UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Inspeção de Aplicações Java através da Identificação de Padrões de Projeto

por

ANDRÉ LUIS CASTRO DE FREITAS

Tese submetida à avaliação, como requisito parcial para
a obtenção do grau de Doutor em
Ciência da computação

Prof[a]. Dr[a]. Ana Maria de Alencar Price
Orientadora

Porto Alegre, julho de 2003.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Freitas, André Luis Castro de

Inspeção de Aplicações Java através da Identificação de Padrões de Projeto / por André Luis Castro de Freitas. - Porto Alegre : PPGC da UFRGS, 2003.

132 f. : il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós Graduação em Computação, Porto Alegre, BR - RS, 2003. Orientadora: Price, Ana M. A.

1. Engenharia de Software. 2. Software Orientado a Objetos. 3. Engenharia Reversa. 4. Reflexão Computacional. 5. Padrões de Projeto. 6. Linguagem Java. I. Price, Ana Maria de Alencar. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Prof[a]. Wrana Panizi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitora de Pós-Graduação: Prof[a]. Jocélia Grazia
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

São alguns anos de trabalho e, nesta jornada, são muitos aqueles que, de alguma forma, deram a sua contribuição e incentivos para que se chegasse ao final desta obra.

Em primeiro lugar, a Deus, pela companhia durante esta jornada e pela luz em meu caminho.

À minha querida esposa Luciane, pelo carinho, apoio e incentivo recebidos para a realização deste trabalho.

Às minhas filhas Luisa e Ana Laura, por me mostrarem que o ciclo da vida é simplesmente perfeito.

A meus pais Elói e Schirley, por minha vida.

Aos meus sogros e padrinhos Luis e Jahyr, pela amizade e companhia.

A minha orientadora Prof[a]. Ana Maria, pela paciência durante a orientação deste trabalho.

Aos professores, funcionários e colegas do pós-graduação, que direta ou indiretamente contribuíram para a realização deste trabalho.

Aos colegas de trabalho que incentivaram a finalização deste trabalho.

Por fim, à FAPERGS – Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, pelo apoio e incentivo financeiros para a realização deste trabalho.

O bom senso é a coisa mais igualmente compartilhada no mundo, pois cada um de nós pensa que é tão bem dotado dele que mesmo os mais difíceis de agradar em todos os outros aspectos não costumam querer mais bom senso do que possuem. É improvável que todos estejam enganados nesse sentido. Ao contrário, isso indica que a capacidade de julgar corretamente e de distinguir o verdadeiro do falso, que é propriamente o que se chama razão ou senso comum, é, naturalmente, igual em todos os homens e, conseqüentemente, a diversidade de nossas opiniões não provêm de alguns serem mais capazes de raciocinar do que outros, mas apenas de conduzirmos nossos pensamentos em linhas diferentes sem examinar as mesmas coisas.

René Descartes
Discourse on the Method

Sumário

Lista de Abreviaturas 07
Lista de Figuras 08
Resumo 10
Abstract 11

1 Introdução 12
1.1 Motivação 13
1.2 Objetivos 14
1.2 Estrutura Geral do Trabalho 15

2 Software Orientado a Objetos 16
2.1 Dimensões de um Software Orientado a Objetos 16
2.2 Reutilização de Software 17
2.2.1 *Frameworks* 18
2.2.2 Padrões de Projeto 19
2.3 Engenharia Reversa e Re-Engenharia 20
2.3.2 Reflexão Computacional 21
2.4 Conclusões 22

3 Referencial Teórico 24
3.1 Automatização no Desenvolvimento de Sistemas de Software OO 24
3.1.1 Agarwal 24
3.1.2 Jiason 26
3.1.3 Freitas 27
3.2 Geração e/ou Formalização de Padrões de Projeto 29
3.2.1 Budinsky 29
3.2.2 Wild 30
3.2.3 Eden 31
3.2.4 Lauder 33
3.2.5 Vlissides 35
3.2.6 Freitas 36
3.3 Verificação e Identificação Automáticas de Padrões de Projeto 37
3.3.1 Krämer 37
3.3.2 Bansiya 38
3.3.3 Seeman 41
3.3.4 Freitas 43
3.3.5 Guéhéneuc 45
3.4 Conclusões 47

4 Identificação de Padrões de Projeto 50
4.1 Detecção de Padrões de Projeto 50
4.1.1 Tornando um Padrão Detectável 51
4.2 Escolha de Padrões para Identificação 52
4.2.1 Padrão *Composite* 52
4.2.2 Padrão *Decorator* 57
4.2.3 Padrão *Strategy* 61

4.2.4 Padrão *Template Method* ... 63
4.2.5 Padrão *Observer* ... 65
4.2.6 Padrão *Chain of Responsibility* ... 68
4.2.7 Padrão *State* ... 71
4.3 Conclusões ... 73

5 Extrações Estática e Dinâmica das Informações ... 74
5.1 Extração de Informações ... 74
5.2 O Modelo Proposto ... 75
5.3 Extração Estática de Informações ... 77
5.3.1 *ANother Tool for Language Recognition* ... 78
5.3.2 Processos do Gerador de Referência Cruzada ... 78
5.4 Extração Dinâmica de Informações ... 79
5.4.1 Linhas de Execução ... 79
5.4.2 Comunicação entre Linhas de Execução ... 79
5.4.3 Clonagem de Objetos ... 80
5.4.4 Modelo da Extração Dinâmica ... 83
5.5 Reflexão Computacional ... 84
5.5.1 Reflexão Computacional ... 85
5.6 Identificação de Padrões de Projeto ... 87
5.7 Conclusões ... 88

6 Demonstração da Ferramenta ... 89
6.1 Geração do Arquivo Fonte Modificado ... 89
6.2 Interfaces e Execução ... 90
6.2.1 Interface de Apresentação ... 91
6.2.2 O Exemplo Proposto ... 91
6.2.3 Extração Estática de Informações ... 93
6.2.4 *Browser* de Visualização ... 94
6.2.5 Extração Dinâmica de Informações ... 96
6.2.6 Inspeção de Objetos ... 96
6.2.7 Verificação de Objetos ... 98
6.2.8 Identificação de Colaborações e Padrões ... 103
6.3 Conclusões ... 106

7 Conclusões ... 107
7.1 Contribuições ... 107
7.2 Trabalhos Futuros ... 108

Anexo 1 Descrição da Análise Léxica e Sintática de ANTLR ... 110

Anexo 2 Métodos para Identificação de Padrões ... 121

Referências ... 129

Lista de Abreviaturas

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
BNF	Backus-Naur Form
CAD	Computer Aided Design
CASE	Computer Aided Software Engineering
HTML	Hypertext Markup Language
OMT	Object Modeling Techinique
OO	Orientado a Objetos
UML	Unified Modeling Language

Lista de Figuras

FIGURA 2.1 – Visão conceitual de um *framework* 19
FIGURA 2.2 – Projeto de Sistemas de Software OO 20

FIGURA 3.1 – Desenvolvimento de Software OO com PATHOS 25
FIGURA 3.2 – Modelos de Especificação 33
FIGURA 3.3 – Diagrama *Composite* 34
FIGURA 3.4 – Diagrama *Composite – Add()* 34
FIGURA 3.5 – Arquitetura da Ferramenta *Pat System* 37
FIGURA 3.6 – Estrutura Padrão *Composite* 40
FIGURA 3.7 – Ferramenta DP++ 40
FIGURA 3.8 – Ferramenta de Detecção Automática 44

FIGURA 4.1 – Estrutura Abstrata do *Composite* para a Linguagem C++ 53
FIGURA 4.2 – Estrutura Abstrata do *Composite* para a Linguagem Java 54
FIGURA 4.3 – Diagrama de Objetos - Padrão *Composite* 54
FIGURA 4.4 – Ciclos em *Composite* 56
FIGURA 4.5 – Estrutura Abstrata do *Decorator* para a Linguagem C++ 57
FIGURA 4.6 – Estrutura Abstrata do *Decorator* para a Linguagem Java 58
FIGURA 4.7 – Diagrama de Objetos - Padrão *Decorator*.................................. 58
FIGURA 4.8 – Ciclos em *Decorator* 60
FIGURA 4.9 – Estrutura Abstrata do *Strategy* para a Linguagem C++ 61
FIGURA 4.10 – Estrutura Abstrata do Strategy para a Linguagem Java 62
FIGURA 4.11 – Estrutura Elementar do *Strategy* 62
FIGURA 4.12 – Estrutura Abstrata do *Template Method* para a Linguagem C++. 63
FIGURA 4.13 – Estrutura Abstrata do *Template Method* para a Linguagem Java. 64
FIGURA 4.14 – Estrutura Abstrata do *Observer* para a Linguagem C++ 66
FIGURA 4.15 – Estrutura Abstrata do *Observer* para a Linguagem Java 67
FIGURA 4.16 – Estrutura do *Chain of Responsability* para a Linguagem C++..... 69
FIGURA 4.17 – Exemplo de Execução para *Chain of Responsability* 69
FIGURA 4.18 – Estrutura Abstrata do *State* para a Linguagem C++ 71
FIGURA 4.19 – Estrutura Abstrata do *State* para a Linguagem Java 72

FIGURA 5.1 – Processos Executados na Ferramenta de Inspeção 76
FIGURA 5.2 – Diagrama de Classes – Geração de Referência Cruzada 77
FIGURA 5.3 – *Streams* em *Pipe* 80
FIGURA 5.4 – Serialização e Clonagem 81
FIGURA 5.5 – Diagrama de Mensagens – *clone()* 81
FIGURA 5.6 – Fluxo de Saída 81
FIGURA 5.7 – Diagrama de Mensagens – *writeObjeto()* 82
FIGURA 5.8 – Fluxo de Entrada 82
FIGURA 5.9 – Diagrama de Mensagens – *readObject()*...................................... 83
FIGURA 5.10 – Diagrama de Classes – Extração de Informações 84
FIGURA 5.11 – Diagrama de Classes – Reflexão 85
FIGURA 5.12 – Avanço em um Iterador 87

FIGURA 6.1 – Interface de Seleção de Aplicações 91
FIGURA 6.2 – Diagrama de Classes da Aplicação 92
FIGURA 6.3 – Diagrama de Mensagens 92

FIGURA 6.4 – Diagrama de Objetos 93
FIGURA 6.5 – Interface de Visualização Referência Cruzada 95
FIGURA 6.6 – Interface de Representação dos Objetos Identificados 96
FIGURA 6.7 – Interface de Representação de Classe/Superclasses do Objeto 97
FIGURA 6.8 – Diagrama de Mensagens – *findUserObject()* 97
FIGURA 6.9 – Diagrama de Mensagens – *valueChanged()*................................ 98
FIGURA 6.10 – Diagrama de Objetos – Estado 1 99
FIGURA 6.11 – Diagrama de Objetos – Estado 3 99
FIGURA 6.12 – Diagrama de Mensagens – Inspeção de Atributos 100
FIGURA 6.13 – Diagrama de Colaborações – *criaArestasVertices()* 100
FIGURA 6.14 – Interface de Inspeção de Objetos 102
FIGURA 6.15 – Diagrama de Mensagens – *getChild()*.. 103
FIGURA 6.16 – Diagrama de Classes da Aplicação 103
FIGURA 6.17 – Diagrama de Classes c/Identificação de Padrões 105
FIGURA 6.18 – Interface de Ajuda de Reconhecimento 106

Resumo

Para reutilização, manutenção e refatoração, projetistas de sistemas de software, freqüentemente, precisam examinar o código fonte da aplicação para entender os detalhes dos sistemas desenvolvidos. As aplicações orientadas a objetos em geral, tornam-se coleções nebulosas de classes e implementações de métodos. Sem dúvida a habilidade de entender sistemas de software é largamente aumentada visualizando-se esses produtos em níveis mais altos de abstração.

