figura 5.7 mostra a hierarquia das classes descritas nesta seção, usando a notação de [Coad91] [Coad92].

Os serviços fundamentais do processo são elementares o suficiente para tornar simples a tarefa de otimizar o mecanismo para a produção antecipada da seqüência de estados passíveis de representação condensada quando desejado.

5.3.2.4. Um exemplo do mecanismo de tradução

Nesta seção mostramos, passo a passo, a tradução de uma mensagem enviada a uma instância da classe *Calculadora* apresentada na seção 5.3.2.2. A figura 5.8 mostra a mensagem enviada ao objeto.

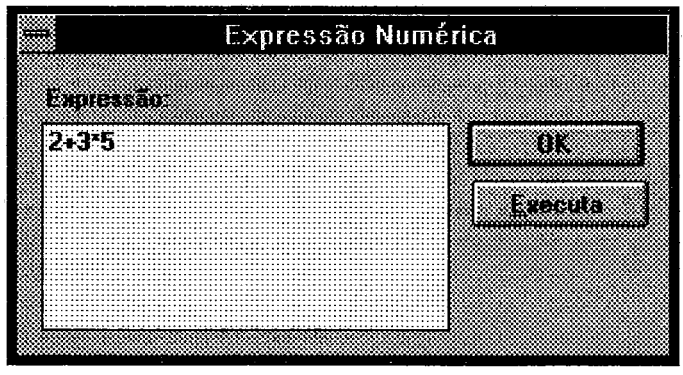


Figura 5.8 - *Mensagem enviada a uma instância da classe* ***Calculadora.***

Na figura 5.9 vemos um diálogo da ferramenta Gremlin, usado para a visualização do estado do autômato a cada passo da interpretação da mensagem. Em cada diálogo, vemos quatro campos que descrevem o estado. O primeiro é o número serial que identifica este estado particular do autômato. Os outros três campos são listas de elementos. A lista identificada como "Possibilidades" mostra as possibilidades de aplicação das produções. Como descrevemos na seção anterior, cada possibilidade é representada por um objeto que indica a produção considerada e o quanto de seu lado esquerdo se encontra na

pilha operacional. Para facilitar a visualização da possibilidade, usamos o caractere ^ para indicar, no lado esquerdo da possibilidade, o que já está na pilha operacional. Na figura 5.9, a pilha operacional está vazia, e todas as produções possíveis possuem a marca ^ no início do lado esquerdo.

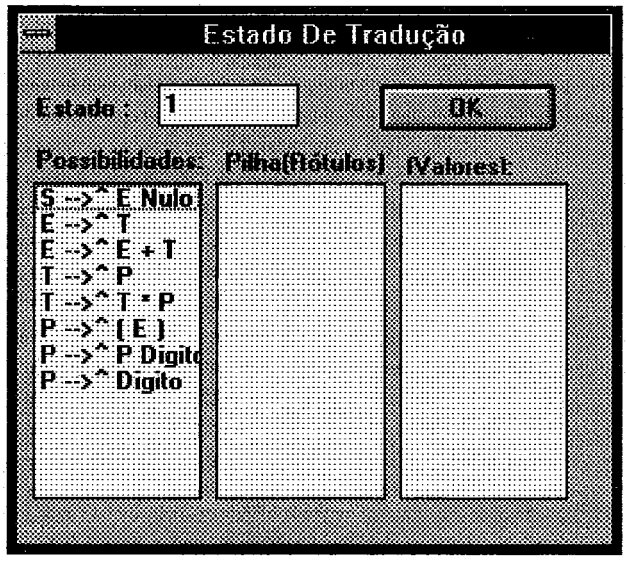


Figura 5.9 - Estado inicial do autômato de tradução.

A pilha operacional é representada pelas duas listas finais do diálogo. A primeira lista os símbolos da pilha operacional, e a segunda, os objetos a eles associados. Os símbolos são usados na seleção das produções que serão aplicadas. Os objetos são resultados intermediários do processamento. São estes objetos os argumentos e resultados dos scripts das produções.

A figura 5.9 mostra o estado inicial do autômato, criado pelo fechamento do símbolo inicial da gramática S (Self). O primeiro passo do autômato é processar o primeiro caractere da sentença, "2". Este caractere, rotulado como dígito, só é compatível

com a possibilidade *P* --> ^ *Digito*, de modo que o estado seguinte, de número 2, é o que aparece na figura 5.10.

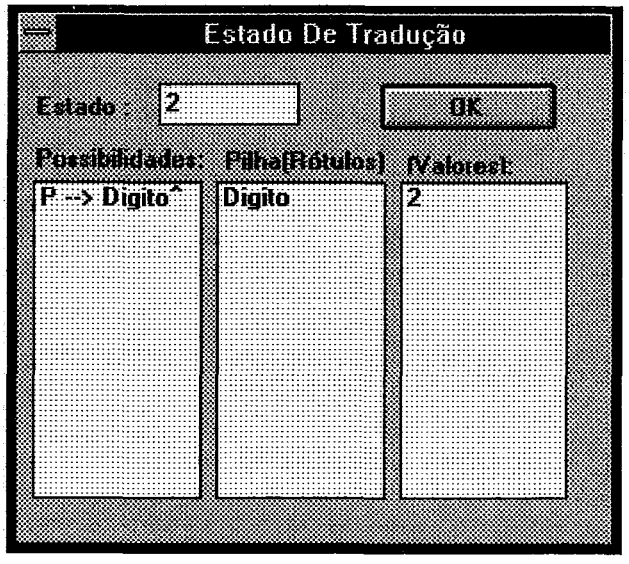


Figura 5.10 - Segundo estado do autômato de tradução.

No estado 2, o caractere 2 foi posto na pilha operacional com o rótulo dígito, e a possibilidade foi avançada de acordo. Neste ponto, temos uma produção totalmente compatível com o conteúdo da pilha operacional. Além disto, não há qualquer outra produção compatível com a pilha operacional atual, já que o estado possui apenas uma possibilidade. Neste ponto, o autômato concluiu que esta produção deve ser aplicada. O script da produção *P* --> *Digito* é aplicado para transformar o caractere "2" no número 2. Esta transformação, realizada na pilha operacional é acompanhada da troca do rótulo do objeto. A seguir, o autômato retorna ao estado onde a produção *P* --> *Digito* foi considerada em primeiro lugar, isto é, ao estado 1. A figura 5.11 ilustra o autômato após a aplicação desta produção.

Figura 5.11 - Estado do autômato após a aplicação da primeira produção.