Os padrões de projeto demonstram um alto índice de abstração e são considerados uma ferramenta efetiva para o entendimento de sistemas de software orientados a objetos. Aplicações orientadas a objetos visualizadas como um sistema de interação de padrões requerem a descoberta, identificação e classificação de grupos de classes relacionadas. Estas visualizações podem representar qualquer padrão conhecido ou agrupamentos que executam uma tarefa abstrata e necessariamente não são uma solução de padrão conhecida.

Os padrões de projeto descrevem, portanto, microarquiteturas que resolvem problemas arquitetônicos em sistemas de software orientados a objetos. É importante identificar estas microarquiteturas durante a fase de manutenção de aplicações orientadas a objetos. Faz-se necessário salientar que estas microarquiteturas aparecem freqüentemente distorcidas na aplicação fonte.

O objeto deste trabalho é demonstrar a viabilidade de construir uma ferramenta para descobrir a utilização de padrões de projeto em aplicações Java. Assim, esta tese examina as características de alguns padrões, determinando a natureza do que faz um padrão ser detectável por intermédio de meios automatizados, e propõe algumas regras pelas quais um conjunto de padrões possa ser identificado. As regras são baseadas nos relacionamentos entre classes e objetos mediante observação dos modelos estático e dinâmico.

Este trabalho também documenta o desenvolvimento do protótipo da ferramenta de inspeção, que tem por objetivo aplicar os processos de engenharia reversa e reflexão computacional sobre código Java, utilizando as informações adquiridas para detectar padrões de projeto. Finalmente, esta tese demonstra a utilização dessa ferramenta em um exemplo pequeno de aplicação Java e forma a base para trabalhos adicionais que investiguem a existência de diferentes padrões de projeto em sistemas de software construídos em Java.

Palavras-chaves: Engenharia de Software, Software Orientado a Objetos, Engenharia Reversa, Reflexão Computacional, Padrões de Projeto, Linguagem Java.

TITLE: "INSPECTION OF JAVA APPLICATIONS THROUGH DESIGN PATTERNS IDENTIFICATION"

Abstract

For reuse, maintenance and refactoration, software developers frequently need to examine source code to understand software systems. In general object-oriented applications become nebulous collections of class and method implementations. There is no doubt that the ability to understand software systems is greatly enhanced by visualizing software systems at higher levels of abstraction.

Design patterns demonstrate a high abstraction level and they are considered an effective tool for understanding object-oriented software systems. Visualizing object-oriented applications as a system of interacting patterns requires detecting, identifying, and classifying groups of related class. These visualizations represent either known design patterns or clusters that perform an abstract task and they are not necessarily a known pattern solution.

Therefore, design patterns describe micro-architectures that solve recurrent architectural problems in object-oriented software systems. It is important to identify these micro-architectures during the maintenance of object-oriented applications. But these micro-architectures often appear distorted in source applications.

The purpose of this research has been to demonstrate the feasibility of building a tool to detect the use of software design patterns in Java applications. So this work examines the characteristics of design patterns, determines the nature of what makes a design pattern detectable by automated means, and outlines rules by which a small set of design patterns can be detected. The rules are based on the structural and dynamical relationship between classes and objects.

This work also describes a prototype inspection tool, which can apply reverse engineering added by a computational reflection process on Java code, and use the retrieved information to detect design patterns. Finally, this thesis demonstrates the use of this tool on a small example and forms a basis for further work investigating the different design patterns on existing Java software.

Keywords: Software Engineering, Object-Oriented Software, Reverse Engineering, Computational Reflection, Design Patterns, Java Language.

1 Introdução

Durante as etapas de desenvolvimento de um sistema de software orientado a objetos, o projetista adota uma seqüência de atividades as quais produzem inúmeros documentos que finalizam com a produção da aplicação especificada (BOOCH, 1994). Em geral, estas atividades são definidas por meio do uso de metodologias apropriadas ao paradigma de desenvolvimento de software OO, escolhidas pelo projetista para a realização do trabalho em questão.

Essas metodologias de desenvolvimento de software e as linguagens de programação orientadas a objetos vêm evoluindo durante as últimas décadas no intuito de aprimorar cada vez mais o produto final esperado e, portanto, minimizar os problemas de manutenção do sistema de software construído. Entretanto, essas mesmas metodologias falham no fornecimento de suporte adequado à compreensão dos sistemas envolvidos, pois muitas vezes não apresentam formalismos adequados para refletir o mapeamento entre o projeto e a implementação (ARANGO, 1993).

Quando uma pessoa é inexperiente em relação à tecnologia OO, freqüentemente tenta entender como os conceitos trabalham dentro do espaço da solução do próprio problema que está sendo analisado. As várias metodologias de desenvolvimento de software OO mostram ao leitor como aplicar um processo de desenvolvimento particular utilizando um conjunto de diagramas para representar tais processos (COAD, 1997), o que restringe o leitor na aplicação dos conceitos.

É reconhecido que, para tornar-se um projetista especializado em técnicas de desenvolvimento de software OO, são necessários dedicação e treinamento em comparação ao aprendizado de técnicas estruturadas tradicionais. Isto acontece porque a orientação a objetos utiliza-se das técnicas estruturadas, e além de o projetista preocupar-se com as características de projeto que devem ser compreendidas, deve, também, observar a adição de novas estruturas às linguagens de programação e estudar as bibliotecas de recursos disponíveis.

Portanto, a dificuldade de construção de um bom sistema orientado a objetos pode ser atribuída a uma combinação de fatores que abrange aspectos tais como: complexidade do domínio da aplicação, limitações cognitivas dos projetistas, práticas de desenvolvimento pouco apropriadas e características da linguagem de programação. (CAMPO, 1997).

Sob esta ótica, acreditando que o software gerado não condiz muitas vezes com as especificações desejadas, parte-se do princípio de que a sua documentação não é de confiança absoluta. A documentação sofre as mesmas interferências, pois também faz parte do processo de desenvolvimento do sistema de software.

Acredita-se que ainda não se possa garantir com precisão a qualidade de um sistema de software orientado a objetos, devido aos fatores limitantes citados anteriormente. A solução computacional é, portanto, passível de manutenção. A manutenção de um produto de software nos anos 70 era responsável por aproximadamente 40% de todo o esforço despendido por uma organização de produção de sistemas. Nos anos 80, esse esforço passou para 60% e, atualmente, chega a 80%. É importante salientar que a manutenção de um produto de software não se resume somente à correção, mas também aplica-se às modificações e ampliações necessárias ao perfeito funcionamento do sistema (PRESSMAN, 2001). Portanto, na adaptação ou aperfeiçoamento de um sistema de software, devem-se determinar novos requisitos, reprojetar, gerar código e testar o sistema existente. Estas tarefas são definidas como manutenção.

Outros pontos a serem considerados é que, em muitos casos, o projetista que fará a manutenção do produto de software nem sempre é o mesmo que o criou, ou nenhuma metodologia de projeto foi aplicada, ou ainda, quando a documentação é incompleta e o registro das mudanças passadas é superficial. Certamente, este projetista despenderá um esforço de entendimento do software existente para posterior processo de modificações e correções. Assim, deve-se avaliar a manutenibilidade, a qual caracteriza o grau de facilidade com que um software pode ser entendido, corrigido, adaptado e/ou aumentado.

A manutenção deve concentrar-se em toda a configuração do software e não somente nas modificações no código fonte. Os efeitos colaterais da documentação acontecem quando as mudanças no código fonte não são refletidas na documentação do projeto ou nos manuais destinados ao usuário. Pior do que não possuir documentação é possuir uma documentação que não reflita o estado atual do software (PETERS, 2001).

Este trabalho tem como foco a necessidade de facilitar o processo de manutenção da solução computacional. Enfatiza-se aqui o quanto faz-se necessário a utilização da informação além da demonstração de seus elementos. Pensa-se que, ao explicar como são organizadas as classes e o que elas representam, verificando a instanciação dos objetos e suas trocas de estado durante a evolução da aplicação, possa-se reduzir significativamente o esforço despendido nesta fase.

Sob este enfoque faz-se uso dos processos de engenharia reversa e reflexão computacional os quais caracterizam o processo de analisar uma aplicação, com o objetivo de criar uma representação desta em um nível de abstração maior do que o código fonte.

Neste sentido, acredita-se que o estudo de técnicas de investigação, para extração de informações, de uma aplicação orientada a objetos, em tempo de execução, representa uma alternativa para se alcançarem melhores índices de qualidade nas aplicações a serem modificadas.

1.1 Motivação

As metodologias de desenvolvimento de software OO são suficientes para representar: relações estáticas entre objetos, herança, associação e relações de agregação de um determinado projeto. Infelizmente, as ferramentas existentes possuem limitações quanto à representação do aspecto dinâmico que caracteriza a troca de mensagens entre objetos durante a execução das tarefas da aplicação. Em geral, a maioria dos diagramas pode capturar somente uma porção de tempo da aplicação e não demonstra a evolução da estrutura de um objeto. Sabe-se, também, o quanto é difícil representar a natureza dinâmica de uma estrutura em tempo de execução.

A manutenção de um sistema de software OO é uma tarefa que requer um investimento substancial de tempo e de esforço, quando a documentação sobre a estrutura dinâmica da aplicação também é comprometida pela falta de recursos (SOMMERVILLE, 1996). A documentação é também afetada pela diversidade de entradas que podem ser associadas à aplicação, o que inviabiliza uma documentação total e caracteriza uma documentação elaborada para determinados casos.