Voltando ao estado 1, percebemos que o símbolo *P* na pilha é compatível com duas possibilidades, *T->P* e *P->P Digito*. O autômato avança, então, para o estado 3, onde ambas as possibilidades são consideradas (figura 5.12). A decisão entre elas é tomada com base no próximo caractere da mensagem, "+". O exame deste caractere desqualifica a segunda possibilidade, de modo que temos uma nova aplicação de produção,. Desta vez a produção aplicada é *T->P*. Novamente voltamos ao estado 1, e o processo se repete até que o símbolo *E* esteja na pilha (figura 5.13).

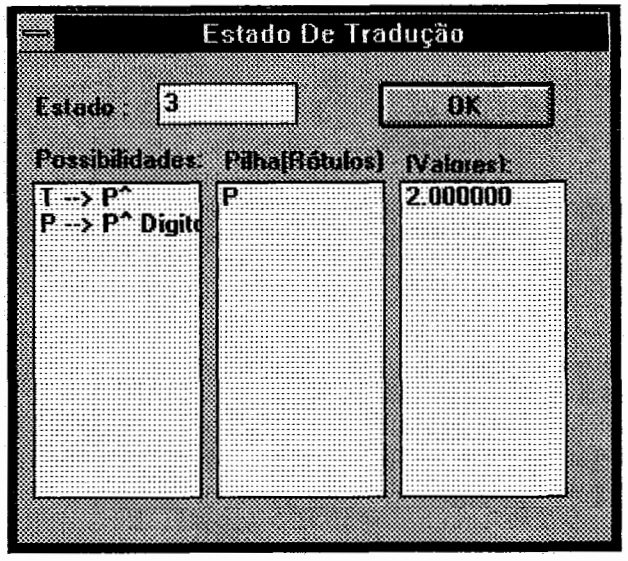


Figura 5.12 - Decisão entre produções com base no próximo elemento da mensagem.

Figura 5.13 - Após algumas trocas de estado, o número 2.0 é rotulado como E na pilha operacional.

Novamente, o símbolo E é compatível com duas possibilidades, de modo que o estado seguinte, o de número 4, fica como na figura 5.14.

Figura 5.14 - Estado 5 do autômato de tradução

A evolução deste estado difere das anteriormente descritas. Agora, a produção $E->E+T$ é compatível com o caractere "+" da entrada, de modo que esta possibilidade é considerada para o próximo estado, o de número 6, representado na figura 5.15. Neste estado, a operação de fechamento inclui uma possibilidade para cada produção iniciada em T ou P. Vale a pena lembrar que estas possibilidades estarão sempre vinculadas ao estado número 6, de modo que, quando qualquer delas for aplicada, o estado 6 é retomado pelo autômato. Isto acontece no estado número 12, ilustrado na figura 5.16. Nele, a expressão foi totalmente transferida para a pilha operacional, e a possibilidade $T->T*P$ está completa. A aplicação desta produção ativa o script que multiplica os dois números

presentes na pilha operacional e insere o resultado na pilha com o rótulo P . Após esta aplicação, o estado 6 é retomado, como mostra a figura 5.17.

Figura 5.15 - Estado em que há inclusão de novas possibilidades pela operação de fechamento.

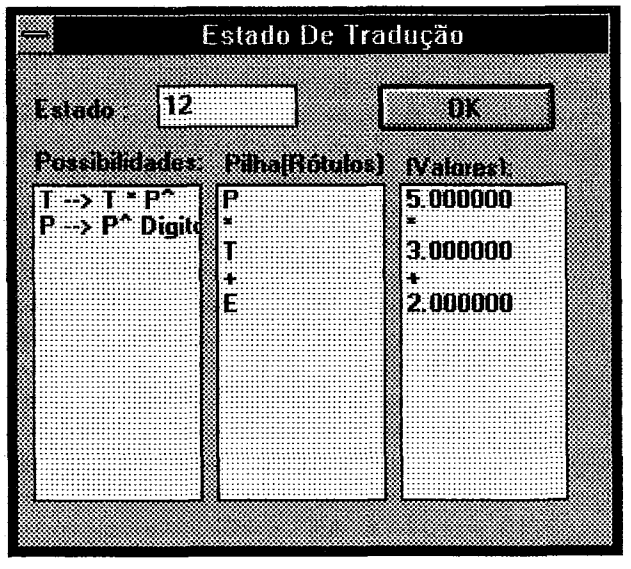


Figura 5.16 - Estado que precede a aplicação da produção T->T*P

Figura 5.17 - Após a multiplicação, o estado número 6 é retomado

O autômato passa por um total de quinze estados até atingir o estado final da tradução, ilustrado na figura 5.18. O resultado da mensagem é 17,0.

Figura 5.18 - Estado final da tradução da mensagem. Resposta 17,0.

5.3.3. O Protótipo do Gremlin

O modelo de objetos estendido foi tornado operacional em um protótipo da ferramenta Gremlin que incorpora os principais aspectos discutidos neste capítulo. A interface do Gremlin aderem ao padrão MDI (Multiple Document Interface) [Micr94]. Este padrão baseia a interface em uma janela principal, dentro da qual janelas auxiliares permitem a manipulação de documentos. Cada documento é representado por um objeto, cuja funcionalidade está disponível através do menu principal da ferramenta, ou dos botões que representam as ações mais freqüentemente tomadas. Estes botões ficam na barra de ferramentas, logo abaixo do menu da janela principal.

A ferramenta foi estruturada para lidar com documentos de quatro tipos:

- **Documentos** - Armazenados em arquivos de extensão (.txt), são textos livres usados como elemento do subsistema de documentação.

- **Classes** - Armazenados em arquivos de extensão (.cls), cada documento deste tipo descreve uma classe do modelo estendido de objetos, usadas para integrar o subsistema de modelagem.

- **Linguagens** - Armazenados em arquivos de extensão (.lin), cada linguagem é uma classe do modelo de objetos estendido especializado na tradução de especificações textuais.

- **Patamar de Reúso - Armazenado em arquivos de extensão (.pre), o patamar de reúso agrega documentos dos demais tipos, conferindo**

a eles estrutura do domínio de aplicação. O patamar de reúso é responsável pela classificação dos diversos tipos de documentos, pelo mecanismo de consulta e seleção, e pela rede que representa as interconexões entre os componentes.

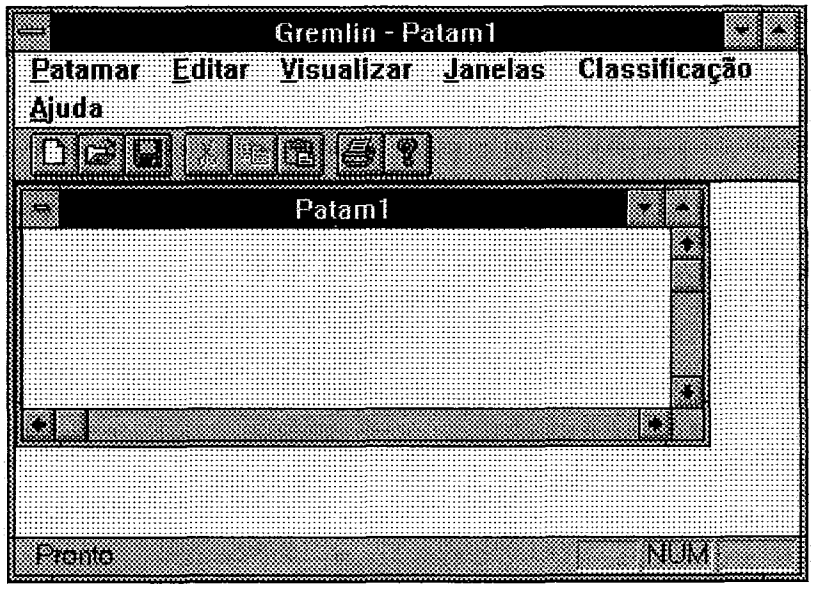
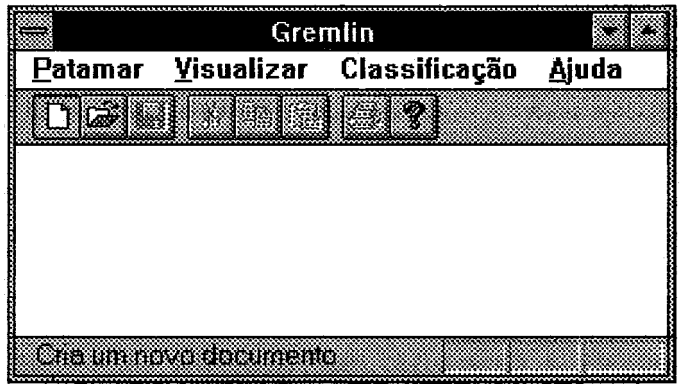


Figura 5.19 - Janela principal do protótipo do Gremlin

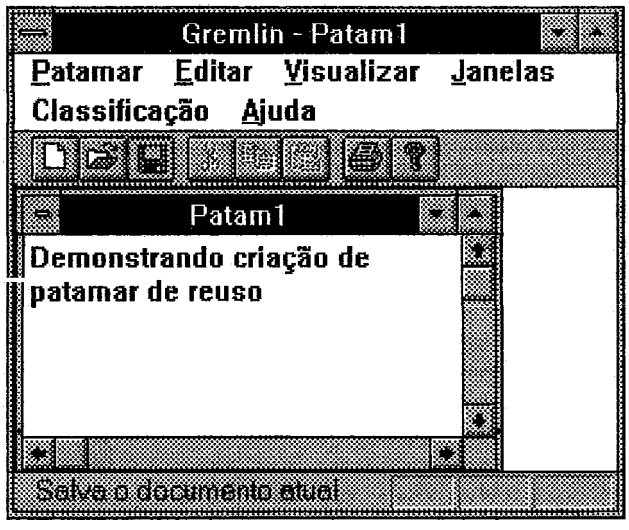
A interface foi concebida para ser simples e intuitiva. Os menus que caracterizam cada um dos documentos obedecem a um padrão de estrutura e funcionalidade. As operações comuns a mais de um objeto (como abrir, salvar ou criar) são sempre exercidas do mesmo modo. A figura 5.20 ilustra a criação de um patamar de reúso.


Gremlin
Patamar Visualizar Classificação Ajuda
Cria um novo documento

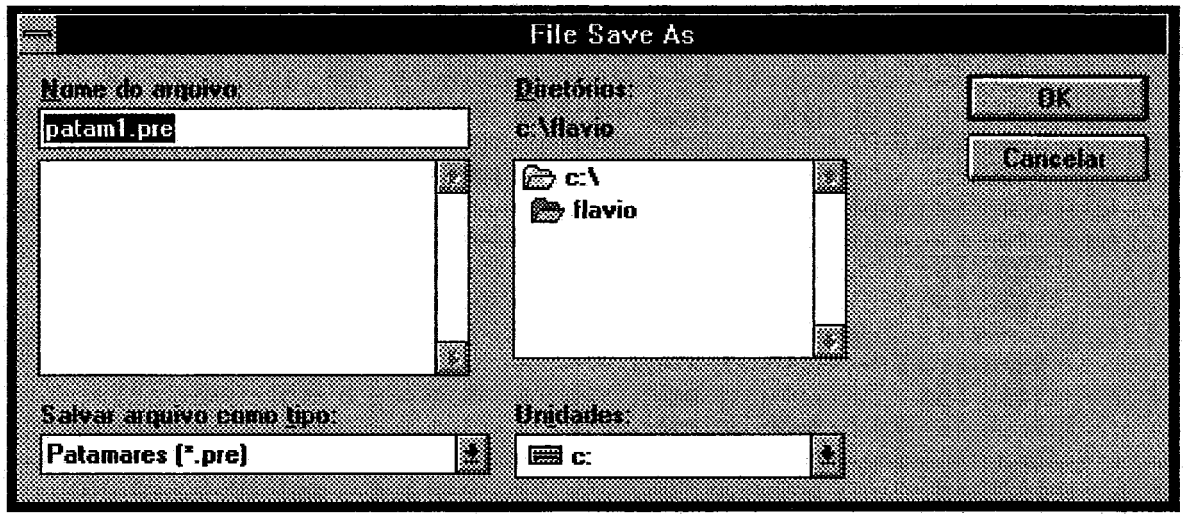
(a)


Gremlin
Patamar Visualizar Classificação Ajuda
New
New
Patamar de Reuso
Documento
Classe
Linguagem
OK
Cancel

(b)


Gremlin - Patam1
Patamar Editar Visualizar Janelas
Classificação Ajuda
Patam1
Demonstrando criação de
patamar de reuso
Salva o documento atual

(c)



(d)

Figura 5.20 - Exemplo de criação de um patamar - *Na figura (a) acionamos o botão "Novo documento". A ferramenta pergunta, então, em que tipo de documento estamos interessados (b). O documento é exibido em uma janela que permite a edição direta do texto que descreve, em linhas gerais, o domínio de aplicação (c). Para concretizar a criação do patamar de reúso, importa salvá-lo em disco. Um dos modos de fazer isto é escolhendo o botão "Salvar" da barra de ferramentas. O nome do arquivo que armazenará as informações do domínio é solicitado por um diálogo padrão do Windows (d).*

Quando um documento de qualquer tipo é ativado, o menu da ferramenta altera-se para representar as operações disponíveis para sua manipulação (figura 5.21).