Neste sentido, pesquisadores dedicam-se à definição de novas ferramentas de projeto/manutenção que possam garantir a qualidade do software produzido/modificado. Nestas ferramentas, faz-se necessária a utilização da informação com vistas a caracterizar e explicar a organização das classes e/ou objetos e suas respectivas representações, no intuito de verificar as trocas de estados dos objetos durante a evolução da aplicação.

Com a utilização de tais ferramentas, acredita-se que projetistas novatos possam compreender a existência das entidades envolvidas em um software, provendo um quadro de referência pelo qual essas entidades possam ser comparadas a outros exemplos.

Um outro aspecto a ser considerado é que as atividades inerentes à manutenção de software OO enquadram-se como atividades que se valem da experiência pessoal e da inteligência humana. Portanto, a utilização de modelos pode prover auxílio para a solução de problemas de qualquer software OO, independentemente do domínio da aplicação.

Os padrões de projeto são estruturas de colaboração de classes identificadas freqüentemente em projetos de software OO. Utilizando textos e diagramas, um padrão descreve, como um problema específico, que acontece repetidamente num projeto maior, pode ser resolvido (GAMMA, 1994). A proposta definida no trabalho de Gamma apresenta um catálogo de padrões. O objetivo desse catálogo é relacionar os problemas de projeto mais comumente encontrados e o modo como estes problemas podem ser resolvidos.

Possivelmente, a melhor idéia para projetar e entender software OO seja a criação de padrões de projeto. Os padrões em geral, mostram como representar a comunicação da informação sobre abstrações e soluções, apresentando um potencial para representar as abstrações comuns que projetistas experientes de software utilizam para resolver problemas básicos de projeto. De certo modo, este mecanismo deveria ser entendido e utilizado por um projetista novato.

Segundo Appleton (2001), a escrita de padrões bons é muito difícil. Padrões não só deveriam prover fatos como uma referência manual ou um guia de usuário, mas também contar uma história que capture a experiência que eles estão tentando demonstrar. Um padrão teria como função ajudar seus usuários a: compreender sistemas existentes; personalizar sistemas para ajustar necessidades de usuário e construir sistemas novos.

Considerado sob esse enfoque, no qual a principal função dos padrões de projeto é compreender os sistemas existentes, acredita-se que a identificação de padrões na aplicação possa contribuir positivamente para os processos de documentação e manutenção. O mais difícil para um projetista é identificar um padrão quando encontrar características da existência desse. O projetista dispõe inicialmente de algumas informações que julga serem indícios do reconhecimento, mas estas podem mostrar que nem sempre fazem sentido. O melhor modo para aprender como reconhecer e documentar padrões úteis é verificar como outros projetistas construíram tais padrões (ALPERT, 1998).

Diante de tais informações, acredita-se que o uso automatizado de ferramentas sobre o trabalho de projetistas durante o processo de manutenção de sistemas de software OO, é de grande importância para ajudar a reduzir a complexidade inerente do processo de compreensão. Identifica-se a necessidade de ferramentas que auxiliem o projetista na construção de modelos, mediante mecanismos de análise, exploração e visualização da informação em diferentes níveis de abstração.

1.2 Objetivos

Este trabalho tem os seguintes objetivos:

- Apresentar, de forma resumida e crítica, trabalhos desenvolvidos na área de padrões de projeto, considerando a sua importância nos processos de desenvolvimento e manutenção de sistemas de software orientados a objetos;

- Propor, com base em trabalhos científicos aqui apresentados, a identificação de padrões de projeto em aplicações desenvolvidas utilizando-se a linguagem Java. A identificação dos padrões utiliza como premissas os processos de engenharia reversa e reflexão computacional;
- Caracterizar um protótipo de ferramenta de inspeção para validar as idéias apresentadas. A ferramenta visa a identificar, em tempo de execução, os aspectos estático e dinâmico da aplicação no intuito de possibilitar ao projetista visualizar as informações sobre o sistema de software avaliado.

1.3 Estrutura Geral do Trabalho

O capítulo 2 caracteriza o desenvolvimento de software OO. Nesse capítulo, faz-se uma breve introdução e recapitulação dos conceitos fundamentais utilizados no paradigma OO, bem como dos conceitos pertinentes ao desenvolvimento deste trabalho.

O capítulo 3 apresenta um referencial teórico abordando os trabalhos que embasaram o desenvolvimento desta tese. Nesse capítulo, faz-se uma descrição dos trabalhos encontrados, bem como uma avaliação de suas contribuições.

O capítulo 4 avalia alguns padrões de projeto utilizados como exemplos para investigação e identificação na ferramenta proposta. Nesse capítulo, citam-se as características básicas de cada padrão para a sua identificação nas aplicações. Para manter a originalidade das figuras apresentadas nesse capítulo, optou-se por manter as notações utilizadas pelos autores citados. Portanto, serão apresentados exemplos utilizando-se as notações OMT e UML.

O capítulo 5 demonstra os mecanismos de extração estática e dinâmica de informações em aplicações orientadas a objetos. Nesse capítulo, são apresentadas algumas particularidades da extração de informações para aplicações desenvolvidas na linguagem Java.

O capítulo 6 caracteriza os processos que compõem a ferramenta proposta. São identificados todos os passos pelos quais passa uma aplicação até que se disponha das informações para avaliação em alto nível de abstração.

Por fim, as conclusões caracterizam, resumidamente, o trabalho realizado, encaminhando a trabalhos futuros com vistas à escrita e apresentação do relatório de conclusão de projeto de pesquisa vinculado à FAPERGS – Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul.

2 Desenvolvimento de Software Orientado a Objetos

A percepção de aspectos relevantes do mundo real, para fins de representação em computador, envolve o processo de identificação de abstrações. Se estas abstrações não apresentarem uma expressão direta no mundo computacional, a complexidade da solução será, evidentemente, aumentada pela distância entre os dois espaços.

No paradigma orientado a objetos, o mundo real consiste de objetos autônomos e concorrentes que interagem entre si, e cada objeto no projeto do software tem seu próprio estado e comportamento, semelhante ao seu correspondente no mundo real. Partindo do princípio de que o mundo real é um mundo de objetos que interagem entre si, o desenvolvimento de sistemas segundo o paradigma OO constitui uma abordagem que diminui a distância entre os problemas do mundo real e as soluções do mundo computacional (BOOCH, 1994).

Portanto, o projetista cria algoritmos que, quando executados no computador, produzirão resultados que permitem o mapeamento físico para alguma ação do mundo real ou, ainda, que podem ser examinados e interpretados por outras pessoas. Assim torna-se evidente pensar que, quanto mais próximo conceitualmente estiver o espaço das soluções do espaço do problema, mais fácil serão o desenvolvimento, a compreensão, a confiabilidade e a manutenção da aplicação.

Mas esta realidade não é segura. Conforme citado no capítulo 1, é necessário considerar que tanto os processos de desenvolvimento, como os de entendimento e de manutenção de um software OO não representam tarefas que possam ser realizadas facilmente por projetistas inexperientes na tecnologia.

Durante a fase de manutenção, por exemplo, são realizadas modificações em elementos individuais como estruturas, classes, métodos etc. Quando esses elementos são modificados, o projeto do software original já não apresenta mais uma especificação coerente e erros podem ocorrer. A manutenção deve, portanto, concentrar-se em toda a configuração do software, e não somente nas modificações propostas no código fonte. Assim, os efeitos colaterais na documentação acontecem quando as mudanças no código fonte não são refletidas na documentação do projeto ou nos manuais destinados ao usuário (PETERS, 2001).

Acredita-se, então, que é importante reduzir o intervalo entre documentação/projeto atualizados em relação ao código fonte. Por meio da utilização de uma ferramenta de inspeção de aplicações Java, tem-se como objetivo investigar o processo de execução de aplicações, levando-se em consideração diferentes entradas, proporcionando ao projetista um maior entendimento da aplicação durante o processo de manutenção de software OO.

2.1 Dimensões de um Software Orientado a Objetos

A modelagem orientada a objetos possibilita três dimensões para descrever um sistema relativamente complexo: a dimensão estrutural dos objetos, a dimensão funcional dos requisitos e, ainda, a dimensão dinâmica do comportamento (RUMBAUGH, 1994).

A dimensão estrutural centra-se no aspecto estático ou passivo. A dimensão estrutural relaciona-se com a estrutura estática dos objetos que fazem parte do sistema. A estrutura reflete a identidade de cada objeto, sua classificação, seu encapsulamento e seus relacionamentos estáticos (hierarquias de generalização e especialização, agregação e associações específicas). Esta dimensão é fundamental na modelagem orientada a objetos porque constitui a base sobre a qual os outros aspectos trabalham. É

claro que, se esta dimensão não estiver bem documentada pode-se incorrer em sérios erros de entendimento sobre a aplicação. Mas, em geral, esta dimensão, durante o processo de manutenção, pode ser visualizada por intermédio da observação global do código fonte. Mesmo um projetista novato na tecnologia pode visualizar estes elementos com certa facilidade.

A dimensão funcional dos requisitos considera o aspecto relativo à função de transformação global do sistema, à conversão de entradas em resultados. Em geral, esta dimensão é pouco afetada em sua documentação levando-se em consideração alterações no código fonte. Acredita-se que apenas mudanças radicais na estrutura básica da aplicação forcem modificações na documentação desta dimensão.

Por fim, na dimensão dinâmica, vê-se o aspecto dinâmico ou ativo que descreve o comportamento dos objetos que constituem o sistema. A documentação desta dimensão, na grande maioria das vezes, não retrata claramente o comportamento da aplicação em tempo de execução, devido à grande variedade de entradas, principalmente em sistema maiores. A partir desta constatação, acredita-se que visualizar os estados dos objetos com a possibilidade de acompanhar as modificações no seu comportamento represente uma diminuição no esforço do entendimento do software avaliado.

2.2 Reutilização de Software

A reutilização de software é vista como um fator que pode levar ao aumento da produtividade da atividade de desenvolvimento de software, na medida em que o uso de elementos de software já desenvolvidos e depurados reduz o tempo de desenvolvimento, de testes e as possibilidades de introdução de erros ao longo do desenvolvimento.

A engenharia de software baseada na reutilização é uma abordagem para o desenvolvimento que tenta maximizar o reuso de software já existente. Segundo Sommerville (1996), as unidades de software que são reutilizadas podem apresentar características totalmente diferentes:

- reuso de sistemas em aplicações: todo sistema de aplicações pode ser reutilizado pela sua incorporação, sem modificações, em outros sistemas;
- reuso de componentes: componentes de uma aplicação que variam em tamanho, incluindo desde subsistemas até objetos isolados, podem ser reutilizados;
- reuso de funções: componentes de software que implementam uma única função podem ser reutilizados.

Uma aplicação orientada a objetos constrói-se com classes que colaboram entre si, por meio da troca de mensagens para realizar as tarefas do sistema. A distribuição de responsabilidades entre essas classes e os padrões de colaboração entre elas constituem o projeto da aplicação.