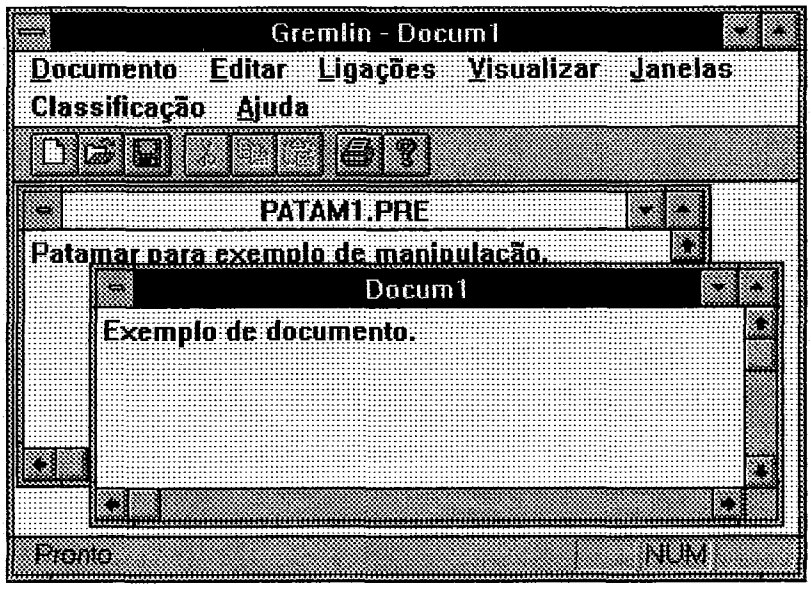


Figura 5.21 - Menus especializados por tipos de documento.

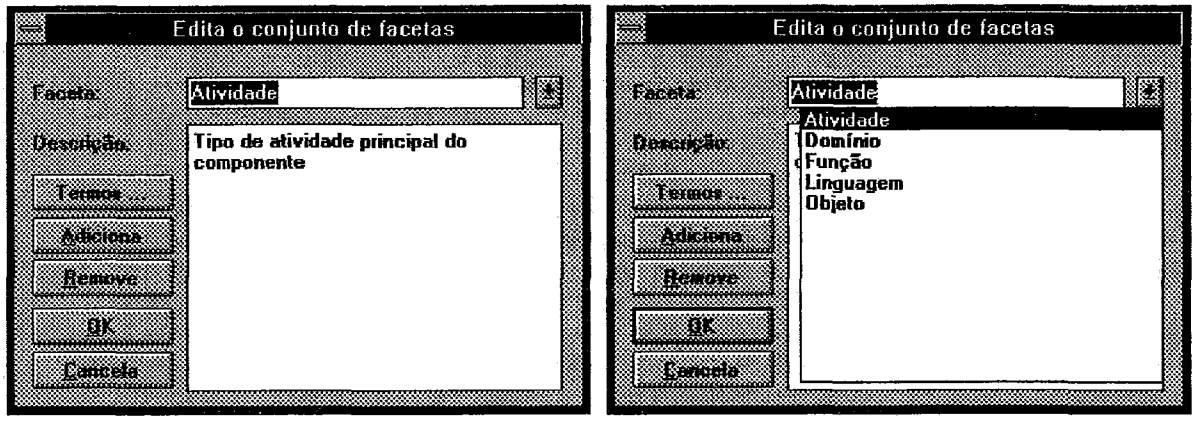


Figura 5.22 - Diálogo de edição de facetas - *À esquerda vemos uma faceta sendo examinada ou alterada. À direita, a lista de facetas do domínio está aberta.*

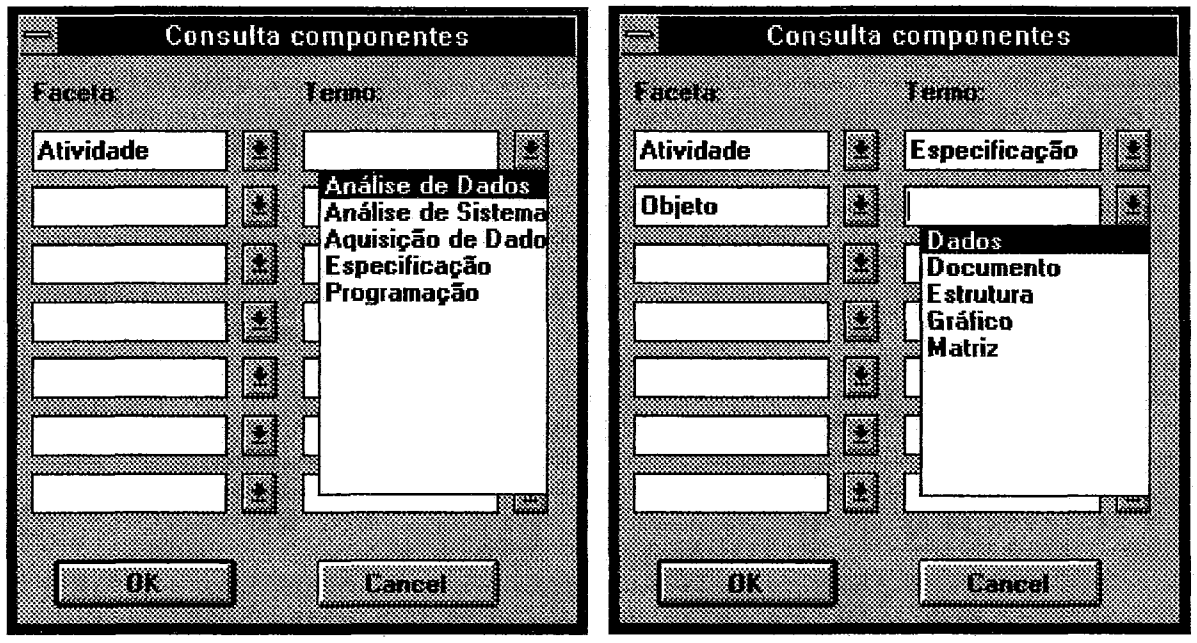


Figura 5.23 - Diálogo de seleção de componente - *Na coluna esquerda escolhemos as facetas que integram a consulta. Escolhida a faceta, os termos desta tornam-se valores válidos para o valor da coluna esquerda. Como resultado da consulta, todos os documentos classificados com a combinação escolhida de facetas e termos são abertos.*

A classificação facetada de componentes está disponível a partir do menu de classificação. A partir dele podemos criar facetas e termos dentro do domínio de aplicação, classificar componentes e fazer consultas. A figura 5.22 mostra o diálogo de criação de facetas, e a figura 5.23 mostra um diálogo em que especificamos critérios de seleção de um componente.

No protótipo, a composição de classes e linguagens é exercida em editores especializados, aptos a realizar a tradução de especificações descritas no modelo orientado a

objetos estendido em forma executável. A figura 5.24 mostra uma seção do Gremlin com um documento de cada tipo ativo.

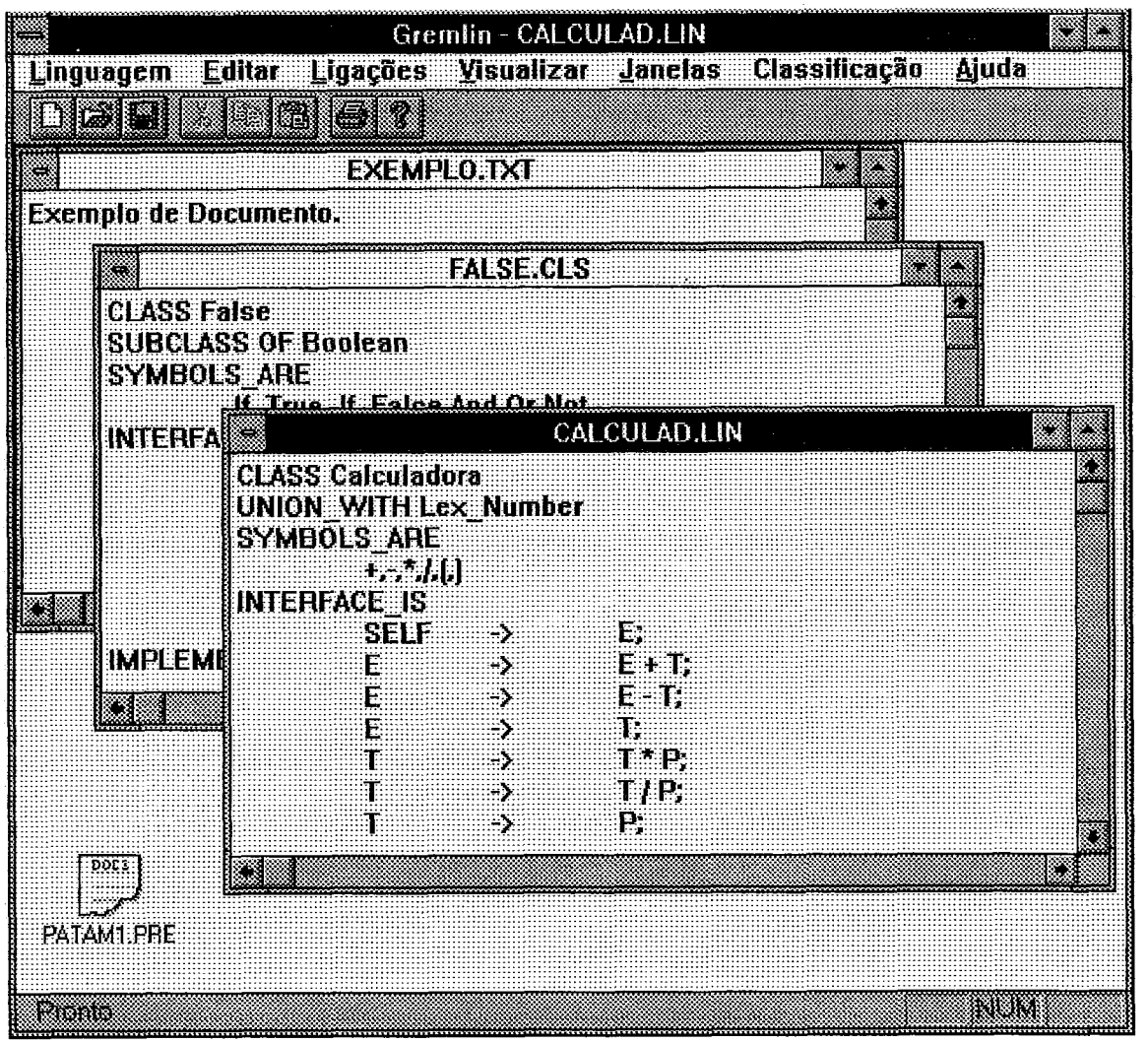


Figura 5.24 - Seção típica do protótipo do Gremlin

5.4. Conclusão

Neste capítulo descrevemos os principais aspectos da implementação da ferramenta Gremlin, discutindo a organização geral da ferramenta, e entrando em detalhes sobre os mecanismos através dos quais um modelo de objetos estendido foi desenvolvido.

6. Conclusões

6.1. Considerações finais

O objetivo desta tese foi discutir o potencial da integração de técnicas de reutilização compositiva e gerativa em um contexto de reutilização orientada a domínios de aplicação, propondo uma estratégia para esta integração e uma ferramenta que lhe dê suporte.

Fez-se, inicialmente, um estudo da reutilização, com ênfase no potencial que as tecnologias orientadas a objetos oferecem a ela. Este estudo revelou que as abordagens orientadas a objetos tornam-se progressivamente mais poderosas para a reutilização de software, graças ao amadurecimento conceitual e tecnológico que advêm da pesquisa de novas soluções desta área.

Uma das principais contribuições deste trabalho foi a elaboração do modelo de objetos estendido, que enriquece o paradigma da orientação a objetos, aumentando seu poder de representação de transformações. Este modelo traz em si uma plena integração das técnicas de natureza eminentemente compositiva da orientação a objetos e das técnicas eminentemente transformacionais envolvidas na tradução de sentenças em representações computacionais. Esta integração é plenamente possível na medida em propomos o uso de linguagens para descrever o protocolo de uso de objetos.

Outra contribuição está na proposta de organização de domínios de aplicação como ambientes baseados neste modelo de objetos estendido. Estas

representações, chamadas patamares de reúso, modelam a informação a ser utilizada em uma arquitetura de classes, disponível para reúso através de linguagens especializadas neste domínio. O uso do modelo de objetos estendido permite que modelos e linguagens sejam criados a partir de componentes de mesma natureza, classes.

Outro ponto levantado nesta tese é a necessidade do exercício de reutilização nas atividades de caracterização do domínio de aplicação. Para isto foi proposto o suporte genérico ao reúso compositivo, através de mecanismos de classificação [Prie85] e recuperação de componentes [Wern93], e do estabelecimento de padrões para o exercício da integração destes componentes.

Por último, o trabalho contribui com uma visão da evolução de domínios de aplicação baseada em uma analogia com a visão de Kuhn [Kuhn77] para a evolução da ciência. Esta analogia indica que cada domínio particular possui um paradigma próprio de investigação, que se sustenta enquanto as soluções construídas neste domínio são percebidas como adequadas. Um novo domínio apenas surge da percepção de que algumas soluções construídas são representadas de modo inadequado em um domínio.