A reutilização de elementos de software pode ocorrer no código ou no projeto. (DEUSTCH, 1989). A reutilização de código consiste na utilização direta de trechos de código já desenvolvidos. A reutilização de projeto consiste no reaproveitamento de concepções arquitetônicas de uma aplicação em outra aplicação, não necessariamente com a utilização da mesma implementação.

Para que ocorra um processo eficiente de reutilização, é importante que a documentação dos elementos de software esteja atualizada. Considerando-se, portanto, uma documentação inadequada, torna-se muito difícil aproveitar partes do sistema que não demonstrem um entendimento claro quanto à execução.

2.2.1 *Frameworks*

Mediante os mecanismos fornecidos pelo paradigma OO, é possível reutilizar uma aplicação tanto como uma caixa-preta, por meio de uma interface que permita acessar os serviços que implementa, quanto como uma caixa-branca, redefinindo o comportamento de algumas subclasses. Pode-se, assim, obter diferentes aplicações utilizando como modelo uma aplicação existente, reutilizando tanto o código como o projeto geral dessa aplicação base. A quantidade de comportamento redefinido dependerá do grau de semelhança entre as aplicações. Se as aplicações forem altamente semelhantes, provavelmente só alguns métodos devam ser redefinidos e poucas classes ser especializadas, obtendo-se, em conseqüência, uma grande reutilização. Ao contrário, se as aplicações diferem muito, provavelmente muitas classes precisam ser redefinidas, diminuindo o ganho na reutilização (CAMPO, 1997).

Analisando, posteriormente, o comportamento redefinido, é possível verificar que existe uma parte de comportamento que é comum e que pode ser generalizada. Portanto, o projetista pode dividir o projeto em duas partes: o código original que implementa a porção específica do comportamento abstrato, e a criação de novas classes, superclasses das anteriores, que contenham todo o código comum a ambas. Isto significa que a generalização da fatoração de vários casos concretos torna possível obter-se uma classe abstrata que defina o comportamento genérico deste conjunto de casos, através de métodos abstratos, *template*, base e *hook*.

Os métodos abstratos definem apenas a interface para as operações que serão específicas a cada aplicação. Qualquer aplicação instanciada a partir do *framework* deverá fornecer a implementação adequada para cada um dos métodos abstratos. Os métodos *template* implementam algoritmos genéricos que definirão o fluxo de controle do *framework*. Os métodos base são aqueles que possuem um comportamento padrão. E os métodos *hook* apresentam um comportamento padrão para alguns aspectos do *framework*. Os últimos podem ainda ser redefinidos por subclasses concretas de uma aplicação, a fim de fornecer um novo comportamento para eles (PREE, 1995).

Um conjunto de entidades relacionadas sugere a criação de classes abstratas que representem formalizações genéricas relativas a este conjunto. Cada entidade representa um caso em particular da abstração e será representada por uma subclasse concreta. Esta subclasse fornecerá uma variante específica do comportamento abstrato definido na classe abstrata. As classes abstratas representam modelos para as suas subclasses e, portanto, um projeto construído por classes abstratas funciona como um modelo para aplicações. Um projeto construído por classes abstratas é denominado um *framework* de aplicação orientado a objetos (JOHNSON, 1992).

Um *framework* é composto de dois níveis, conforme a Figura 2.1. Um nível mais abrangente, que fornece a estrutura de controle da aplicação, constituído pelas classes do *framework*, e um nível inferior, constituído por subclasses concretas implementadas pelo projetista. O nível inferior permite a implementação de operações específicas cuja descrição é modelada no nível genérico. As operações podem especializar a estrutura de controle de acordo com os requisitos da aplicação específica, bem como chamar operações definidas no nível genérico.

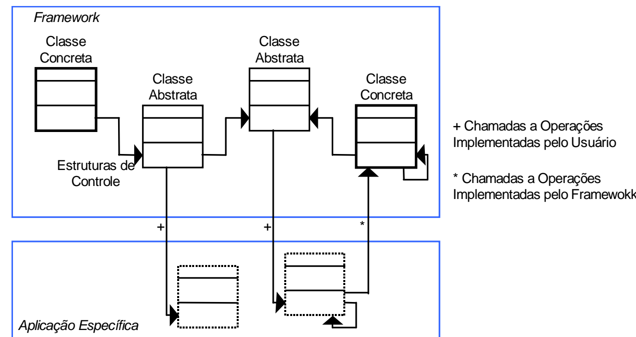

FIGURA 2.1 - Visão conceitual de um *framework*

A manutenção de uma aplicação que utiliza um *framework* não é uma atividade trivial, pois o comportamento da aplicação subdivide-se em uma parte representada pela porção do *framework* e e m uma outra parte constituída por subclasses concretas implementadas pelo projetista. Quando pequenas modificações são necessárias, provavelmente só alguns métodos devem ser redefinidos e poucas subclasses devem ser especializadas. Em contrapartida, se grandes modificações forem propostas, os efeitos colaterais na documentação podem tornar a acontecer. Este precedente acarretará o difícil entendimento da aplicação.

2.2.2 Padrões de Projeto

Padrões de Projeto são estruturas de colaboração de classes freqüentemente identificadas no projeto de *frameworks* orientados a objetos. Utilizando textos e diagramas, um padrão descreve como um problema específico, que acontece repetidamente num projeto maior, pode ser resolvido (GAMMA, 1994).

O trabalho de Gamma (1994) apresenta um catálogo de padrões. O objetivo deste catálogo é relacionar os problemas de projeto mais comumente encontrados na construção de *frameworks* e o modo como estes problemas podem ser resolvidos.

Um critério de classificação de padrões é o escopo de atuação do padrão, o qual especifica se a estrutura de colaboração é definida estaticamente por classes (escopo de classes) ou dinamicamente por objetos (escopo de objetos). O escopo de classes preocupa-se como o modo pelo qual as responsabilidades entre classes e subclasses estão distribuídas hierarquicamente, para a realização de uma determinada funcionalidade. Em geral, o escopo de classes baseia-se em classes abstratas que definem algum protocolo, que é completado ou implementado por classes concretas. O escopo de objetos preocupa-se com as formas de combinação ou composição de objetos. Os padrões de objetos são mais flexíveis que os padrões de classe, pois composições de objetos são definidas e modificadas dinamicamente, enquanto que em relacionamentos de herança, isso pode ser realizado apenas estaticamente (ZANCAN, 1997).

Outro critério de classificação é o propósito do padrão, o qual especifica o tipo de problema do *framework* no qual o padrão se aplica. Os padrões criacionais tratam do processo de criação dos objetos. Os padrões comportamentais caracterizam a forma pela qual classes e objetos interagem e distribuem responsabilidades do sistema. Por fim, os padrões estruturais estabelecem a forma pela qual classes ou objetos são compostos e usados quando a solução do problema envolve um projeto complexo.

Acredita-se que os padrões auxiliem consideravelmente no processo de documentação de um projeto, pois estes, em geral, representam as características estáticas e dinâmicas do projeto em questão.

Os padrões representam, portanto, abstrações de alto nível, que documentam soluções bem-sucedidas de projeto e são fundamentais para o reuso no desenvolvimento de sistemas de software orientados a objetos (SOMMERVILLE, 1996).

Os padrões de projeto formam um elo crítico das informações de projeto que não são providas por meio das metodologias de desenvolvimento de software OO. Conforme a Figura 2.2, os padrões podem ser úteis tanto em projetos de sistemas como também em projetos de *frameworks*. Os padrões podem ajudar a diminuir o intervalo entre métodos de projeto OO (que identificam e definem os objetos e seus relacionamentos em um sistema) e a arquitetura do projeto do sistema (onde se verificam as interações entre grupos de objetos pertencentes ao sistema).

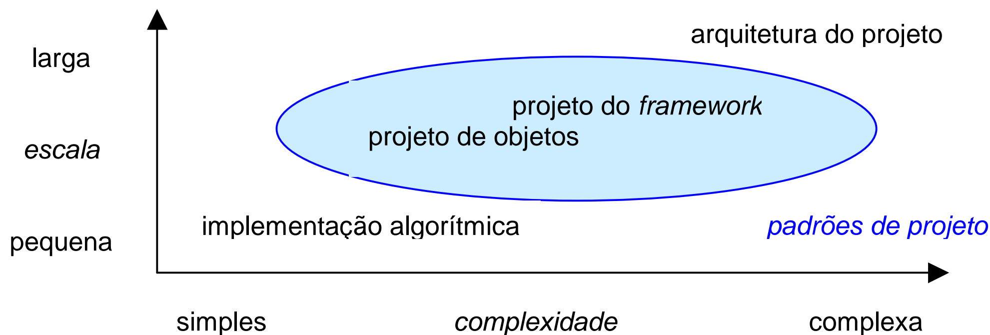

FIGURA 2.2 – Projeto de Sistemas de Software OO

Se durante a fase de manutenção o projetista dispõe da documentação sobre padrões, certamente possui à disposição um forte recurso sobre a informação. O que muitas vezes acontece é que, além de o projetista não conhecer com precisão as funcionalidades dos padrões, estes também não são devidamente documentados.

2.3 Engenharia Reversa e Re-Engenharia

A engenharia reversa tem como objetivos extrair informações da especificação de um software para posterior análise. A engenharia reversa resume-se no processo de análise de um sistema para identificar os componentes deste, bem como seus inter-relacionamentos, no intuito de criar representações do sistema em uma outra forma a níveis mais altos de abstração (CHIKOFSKY, 1990).

A engenharia reversa restringe-se a investigar sistemas e representá-los a fim de analisar seus comportamentos. A adaptação de um sistema está além da engenharia reversa, mas dentro da extensão de renovação do sistema em análise. A re-engenharia focaliza seus objetivos em construir para uma descrição semântica equivalente ao mesmo nível de abstração. Ferramentas para re-engenharia incluem como recursos, entre outros, geradores de diagramas, geradores de referência-cruzada e reflexão computacional.

Em relação à recuperação do domínio de conhecimento, é utilizada a informação externa para fazer uma descrição equivalente de um sistema a um nível mais alto de abstração. Portanto, mais informações que apenas o código fonte da aplicação são utilizadas.

Um aspecto essencial de reestruturar uma aplicação é que o comportamento semântico do sistema original e o do novo sistema devem permanecer o mesmo. O propósito da re-engenharia é estudar o sistema, fazendo uma especificação em um nível

de abstração mais alto, acrescentando novas funcionalidades a esta especificação (PETERS, 2001).

Pretende-se, portanto, utilizar os processos de engenharia reversa e re-engenharia sobre uma aplicação desenvolvida na linguagem Java, no intuito de facilitar o processo de entendimento desta aplicação em tempo de execução.

2.3.1 Reflexão Computacional

Reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre (possivelmente afetando) as suas próprias computações (MAES, 1987). A reflexão computacional gera informações consideráveis, levando-se em consideração um processo de re-engenharia.

A reflexão computacional define, portanto, uma arquitetura em níveis, denominada arquitetura reflexiva. Em uma arquitetura reflexiva, um sistema computacional é visto como incorporando dois componentes: um representando o objeto, e outro a parte reflexiva. Uma arquitetura reflexiva tem por objetivo demonstrar as seguintes características:

- que o próprio software possa analisar seu comportamento, sua forma de execução e a estrutura dos dados com os quais opera, e a partir destas informações adaptar-se para tentar maximizar sua execução;
- que exista um nível base para o desenvolvimento de uma aplicação, e que chamadas de procedimento neste nível base sejam automaticamente desviadas para um outro nível (metanível), que se preocupe com as atividades do primeiro.

Uma arquitetura de metanível provê duas interfaces para os níveis de funcionalidade distintos, porém conectados: uma interface de nível base que provê funcionalidade do domínio e uma interface de metanível que permite que o comportamento, a forma, ou a implementação da interface de nível base sejam manipulados, regulados ou influenciados (RAO, 1991).

A transferência do fluxo de execução do nível base para o metanível pode ser feita de forma transparente ou por meio de uma API que permita que um objeto no nível base se comunique com um metaobjeto. Quando um objeto se comunica com um metaobjeto, há dois passos a serem investigados: a materialização, que representa o ato de disponibilizar o estado e estrutura de um objeto para que um metaobjeto consiga atuar sobre ele; e a reflexão, que caracteriza o ato de um objeto ter seu estado, estrutura ou comportamento alterados por um metaobjeto. Verificam-se, portanto, duas propriedades do metanível: se ele tem acesso à representação no nível base e, também, se ele consegue alterar esta representação.

É importante ressaltar que as invocacões a metaníveis, bem como as reflexões de estado, estrutura ou comportamento, podem ser feitas de forma dinâmica, em tempo de execução.

Deve-se levar em consideração que a reflexão computacional por si só não provê flexibilidade ou capacidade incremental à aplicação. A combinação de reflexão com orientação a objetos, segundo Singh (2001), exige que:

- se defina um conjunto de tipos de objetos e operações sobre eles; este conjunto possui um conjunto de comportamentos denominado protocolo;
- se defina um comportamento padrão denotado por classes e métodos;
- sejam feitos ajustes incrementais no comportamento padrão por meio de herança/especialização.

Por fim, verifica-se que a reflexão computacional caracteriza uma importante ferramenta de verificação e avaliação na utilização de software OO.

2.4 Conclusões

A reutilização de elementos de software orientados a objetos enfatiza a construção de novos componentes com o menor esforço possível, pois, em muitos casos, estes componentes compartilham a maior parte das funcionalidades. Uma outra vantagem da reutilização é a adaptação dinâmica das aplicações. As classes podem apresentar um comportamento bem específico e ser utilizadas através de uma interface bem definida. Este mecanismo estabelece que os clientes externos não necessitam ter conhecimento sobre a estrutura e implementação de serviços dos componentes envolvidos.

Sob esse enfoque, adota-se a utilização de *frameworks* em projetos orientados a objeto. A arquitetura de um *framework* útil é composta por dois elementos: um conjunto de classes que capturam o domínio do problema e o controle das instâncias destas classes. A existência do fluxo de controle é o que distingue o *framework* de uma biblioteca de classes e permite, portanto, que o cliente, além de utilizar o *framework*, também possa completá-lo com seus serviços, caracterizando o mecanismo de reutilização.

Para ajudar, ainda, na reutilização, é importante definir uma arquitetura utilizando padrões de projeto. Esta consiste em encontrar os padrões capazes de mostrar a solução para o domínio do problema a partir dos requisitos definidos. Trata-se de uma tarefa árdua e, a princípio, obscura, mas, à medida que os padrões são aplicados à arquitetura, essa vai completando-se e soluções vão sendo encontradas (FREITAS, 2000).

A atividade de manutenção consiste em três etapas: entendimento, modificação e revalidação do sistema. As etapas de entendimento e modificação estão relacionadas com a disponibilidade das informações do sistema de software, ou seja, apóiam-se na existência, consistência, completude e atualização correta dos documentos que o compõem (SCHNEIDEWIND, 1987).

A engenharia reversa procura contribuir para o aumento da manutenibilidade do software, fornecendo informações, primeiramente utilizadas no entendimento e posteriormente, na modificação e revalidação do software. Assim, a engenharia reversa visa à recuperação de informações a partir dos documentos do software, visando a obter sua representação em um nível mais alto de abstração e de forma mais clara. O processo de engenharia reversa não é trivial e exige um alto custo pessoa/tempo para sua realização, devido ao grande volume de informações envolvidas durante o desenvolvimento do processo. Além disso, manter a consistência dos relacionamentos existentes entre as informações recuperadas é crucial para que o processo seja eficiente e permita um melhor entendimento do sistema verificado.

Para manutenção, utilização e reimplementação, os projetistas de sistemas freqüentemente precisam examinar, ainda, o código fonte de uma aplicação para compreenderem o software desenvolvido. A habilidade para aprender e entender o código fonte é ampliada visualizando-se os sistemas em níveis mais altos de abstração, em lugar de vê-los como coleções nebulosas de classes e implementação de métodos (BANSYA, 1998).

Assim, no intuito de apoiar o processo de manutenção de software OO, este trabalho propõe a descrição de uma ferramenta, construída em linguagem Java, que tem por objetivos disponibilizar a visualização das aplicações, em tempo de execução, na busca de descobrir, identificar e classificar grupos de classes/objetos relacionados na aplicação. Essas visualizações representam, em geral, padrões de projeto conhecidos que executam tarefas na aplicação e demonstram, portanto, um nível de abstração

elevado no apoio ao processo de manutenção. Para que a ferramenta alcance este nível de abstração, faz-se necessário utilizar os processos anteriormente descritos.

Durante os próximos capítulos serão apresentadas as estruturas utilizadas na ferramenta, bem como interfaces e características de funcionamento.

No capítulo seguinte, faz-se um referencial teórico com o objetivo de abordar diversos trabalhos que enfatizam contribuições nas áreas citadas neste capítulo, cuja função é propor um embasamento teórico.

3 Referencial Teórico

O desenvolvimento de sistemas de software de boa qualidade é uma preocupação constante entre os projetistas. A produção de sistemas é considerada uma tecnologia recente em comparação a outras atividades industriais. A produção e manutenção ainda consomem muito esforço e apresentam um custo final elevado. Levando em consideração esta motivação, verifica-se que vários trabalhos se preocupam com o fator qualidade em se tratando de sistemas de software OO.

3.1 Automatização no Desenvolvimento de Sistemas de Software OO

A complexidade crescente dos sistemas de software e os problemas advindos do seu desenvolvimento e manutenção realçam a insuficiência de métodos formais e informais para construir tais sistemas. Esses problemas manifestam-se nos sistemas de computador que são freqüentemente intratáveis, incertos, inflexíveis e, conseqüentemente, difíceis de manter. A seguir faz-se uma seleção de trabalhos que visam a automatizar o processo de desenvolvimento de sistemas de software OO. Justifica-se a apresentação destes trabalhos porque todos eles preocupam-se com a correção e verificação de informações, vindo a servir como subsídio para o processo de detecção da ferramenta proposta.

3.1.1 Agarwal

Os usuários exigem, freqüentemente, segurança para sistemas computacionais, pois eles percebem que a maioria dos fracassos ocorre devido à especificação pobre de projeto. Esta preocupação ocasionou o aparecimento de várias metodologias de desenvolvimento e ambientes automatizados. O trabalho de Agarwal (1995), intitulado "*PATHOS - A Paradigmatic Approach To High-level Object-oriented*", apresenta uma ferramenta que visa a ajudar o desenvolvimento de sistemas, não só conduzindo a uma boa concepção, mas também representando o conhecimento sobre as relações de negócios em tempo de execução.

O autor enfatiza que as linguagens de programação oferecem suporte:

- ao encapsulamento – é o esforço de esconder as informações, o qual ajuda a minimizar as interdependências entre módulos escritos separadamente;
- ao reuso – é o processo de utilizar partes do software em outras atividades diferentes daquelas para as quais foi projetado;
- e à extensibilidade – é o processo de somar novas funcionalidades a um software existente.

O trabalho propõe o aumento das funcionalidades, adicionando os conceitos de imagem e paradigma:

- a imagem é uma idéia ou concepção de experiências passadas que são armazenadas em si mesmas. As mensagens consistem em informações e o valor das mensagens é determinado pela quantidade de mudanças que se fazem em uma imagem.
- o paradigma é um modelo aceito, interpretado e aplicado por diferentes pessoas. Todos os projetistas possuem uma imagem inicial de um processo de desenvolvimento de software que caracteriza as experiências iniciais. Os proprietários das imagens encontram uma série de mensagens que os informam sobre o sucesso ou incapacidade de modificar os conteúdos das imagens.

O desenvolvimento do processo por PATHOS auxilia a focalizar e construir estruturas que são capazes de se adaptar ao ambiente dinâmico. O problema é expresso em termos das expectativas do usuário. A solução para um problema é um sistema de software adaptável, que realmente satisfaz a essas expectativas com restrições de custo. O problema do usuário será traduzido de uma declaração do problema específico em instruções válidas de computador. Os três passos a serem seguidos são: especificação dos meios de comunicação, especificação da hierarquia da informação e especificação do nível de abstração da informação.

A Figura 3.1 demonstra um ciclo de vida típico de desenvolvimento de software OO que caracteriza os princípios de paradigmas de PATHOS. Os três processos maiores são a definição de: atividades técnicas, atividades técnicas de controle e atividades de controle organizacionais.