6.2. Sugestão para trabalhos futuros

O trabalho desenvolvido nesta tese serve como base para discussão de tecnologias de suporte à reutilização baseada na modelagem de domínios de aplicação. Além da proposta de integração tecnológica da orientação a objetos a abordagens transformacionais, está a convicção da fundamental importância da integração de abordagens compositivas e gerativas para o reúso.

Esta tese apresenta contribuições a diversos aspectos da reutilização baseada em domínios. Os possíveis desdobramentos deste trabalho situam-se em três níveis, a saber, no processo de percepção e estruturação de domínios, na tecnologia de construção de linguagens e no modelo orientado a objetos estendido.

O modelo orientado a objetos implementado na ferramenta Gremlin pode ser refinado tanto em sua implementação quanto em suas bases conceituais. O principal problema apresentado neste modelo foi a penalidade de desempenho introduzida pela simulação de autômato de pilha em cada troca de mensagem. A redução deste custo poderá ser encontrada através da investigação de mecanismos para antecipar para o tempo de compilação o maior número possível de decisões sobre a interpretação das mensagens. Outro ponto que merece investigação é o desenvolvimento de mecanismos de verificação de tipos adaptado à especificação de interações através de linguagens. A existência de tais mecanismos é indispensável para determinar a validade das operações de composição exercidas sobre o modelo estendido.

Propomos que a continuidade do desenvolvimento das bases conceituais do modelo de objetos do Gremlin investigue a caracterização formal do modelo e o estabelecimento de critérios de qualidade para o desenvolvimento nestas bases.

Dois pontos de investigação sugeridos são o estudo de novas formas de representar transformações e da exploração de paralelismo na tradução de mensagens.

Em sua versão atual, o modelo de objetos estendido representa transformações de modo procedimental, associando a cada produção um comportamento

específico que realiza um passo da transformação. O desenvolvimento de uma base conceitual não procedimental para a representação de transformações facilitará a especificação de transformações reutilizáveis, minimizando efeitos colaterais e facilitando a verificação da correção da composição.

A aplicação de paralelismo no processo de tradução de mensagens encapsulará o comportamento paralelo sob a interface bem definida da linguagem do receptor da mensagem. O processo de tradução passará, também, a especificar padrões de interação entre processos paralelos.

No âmbito das técnicas de construção de linguagens, é fundamental o desenvolvimento de uma abordagem gráfica para melhorar a legibilidade das descrições das linguagens.

A construção de linguagens a partir de sentenças já definidas é uma área promissora para a engenharia reversa de domínios. Para isto, é necessário o desenvolvimento de um ambiente especializado em adquirir e transformar programas fonte das aplicações para as quais se deseja realizar a engenharia reversa. Em tal ambiente a base conceitual e lingüística será construída a partir da tradução destes fontes em representações internas. Estas representações internas são o objeto primário da investigação, sofrendo enriquecimento semântico e transformações generalizadoras que dão origem ao modelo conceitual do domínio de aplicação.

No âmbito do suporte a engenharia de domínios, sugerimos que o seguimento deste trabalho se dê pela avaliação empírica do paradigma proposto dentro de

domínios de aplicações específicos. Esta avaliação deve contemplar o esforço de aprendizado das técnicas apresentadas, os benefícios e os problemas eventualmente encontrados, bem como buscar indícios de medidas que indiquem a facilidade com que um domínio de aplicação permite representar soluções de problemas, com a finalidade de antecipar a percepção de anomalias.

Esperamos que os resultados obtidos nesta tese sirvam como subsídios para a importante investigação de tecnologias de apoio à reutilização, e que experiências futuras venham prover o "feedback" necessário para o aprimoramento da proposta.

7. Referências Bibliográficas

[Aure94] Aurélio Buarque de Holanda, *Dicionário Aurélio*, Editora Nova Fronteira, 1994.

[Aran91] Guillermo Arango; Rubén Prieto-Díaz - *"Domain Analysis Concepts and Research Directions"* em Domain Analiysis and Software Systems Modeling, (ed.) Rubén Prieto-Díaz & Guillermo Arango - IEEE Computer Society Press, 1991

[Aran94] Guillermo Arango - *"Domain analysis methods"* em Software Reusability, (ed.) Wilhelm Schäfer, Rubén Prieto-Díaz, Masao Matssumoto - Ellis Horwood, 1994.

[Bigg87] Ted J. Biggerstaff e Charles Richter - *"Reusability Framework, Assessment, and Directions"*, IEEE Software, Vol 4. #2, março de 1987

[Bigg89] Ted J. Biggerstaff e Anal J.Perlis, (ed.), *"Software Reusability"*, Volume I, Concepts and Models, ACM Press, 1989.

[Birt73] G.Birstwistle, O. Dahl, B. Myhrtag, e K. Nygaard - *"Simula Begin"*, Auerbach Press, Philadelphia, 1973

[Blai89] Gordon S. Blair, John J. Gallegher, and Javad Malik - *"Genericity vs Inheritance vs Delegation vs Conformance vs..."* , JOOP, set/out 1989.

[Booc83] Grady Booch - *"Software Engineering with Ada"*, The Benjamin/Cummings Publishing Company, Inc, 1983

[Booc91] Grady Booch - *"Object Oriented Design with Applications"*, The Benjamin/Cummings Publishing Company, Inc.,1991

[Broo87] Frederick P. Brooks - *"No Silver Bullet: Essence and Accidents of Software Engineering"*, IEEE Computer, vol 20(4).

[Brug62] Walter Brugger - *Dicionário de Filosofia* , Editora Herded, 1962

[Carré90] Bernard Carré, Jean-Marc Geib - *"The Point of View notion for Multiple Inheritance"*, Proceedings of ECOOP/OOPSLA'90, outubro de 1990

[Cima94] Alberto M. De Cima, Cláudia M. L. Werner, Alessandro A. C. Cerqueira - *"The Design of Object-oriented Software with Domain Architeture Reuse"*, Third International Conference on Software Reuse, Rio de Janeiro, 1994

[Coad91] Peter Coad, Edward Yourdon - *"Object-Oriented Design"*, Prentice-Hall Inc, 1991

[Coad92] Peter Coad, Edward Yourdon , *"Análise Baseada em Objetos"* - Editora Campus, 1992.

[Conk87] Conklin J. - *"Hypertext: An Introduction and Survey"*, IEEE Computer, setembro 1987.

[Cour85] P. Courtois - *"On Time and Space Decomposition of Complex Structures"*, Communications da ACM, junho de 1985.

[Deut83] L. Peter Deutsch - *"Reusability in The Smalltalk-80 Programming System"*, ITT Workshop on Reusability in Programming, 1983.

[Ecke91] Bruce Eckel - *C++ : Guia do Usuário*, Makron Books do Brasil Editora Ltda, 1991.