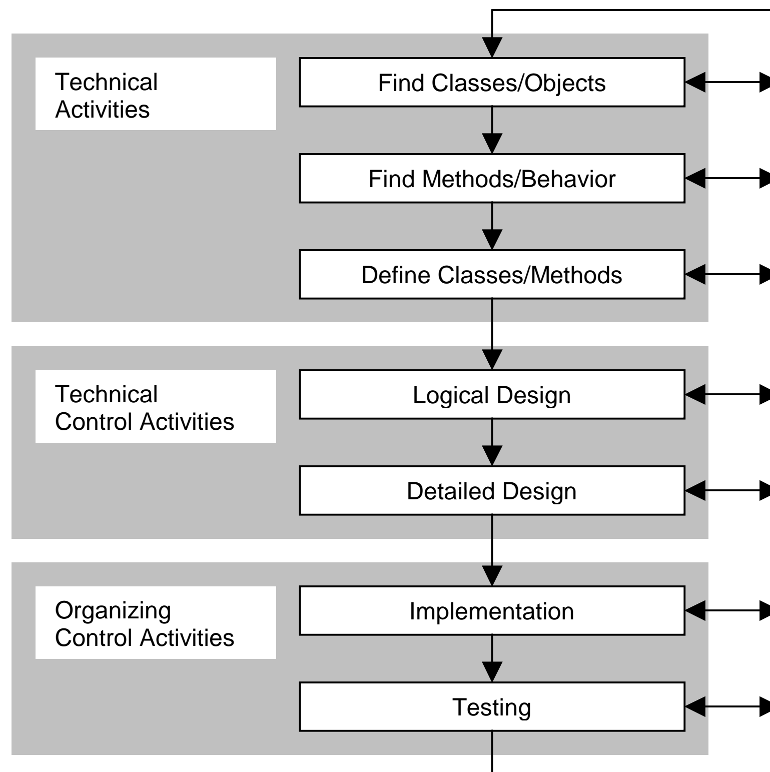

FIGURA 3.1 - Desenvolvimento de Software OO com PATHOS (AGARWAL, 1995)

A definição das atividades técnicas organiza os processos para desenvolver um sistema de software seguro. As atividades técnicas operam em um ambiente caracterizado pela relatividade das informações. Nesta etapa cada classe/objeto traduz contribuições de acordo com planos e procedimentos pré-especificados. É importante identificar relações entre vários tipos de objeto e definir as propriedades comuns deles, ou seja, incluir a compreensão do comportamento dos objetos.

As atividades técnicas de controle visam a desenvolver e manter economicamente o sistema de software, proporcionando modificações caso necessário. Na realidade, as atividades de controle podem ser definidas como a integração e direção das tarefas técnicas por meio de um comportamento coordenado. A meta é integrar o comportamento técnico à extensão do sistema dentro das limitações de tempo e orçamento. Para realizar esta etapa, o controle técnico precisa manter um registro histórico da evolução do sistema, sendo que o registro ajudará a monitorar o progresso do desenvolvimento deste.

Por fim, as atividades de controle organizacional acontecem no mais aberto dos três ambientes, no qual se fazem a implementação e testes do sistema. A etapa considera que toda organização possui metas a cumprir e tais metas representam os recursos disponíveis para a construção de componentes. Portanto, é importante assegurar que a produção dos respectivos componentes seja utilizável.

O trabalho de Agarwal (1995) representa um auxílio ao desenvolvimento de sistemas de software OO. A ferramenta acompanha o projetista durante o ciclo de vida de um sistema de software apoiando e verificando a sua evolução. Algumas exigências são necessárias, como: o projetista deve concentrar-se no reuso das informações como parte do projeto; aplicações com classes similares devem examinar o controle técnico para que se possam generalizar tais informações; e todos os componentes que são colocados em uma biblioteca para reuso devem ser validados pelas características da organização.

3.1.2 Jiason

Geralmente a análise de requisitos pode ser visualizada a partir de dois pontos: a aquisição dos requisitos e a sua especificação formal. As duas tarefas não são necessariamente seqüenciais, mas sim freqüentemente relacionadas. O processo de aquisição leva à construção de um modelo preliminar de especificação do sistema de software, que deverá gradativamente ser elaborada e expressa em uma linguagem bem formada.

A proposta descrita por Jiason (1996), intitulada "*NDHORM: An OO Approach to Requirements Modeling*", caracteriza-se pela captura e simulação dinâmica de relacionamentos entre objetos do mundo real, proporcionando refinamentos sucessivos durante o processo de análise de requisitos. O trabalho apresenta, também, uma representação formal para caracterizar o domínio do conhecimento, a qual torna mais práticos e precisos os processos de transformação durante a modelagem do sistema.

O modelo NDHORM "*NanDa* (Nanjing University) *Hierarchical Object-Oriented Requirements Model*" representa um modelo multivisão que possui três partes:

- modelo objeto-relação – é baseado na idéia de objetos e seus comportamentos, levando em consideração as trocas de mensagens entre estes. O modelo adota um método de comportamento direcionado que significa a avaliação do comportamento de cada objeto e sua decomposição sucessiva, até que se obtenha uma descrição precisa para os objetos avaliados. Em muitos casos, um objeto pode ser subdividido em um ou vários subsistemas;
- modelo classe-relacionamento – representa a descrição das classes por meio de seus atributos, serviços e colaborações. O modelo prevê três tipos de classes: classe atômica, classes composição e superclasse; e dois tipos de colaborações: agregação e subclasse;
- modelo estado-transição – representa uma classe como uma máquina de estados finitos. É composto de um número finito de estados e transições que representam o ciclo de vida de um objeto.

Cada modelo possui um conjunto de diagramas predeterminado e, em adição aos modelos, o trabalho prevê o uso de um dicionário de classes e objetos. O objetivo do dicionário é representar o conhecimento sobre um domínio particular de problema. O trabalho adota uma linguagem de descrição formal baseada na linguagem Z. A forma de entrada das classes e objetos é definida conforme segue:

```
object    <object-name>
          <attribute-name>=<value>
          {,<attribute-name>=<value>}*
endobject

class     <class-name>
          [<informal-description>]
          [<constant-definition>]
          [<type-definition>]
          {attribute<attribute-name>:<type-name>
                    [constraint<attribute-constraint>]}+
          {service<service-name>
                    input<input-arguments>
                    output<output-arguments>
                    [description<funcional-description>]
                    funcionality
                              pre<pre-assertion>
                              post<post-assertion>
                    [constraint<service-constraint>]}+
endclass
```

A construção do modelo NDHORM é auxiliada por uma ferramenta que provê um editor gráfico para edição dos modelos de objetos, classes e estado-transição. A ferramenta propõe o gerenciamento do dicionário, correção e verificação automática das informações. Finalmente, verifica-se que o modelo de construção é fácil de entender e executar mediante um processo de modelagem por refinamentos sucessivos. Destaca-se que a verificação das informações pode ocorrer com alguns erros devido ao grau de complexidade dos sistemas envolvidos.

3.1.3 Freitas

Com o objetivo de auxiliar o processo de formalização no desenvolvimento de sistemas de software orientados a objetos, o trabalho de Freitas (1998), intitulado "Formalização de Heurísticas para o Apoio a Modelagem de Sistemas Orientados a Objetos", propõe heurísticas que possibilitem tornar mais automáticas as atividades de modelagem e projeto. As heurísticas propostas foram classificadas em grupos, representando contribuições para a construção de classes e objetos, tendo sido ilustradas e demonstradas, em diversos casos, por meio de exemplos. Por fim, o trabalho apresenta uma ferramenta que visa a auxiliar o projetista na definição de um bom projeto.

As heurísticas propostas no trabalho foram extraídas de decisões de projeto que se apresentam em diferentes domínios e foram enunciadas mediante lógica, no intuito de apresentá-las de maneira formal, com vistas a sua utilização em uma ferramenta de especificação de projetos. Foi construído, também, um dicionário de dados, no formato de tabelas, o qual serve para armazenamento de informações a respeito das classes e colaborações envolvidas na aplicação em estudo.

Um exemplo de heurística desenvolvida no trabalho foi: *esconder os atributos públicos dentro da classe.* Esconder informações, nos níveis de projeto e implementação, traz um grande número de benefícios para a segurança do sistema. Quando um atributo é público, há uma dificuldade para determinar que segmentos do sistema são dependentes do atributo (FREITAS, 1998).

A lógica, descrita a seguir, propõe a identificação de atributos públicos nas classes do sistema e prevê a modificação destes para privados, proporcionando a criação de métodos *get* e *set* do tipo público para acesso aos referidos atributos.

A especificação determina, para todas as classes do dicionário de dados, a procura pelos atributos que forem definidos como públicos. Cada atributo público é transformado em privado e, finalmente, são criados métodos para acesso a estes.

```
∀ Classes ∈ Dicionário
   c | c ∈Classes
     t | ∃a ∈ Atributos(a [ Nome_Classe ] = c [ Nome_Classe] ^
                         a [ Tipo_Atributo ] =  Atributo_Público ^  t = a)
                         t [ Tipo_Atributo ] = Atributo_Privado
                         Entra Nome_Novo_Método
                         s | ¬ ∃m ∈  Métodos (m [ Nome_Classe ] = c [ Nome_Classe] ^
                                             m [ Nome_Método ] =  Nome_Novo_Método ^
                                             s = m)
                        Label
                            Seleciona Caracterísitca_Método (Get, Set)
                            Entra Código_Novo_Método
                                                 s [ Nome_Método ] = Nome_Novo_Método
                             s [ Tipo_Método ] = Método_Público
                             s [ Código ] = Código_Novo_Método
                             s [ Carcterística ] = Característica_Método
                         retorna label
```

No trabalho são, portanto, listadas várias heurísticas relacionadas tanto a projeto como a implementação. Após, um assistente de projeto é descrito. A ferramenta propõe um auxílio ao projetista para diagnosticar e verificar possíveis modificações de projeto, com o intuito de melhorar a qualidade deste.

A ferramenta dispõe de um módulo de manipulação dos elementos envolvidos como classes, objetos e relacionamentos. Esse módulo pode ser ativado via interface gráfica ou textual e conta com opções de manipulação, persistência, remoção e edição, entre outras.

O segundo módulo define as regras que são aplicadas aos elementos criados. As regras traduzem as heurísticas estudadas no trabalho. As heurísticas foram implementadas e definiram um determinado comportamento de acordo com as suas características, isto é, algumas têm por função advertir o projetista sobre algum problema; outras, advertir e propor modificações.

A ferramenta assistente de projeto caracteriza-se pela existência de cinco blocos principais: um módulo extrator de informações, um módulo responsável pela aplicação das regras, um módulo de geração de aplicações gráficas, um módulo de visualização e um módulo de geração de código.

A partir da descrição fonte, produzida em alguma linguagem de programação, pelo próprio projetista, o módulo de extração de informações tem por objetivo extrair do texto as informações relevantes para a aplicação das heurísticas existentes. O módulo de extração cria uma base de dados (um Dicionário) com estas informações.

O módulo responsável pela aplicação das regras aplica sobre a base de dados as heurísticas existentes proporcionando, por vezes, modificações nas informações existentes.

Finalmente, verifica-se que o autor estabeleceu algumas idéias para otimizar a construção de projetos, acreditando que limitar quantitativamente determinadas transformações não represente uma boa idéia porque cada aplicação possui características e necessidades próprias a serem desenvolvidas.