[Fish88] - Charles N. Fischer, Richard J. LeBlanc, Jr. - *"Crafting A Compiler"*, The Benjamin/Cummings Publishing Company, Ic., 1988

[Free87a] Peter Freeman - *"A Perspective on Reusability"* em Tutorial: Software Reusability, (ed.) Peter Freeman, Computer society Press of the IEEE, 1987

[Free87b] Peter Freeman - *"A Conceptual Analysis of the Draco Approach to Constructing Software Systems"* em Tutorial: Software Reusability, (ed.) Peter Freeman, Computer Society Press of the IEEE, 1987

[Germ92] Tom Germond - *"Object Mapping in C++"* em Microsoft Development Library, Microsoft Corporation, 1992

[Ghez91] Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli - *"Fundamentals of Software Engineering"*, Prentice-Hall international, Inc., 1991

[Gold83] Adele Goldberg - *"Smalltalk-80: the language and its implementation"*, Addison-Wesley Publishing Company, 1983.

[Holu90] Allen I. Holub - *"Compiler Design in C"*, Prentice Hall Inc. 1990

[Horo84] Ellis Horowitz e John B. Munson - *"An Expansive View of reusable Software"*, IEEE Transactions on Software Engineering, número 5, setembro de 1984

[IMSL79] IMSL - *"Library Reference Manual"*, IMSL Inc., 1979

[Jaco92] Yvar Jacobson - *"Object-Oriented Software Engineering: A Use Case Driven Approach"* ACM Press, 1992

[Japi91] Hilton Japiassú e Danilo Marcondes - *Dicionário Básico de Filosofia* - Jorge Zahar Editora, 1991.

[John88] Ralph E. Johnson e Brian Foote - *"Designing Reusable Classes"*, Journal of Object-Oriented Programming, SIGS Publications, Inc., junho/julho de 1988.

[Katz87] Shmuel Katz, Charles A. Richter, khe-Sing The - *"PARIS: A System for Reusing Partially Interpreted Schemas"*, Ninth International Conference on Software Engineering, ACM, Inc., 1987

[Khos94] Setrag Koshafian - *Bancos de Dados Orientados a Objetos*, Livraria e Editora Infobook, 1994

[King89] Roger King - *"My Cat Is Object-Oriented"*, em Object-Oriented Concepts, Databases and Applications, (ed) Won Kin e Frederick Lochovsky, ACM Press, NY, 1989.

[Kras83] Glenn Krasner - *"Smalltalk-80, Bits of History, Words of Advise"*, Addison-Wesley Publishing Company, 1983.

[Krue92] Charles W. Krueger - *"Software Reuse"*, ACM Computing Surveys, junho, 1992.

[Kunh90] Thomas Kuhn - *A Estrutura das Revoluções Científicas*, Editora Perspectiva, 1990.

[Leit92] Júlio C. P. Leite, Antonio Prado, Marcelo Sant'Anna - *DRACO-PUC, experiências e resultados de re-engenharia de software*, VI Simpósico Brasileiro de Engenharia de Software, novembro 1992

[Leit94] Júlio C.P.Leite, Marcelo Sant'Anna, F.G. de Freitas - *"Draco-PUC: A Techonology Assembly for Domain Oriented Software development"*, Third International Conference on Software Reuse, Rio de Janeiro, 1994

[Lewi91] John A. Lewis, Sallie M. Henry, Dennis G. Kafura, Robert S. Schulman - *"An Empirical Study of the Object-Oriented Paradigm and Software Reuse"*, OOPSLA 91, 1991

[Lieb86] Henry Lieberman, *"Using prototipical Objects to Implement Shared Behavior in Object Oriented Systems"*, OOPSLA, setembro 1986.

[Mads90] Ole L. Madsen e Boris Magnusson - *"Strong Typing of Object-Oriented Languages Revisted"*, Proceedings of ECOOP/OOPSLA90.

[Matt93] Flavio Araujo de Mattos - *Um Modelo de Organização de Bancos de Software Reusável*, II Workshop de Pesquisas de Tese em Engenharia de Software - COPPE-UFRJ & PUC-Rio, outubro de 1993

[McCa92] Francis G. McCabe - *"Logic and Objects"*, Prentice Hall, 1992

[Mcil68] M. D. Mcilroy - *"Mass-produced software components"*, Software Eng. Concepts and Techiniques, NATO Conf. Software Eng., J.M.Buxton, P. Naur, and B.Randell, (eds.) 1976.

[Melle89] Fred Mellender, Steve Riegel, Andrew Straw - *"Optimizing Smalltalk Message Performance"*, Object-Oriented Concepts, Databases and Applications, (ed.) Won Kin and Frederick Lochovsky, ACM Press, NY, 1989

[Meye87] Bertrand Meyer - *"Reusability: The Case For Object-Oriented Design"* , IEEE Software, março 1987.

[Meye88] Bertrand Meyer - *"Object-oriented Software Construction"*, Prentice Hall International, 1988

[Meye89] Bertrand Meyer - *"The New Culture of Software Development: Reflections on the practice of Object-Oriented Desgign"*, TOOLS'89

[Meye92] Bertrand Meyer - *"Eiffel, The Language"*, Prentice Hall International, 1992

[Micr93] Microsoft - *"Microsoft Visual C++ Class Library Reference"*, Microsoft Corporation, 1993.

[Micr94] Microsoft - *"Microsof Windows - Guia do Usuário"*, Microsoft Corporation

[Moss88] Moss, E. e wolf, A. - *"Toward Principles of Inheritance and Subtyping in Programming Languages"*, COINS Technical Report 88-95, University of Massachusetts.

[Neig84] James M. Neighbors, *"The Draco Approach to Constructing software from Reusable Components"* , IEEE Transactions on Software Engineering, setembro 1984

[Neig89] James M. Neighbors, *"Draco - A Method For Engineering Reusable Software systems"*, Software Reusability, Volume I, Concepts and Models, (ed.) Ted J. Biggerstaff e Alan J.Perlis, ACM Press, 1989.

[Neig94] James M. Neighbors - *"An Assessment of Reuse Technology after Ten Years"*, Third International Conference on Software Reuse, Rio de Janeiro,1994

[Nier89] Oscar Nierstrasz - *"A Survey of Object-Oriented Concepts"*, em Object-Oriented Concepts, Databases and Applications, (ed.) Won Kin e Frederick Lochovsky, ACM Press, NY, 1989.

[Page88] Meilir Page-Jones, *Projeto Estruturado de Sistemas* , Editora McGraw-Hill, 1988

[Parn83] D. L. Parnas, P. C. Clements, D.M. Weiss - *"Enhancing reusability with Information Hiding"* - ITT Workshop on reusability in Programming, 1983.