3.2 Geração e/ou Formalização de Padrões de Projeto

Embora os projetistas de sistemas de software freqüentemente utilizem padrões de projeto, muitas vezes isto é feito de forma inconsciente, e não por meio do projeto do sistema. Sempre que existe a utilização de um padrão, está se aplicando uma aprendizagem baseada em um experimento, ou seja, tende-se a imitar as estratégias e ações que fizeram aquele experimento próspero no passado. A preocupação com a geração e/ou formalização de padrões de projeto levam à motivação para a realização de diversos trabalhos. A seguir, faz-se a descrição de alguns trabalhos relacionados à especificação de padrões de projeto.

3.2.1 Budinsky

Os padrões de projeto elevam o nível de abstração na construção de sistemas de software OO. Porém, em muitos casos, os mecanismos de implementação dos padrões são deixados de lado pelos implementadores. O trabalho de Budinsk (1996), intitulado "*Automatic Code Generation from Design Patterns*", descreve a arquitetura e implementação de uma ferramenta que automatiza a implementação de padrões de projeto. A ferramenta propicia ao usuário informações para um determinado padrão a partir do qual a ferramenta gera todo o código necessário.

A experiência no projeto de sistemas distingue um projetista novato de um experiente. É muito difícil ao experiente transmitir suas experiências capturadas ao longo do tempo. Os padrões de projeto capturam a experiência de projetistas experientes e descrevem uma solução para um problema que ocorre periodicamente, de modo sistemático e geral (BUDINSK, 1996).

Além de uma descrição do problema e sua solução, o projetista precisa entender e adaptar a solução a possíveis variantes desse problema. Conseqüentemente, um padrão de projeto explica a aplicabilidade e conseqüências da solução. Mas freqüentemente há muitos detalhes a considerar em um padrão, o que pode resultar em formas diferentes de implementação. É então verificado que os projetistas duplicam esforços a cada vez que for necessária uma implementação.

Esse trabalho descreve uma abordagem ao problema em questão, na qual é apresentada uma ferramenta para a geração automática de código para padrões de projeto. O gerador utiliza uma quantia de informação provida pelo usuário e oferece uma referência *on-line* integrada ao desenvolvimento.

A ferramenta exibe as características de um padrão como a intenção, motivação, aplicabilidade, estruturas participantes, entre outras, em janelas separadas. Essas janelas refletem as seções correspondentes na bibliografia de Gamma (1994). O projetista tem acesso às janelas das seções e pode gradativamente fornecer informações, sendo permitido saltar as seções que julgar menos importantes. Além de texto na janela, o projetista pode anexar hipertexto adicional, que une as informações das diferentes seções.

Após a entrada das informações, o projetista pode acessar a janela de geração de código, a qual é integrada com as outras janelas de onde são colhidos os detalhes para a geração do código.

A geração de código automática soma uma dimensão de utilidade para projetar padrões. O projetista pode verificar como conceitos do domínio podem traçar o código que implementa o padrão. Mas a ferramenta é limitada, pois explora só uma fração do conhecimento de implementação de padrões porque não apóia a análise do domínio, especificação de requisitos, documentação e depuração.

3.2.2 Wild

Projetistas de software experientes utilizam padrões em larga escala no que se refere a aspectos de projeto. Os padrões tendem a desenvolver sistemas mais produtivos, com maior probabilidade de êxito no produto implementado.

O trabalho de Wild (1996), intitulado "*Instantiating Code Patterns - Patterns Applied* to *Software Development*", apresenta a arquitetura de uma ferramenta denominada SNIP, cujo objetivo é auxiliar os processos de definição e instanciação de padrões de projeto, por meio de códigos padrões de implementação.

Os padrões de projeto são lógicos por natureza. Eles representam restrições e estratégias que são pertinentes a várias implementações. Em contrapartida, os códigos de implementação dos padrões são físicos. Eles focalizam como uma estrutura particular ou sucessão de ações são utilizadas dentro dos mecanismos específicos de uma linguagem de programação. Os projetistas em geral, fazem a codificação dos padrões à mão, utilizando uma amostra de código de acordo com a linguagem utilizada.

Os ambientes para desenvolvimento de sistemas operam geralmente, sob mau tempo e sujeitos a pressões de índices de qualidade. Como resultado, os códigos que deveriam ser utilizados, em muitos casos são distorcidos não porque o projetista desconheça os padrões, mas sim pelas exigências do ambiente. Portanto, para se fazer uso de padrões de código, precisa-se de um modo fácil para caracterizar objetos durante o projeto e permitir que essas características influenciem o código produzido (WILD, 1996).

SNIP é uma ferramenta que permite a implementação de código padrão a partir da derivação do modelo de objetos, ou seja, define códigos que são importantes, baseados em seus próprios padrões locais e características dos objetos. A ferramenta provê a capacidade de instanciar essas codificações para qualquer conjunto de objetos e permite habilitar códigos a partir de regras predefinidas.

Para utilizar SNIP, precisa-se de uma estratégia de implementação bem definida. É necessário verificar as características dos objetos e suas colaborações, decidindo o modo como o código deve ser criado para cada um deles. Dada a estratégia, cria-se um conjunto de regras que são armazenadas em um arquivo de modelo. Vários arquivos de modelo servem como ponto de partida para boas implementações. O projetista pode desenvolver o modelo iterativamente a partir da interface de usuário, e uma vez satisfeito com o código gerado, poderá adicionar essas características para uso em construções não-iterativas.

O autor desse trabalho apresenta um pequeno estudo de caso em que utiliza como exemplo a tarefa de administrar dinamicamente a alocação de objetos. Primeiro, são definidas algumas restrições e, após, a ferramenta procura por um modelo semelhante para prover a implementação. A ferramenta apresenta facilidades na criação de código a partir de informações de projeto em alto nível. O foco é na utilização de trechos de código mais oportunos para a resolução de determinados problemas.

Concluindo, SNIP é uma ferramenta que permite o controle da geração de código. É flexível o bastante para que se possam definir códigos padrões que se relacionam com um modelo de objetos proposto. A identificação de estratégias consiste em produzir regras que aumentam a qualidade para as próximas gerações de código, demonstrando assim uma grande vantagem de produtividade. SNIP não provê um modo revolucionário para produzir código, mas sim oferece a oportunidade de automatizar o processo de geração de padrões.

3.2.3 Eden

Os padrões de projeto representam um conjunto principal de heurísticas para o projeto de sistemas de software OO. Os padrões são extensamente utilizados em numerosos domínios de aplicações. Porém, caracteriza-se a falta de formalismos satisfatórios para a automatização dos aspectos técnicos de implementação.

A proposta de Eden (1997), intitulada "*Automating the Application of Design Patterns*", apresenta uma abordagem de especificação rigorosa de padrões de projeto e um protótipo de ferramenta que automatiza a maioria da aplicação destes.

Uma descrição dos padrões de projeto em uma biblioteca de classes bem projetada é o método mais eficiente de resumir sua descrição e fornecer informações confiáveis a respeito. Por essas razões, muitos trabalhos são realizados na tentativa descobrir e descrever padrões. Eden (1997) cita três tipos de atividades na pesquisa de padrões:

- descoberta ou definição de padrões;
- classificação e descrição de padrões;
- aplicação de padrões.

No entanto, estas atividades caracterizam-se como boas candidatas para um processo de automatização. A aplicação manual de muitos padrões, freqüentemente, consome tempo e muitas vezes apresenta erros. Em geral exige-se que um projetista que utiliza uma linguagem de programação aplique definições utilizadas repetidamente. Sob este enfoque, a proposta tenta automatizar a aplicação e validação de padrões em um contexto particular.

A solução para os problemas relacionados à aplicação de um padrão, aparentemente, só está dentro do alcance se existe um modo de formalizar as operações e os atores que participam na implementação do respectivo padrão. Até a elaboração desse trabalho, o formalismo utilizado é descrito por meio de diagramas de classes, diagramas de objetos e um texto representado pelo código fonte, o qual ilustra uma implementação particular.

Eden (1997) considera que nenhuma dessas ferramentas está completa como referência para projetar padrões, pois esses são, principalmente, descritos através de idioma natural. A utilização do idioma natural utilizado como o fundamento da especificação de padrões é caracterizado nos trabalhos de Alexander (1977), que propiciou a revolução dos padrões.

Portanto, dentro do domínio de sistemas de software, a falta de formalismos prejudica a expressividade das especificações de padrões e restringe a automatização e validação da aplicação desses padrões. A abordagem desse trabalho caracteriza a especificação rigorosa mediante um algoritmo para a aplicação de cada padrão.

O exemplo citado a seguir descreve um algoritmo que implementa uma variação particular do padrão *Prototype* em C++. O padrão *Prototype* especifica o tipo dos objetos que serão criados dinamicamente, utilizando uma instância *default* que será reproduzida cada vez que um novo objeto for criado (Gamma, 1994). Por meio do padrão *Prototype,* é possível incorporar classes com novas funcionalidades após a construção da aplicação ou carregá-las apenas em tempo de execução. O padrão declara uma interface para reprodução (*clone*) e uma classe cliente, a qual cria novos objetos via solicitação ao objeto do tipo *Prototype* para reproduzir-se através do método *createClone*.

A especificação de Eden prevê a criação de um objeto *cloned_list*, responsável pelo armazenamento dos objetos clone através de uma lista. A lista é consultada sempre que houver uma solicitação de objeto *clone* e, caso este já exista, será retornado. Caso

contrário, será criado um novo objeto que será armazenado na lista. Para cada objeto é, portanto, necessário verificar seus atributos (dados membros) para proceder à sua duplicação. Em caso de atributos que representem tipos básicos, o autor utiliza o método *copy*. Já para o caso de ponteiros, é necessário utilizar o método *clone* para proceder à criação de novas áreas de memória, para que os ponteiros dos objetos duplicados não apontem para os mesmos locais.

```
Add the routine clone to class C:
Set the routine's return value to: C*
Set the routine's arguments to: List cloned_list of pairs of objects and their
        clones (both of type void*)
Set the routine's  name to be: "clone"

Set the routine's body to be the following:
        if (cloned_list) contains this object
                then return its clone, else continue.
        Create a new object result of class C on the free-store
        Add the pair <this,result>  to cloned_list
        For each data member m of result do:
                result.m := duplicate(this.m)
                where the interpretation of the function duplicate depends on the type of its argument:
                        If m is of a pointer to a fundamental type and m≠nil then
                                return a newly allocated copy of m
                        else if m is of a pointer to a class and m≠nil then
                                recursively call m.clone(cloned_list)
                        else if m is held by value then
                                use the assignment operator for copying  m
        Return result

Add another (overloaded) version of clone to class C:

Set this routine's signature to be the same as the other's, only this one has no  arguments.
Set the routine's body to have the following statements:
        Create a local list<void*,void*> object l
        Return the result of calling clone(l)
```

Se os padrões de projeto podem ser especificados por meio de um algoritmo, é natural usar uma linguagem de programação como linguagem pela qual os algoritmos dos padrões venham a ser expressos. O termo metalinguagem define a linguagem pela qual é possível manipular outra linguagem. A meta-linguagem proposta é OOPL (*object-oriented program language*), porque nesta abordagem os padrões de projeto são tratados como aplicações que manipulam outras aplicações.