[Pepe93] Vera Lúcia Edais Pepe - *Breve histórico do percurso de Kuhn: do paradigma ao exemplar.* - UERJ/IMS - Rio de Janeiro, 1993

[Prie85] Rubén Prieto-Díaz - *"A Software Classification Scheme"*, Ph.D.thesis, Universidade da California, 1985

[Prie87] Rubén Prieto-Díaz - *"Domain Analysis For Reusability"* , COMPSAC'87, 1987.

[Prie89] Rubén Prieto-Díaz - *"Classification of Reusable Modules"* em Software Reusability, Volume I, Concepts and Models, (ed.) Ted J. Biggerstaff e Alan J.Perlis, ACM Press, 1989.

[Prie90] Rubén Prieto-Díaz - "Domain Analysis: An Introduction" , ACM SIGSOFT Software Engineering Notes, abril 1990.

[Prie91] Rubén Prieto-Diaz - *"Implementing Faceted Classification for Software Reuse"*, Communications of the ACM, maio 1991.

[Rine91] D. C. Rine - *"A Proposed Standard Set of Principle for Object-Oriented Development"*, ACM Software Engineering Notes, janeiro, 1991

[Roch87] Ana Regina Cavalcanti da Rocha - *Análise e Projeto Estruturado de Sistemas*, Editora Campus, 1987.

[Scio89] Edward Sciore - *"Object Specialization"*, ACM Transacions on Information Systems, abril 1989

[Stein89] Lynn Andrea Stein, Henry Lieberman, David Ungar - *"A Shared View of Sharing: The Treaty of Orlando"* em Object-Oriented Concepts, Databases and Applications, (ed.) Won Kin e Frederick Lochovsky, ACM Press, NY, 1989.

[Stro93] - Bjarne Stroustrup, Margaret A. Ellis - *C++, Manual de Referência Comentado*, Editora Campus, 1993

[Szwa83] Jayme Luiz Szwarcfiter - *Grafos e Algoritmos Computacionais*, Editora Campus, 1983

[Toml89] Chris Tomlinson, Mark Scheevel - *"Concurrent Object-Oriented Programming Languages"* em Object-Oriented Concepts, Databases and Applications, (ed.) Won Kin and Frederick Lochovsky, ACM Press, NY, 1989.

[Trac88a] Will Tracz - *"Software Reuse: Motivators and Inhibitors"* em Tutorial : Software Reuse: emerging Technology, (ed) Will Tracks , Computer Society Press of IEEE, 1988.

[Trac88b] Will Tracz - *"Software Reuse Myths"* - ACM SIGSOFT Software Engineering Notes", janeiro 1988.

[Trac90] Will Tracz - *"Where Does Reuse Start ?"*, ACM SIGSOFT Software Engineering Notes, abril 1990

[Trac92] Will Tracz - *"Domain Analysis Work Group Report"* - First International workshop on Software Reusability, ACM Sigsoft Software Engineering, Notes - julho 1992

[Unga87] David Ungar and Randall B. Smith, *"Self : The Power of Simplicity"* - OOPSLA'87

[Vigo87] L. S. Vigotsky - *Pensamento e Linguagem* - Livraria Martins Fontes Editora Ltda, 1987

[Wegn87] Peter Wegner - *"Dimensions of Object -Based Language Design"* , OOPSLA'87

[Wegn90] Peter Wegner - *"Concepts and Paradigms of Object-Oriented Programming"*, OOPS MESSENGER, Vol I #1, agosto 1990

[Wern91] Cláudia M. L. Werner; Jano M. Souza - *"An Object-Oriented Composition Environment for Scientific Applications"*, Computing in High Energy Fysics Conference, Tsukuba City, Japão, 1991

[Wern92a] Cláudia M. L. Werner - *Reutilização de Software no Desenvolvimento de Software Científico* - Tese de Doutourado, COPPE/UFRJ, Rio de Janeiro, março 1992

[Wern92b] Cláudia M. L. Werner; Flavio A. de Mattos - *Um ambiente para o desenvolvimento baseado na composição de aplicações*, VI Simpósio Brasileiro de Engenharia de Software, novembro 1992

[Wern93] Cláudia M. L. Werner, Flavio A. de Mattos - *CAOS - Sistema de Composição de Aplicações Orientado a Objetos*, Relatório Técnico - COPPE UFRJ, dezembro 1993

[Whit91] Whitewater/Symantec - *"Actor Programming"*; 1991

8. Anexo

Neste anexo apresentamos a sintaxe da linguagem adotada para especificar modelos e gramáticas no modelo de objetos estendido. Esta linguagem foi usada para descrever os exemplos da seção 5.3.2.2.

Classe -> **CLASS** Nome_Classe Superclasse Composição Inteface Implementacao **END_CLASS**

Nome_Classe ::= Identificador

Superclasse ::= SUBCLASS OF Nome_Classe

Composicao ::= TipoComposicao Nome_Classe Renomeando Composicao
Composicao ::=

Tipo-Composicao ::= UNION_WITH
Tipo-Composicao ::= INTERSECTION_WITH
Tipo-Composicao ::= DIFFERENCE_WITH

Renomeando ::= RENAMING Simbolo AS Simbolo Renomeando
Renomeando ::=

Interface ::= SYMBOLS_ARE ListaDeSimbolos INTERFACE_IS ListaDeProducoes

ListaDeSimbolos ::= Simbolo
ListaDeSimbolos ::= Simbolo , ListaDeSimbolos

ListaDeProducao ::= CabecalhoProducao ; ListaDeProducao
ListaDeProducao ::= ;

CabecalhoProducao ::= LadoEsquerdo -> LadoDireito

LadoEsquerdo -> SELF
LadoEsquerdo -> Simbolo

LadoDireito ->
LadoDireito -> Simbolo LadoDireito

Implementacao -> IMPLEMENTATION_IS ListaDeMétodos

ListaDeMétodos->Método ListaDeMétodos
ListaDeMétodos->

Método -> CabecalhoProducao Bloco
Bloco -> { ListaDeInstrucoes }

ListaDeInstrucoes -> Instrucao ;
ListaDeINstrucoes -> Instrucao ; ListaDeInstrucoes

Instrucao -> AlocacaoDeVariavel
Instrucao -> Atribuicao
Instrucao -> EnvioDeMensagem
Instrucao -> RetornoDeValor

AlocacaoDeVariavel -> LOCAL Identificador
Atribuicao -> NomeVariavel := EnvioDeMensagem;
RetornoDeValor -> RETURN EnvioDeMensagem;

EnvioDeMensagem -> NomeVariavel ListaDeValores
ListaDeValores -> Valor ListaDeValores
ListaDeValores ->

Valor -> Simbolo
Valor -> Simbolo [Numero]