A automatização de aplicações com padrões de projeto faz-se, então, possível pela formulação dos padrões na metalinguagem. No trabalho de Eden (1997), foi proposta uma ferramenta que lê declarações de classes na linguagem OOPL e permite gerar uma representação abstrata dessas classes, possibilitando especificar e aplicar padrões de projeto. A análise gramatical do texto em OOPL cria uma estrutura de dados que representa as classes, relações, objetos, declarações, e todos os elementos da linguagem objeto.

É importante ressaltar a necessidade de utilização da metalinguagem devido à facilidade de entendimento e adaptação por parte do projetista. A argumentação na proposta considera a metalinguagem como uma linguagem de 4ª geração ou uma linguagem de manipulação de ferramentas CASE.

A ferramenta necessita de que o projetista complemente e monitore as suas operações, ou seja, resumidamente, a ferramenta, de posse de uma descrição algorítmica gerada pela metalinguagem, faz a análise gramatical proporcionando uma representação abstrata. À representação abstrata são propostas modificações nas quais o produto final é a geração do código fonte, que pode ser manualmente editado. Um protótipo da ferramenta foi gerado utilizando *Smalltalk* como metalinguagem e *Eiffel* como

linguagem objeto. Por fim, o autor apresenta a necessidade de melhorias no que se refere à manipulação semântica dos padrões.

3.2.4 Lauder

A descrição precisa de um padrão de projeto sofre dois tipos de interferências: primeiramente, os padrões capturam instâncias específicas do desenvolvimento em lugar de sua própria essência. Assim, as características básicas do padrão são freqüentemente perdidas nos detalhes supérfluos dos exemplos específicos. Em segundo lugar, a descrição de um padrão de projeto existente é traduzida em diagramas relativamente informais, completados com anotações em idioma natural. Isto resulta em imprecisão e ambigüidade.

A proposta descrita por Lauder (1998), intitulada "*Precise Visual Specification of Design Patterns*", apresenta a especificação de padrões a partir de três modelos distintos: *role* (papel), *type* (tipo) e *class* (classe), conforme Figura 3.2. O modelo do papel apresenta o padrão em sua forma pura, capturando os detalhes essenciais. O modelo do papel é refinado por um modelo do tipo o qual adiciona restrições provindas do domínio específico em desenvolvimento. Conseqüentemente, o modelo do tipo é refinado pelo modelo da classe, o qual virá a gerar um modelo de desenvolvimento concreto. Por fim, é apresentada uma ferramenta que permite ao projetista utilizar os padrões em um nível mais alto de abstração, utilizando os modelos sem ambigüidades.

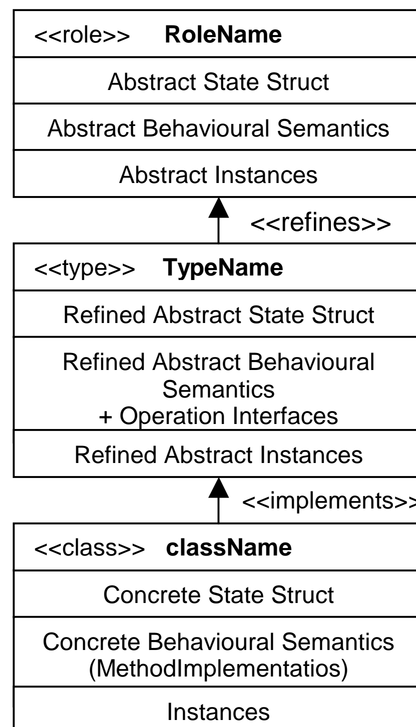

FIGURA 3.2 – Modelos de Especificação (LAUDER, 1998)

O autor apresenta o formalismo para representação do padrão *Composite*. O objetivo do padrão *Composite* é compor objetos em uma estrutura em árvore para representar uma hierarquia todo-parte. O padrão trata clientes e composições de forma igual e consiste de objetos *leaf* (primitivos) e *Composite*, que representam um conjunto de objetos *Components*. A Figura 3.3 representa um diagrama de restrições onde um elemento de um conjunto é representado por um círculo preto. Dois círculos pretos

conectados representam que um elemento deve residir em apenas uma das posições representadas pela conexão, e os arcos representam o relacionamento entre conjuntos e elementos.

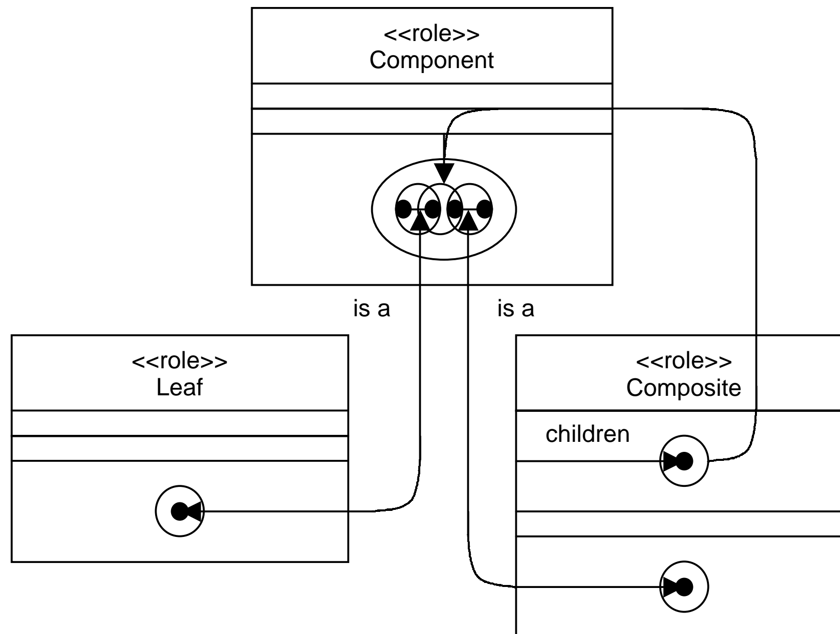

FIGURA 3.3 – Diagrama *Composite* (LAUDER, 1998)

O diagrama mostra, portanto, com clareza, que um elemento *children* possui uma relação com (é um conjunto formado por) elementos do tipo *Leaf* ou *Composite*. Cada elemento *Leaf* ou *Composite* representa um elemento simples e deve possuir existência única no conjunto formado em *children*.

O autor caracteriza, também, no diagrama de transição de estados, Figura 3.4, quando acontece a inserção de um elemento na estrutura *Composite*. Utilizando a mesma notação, é possível demonstrar a inserção do elemento no conjunto.

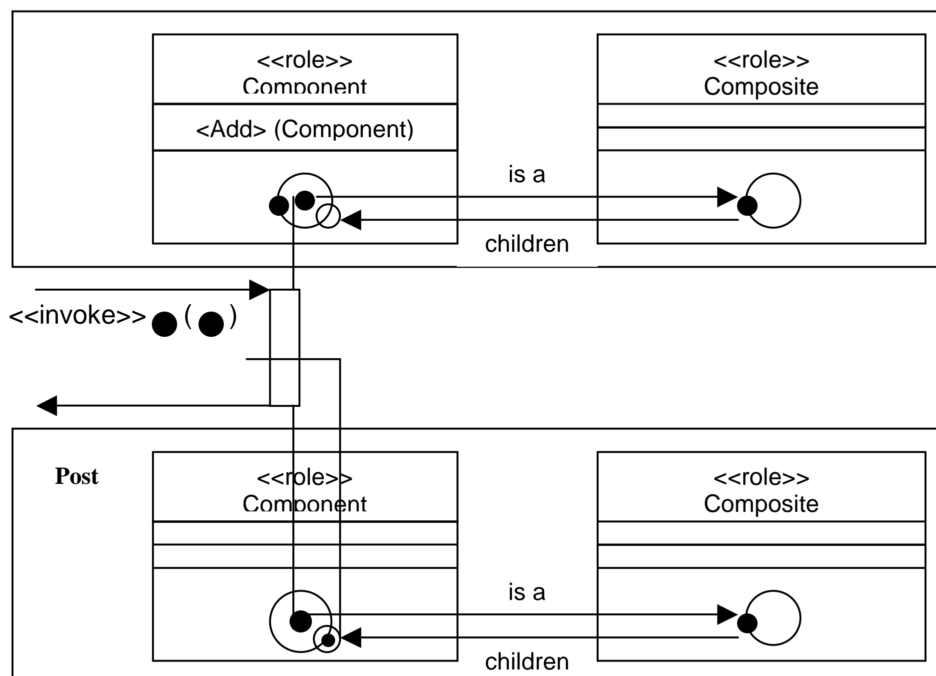

FIGURA 3.4 – Diagrama *Composite – Add()* (LAUDER, 1998)

Finalmente, o trabalho apresenta um grande avanço em notação de modelagem visual, com maior precisão, sem recorrer a símbolos matemáticos. A especificação não-ambígua de padrões puros permite a verificação de projetos inconsistentes e incompletos. A ferramenta permite projetar um catálogo de padrões, propondo os refinamentos necessários, combinando e aplicando características apropriadas ao domínio da aplicação. Um ponto a ser considerado é a grande dificuldade de expressar alguns padrões na especificação formal devido à riqueza de detalhes de alguns padrões.

3.2.5 Vlissides

É comum ver um conjunto de pequenos padrões que cooperam entre si em sistemas diferentes. O trabalho de Vlissides (1998), intitulado "*Composite Design Patterns*", faz uma abordagem sobre a combinação de padrões que cooperam nos serviços prestados. O autor argumenta que, quando padrões cooperam, a própria cooperação pode dar origem a problemas, como, por exemplo, a distorção, pois os padrões podem se expressar através de outros. O objetivo desse trabalho é justamente capturar a sinergia entre os padrões e torná-la explícita.

Vlissides (1998) apresenta uma lista de possíveis padrões os quais trabalham em conjunto, como: *Template Method-Factory Method*, *Prototype-Abstract Factory*, *Composite-Decorator*, *Composite-Flyweight*, *Composite-Iterator-Visitor*, *Composite-Strategy-Observer*, *Mediator-Observer* e *Command-Memento*.

O padrão *Template Method* define o esqueleto de um algoritmo na superclasse, propiciando a definição de alguns procedimentos pelas subclasses, sem mudar a estrutura do algoritmo. Já o padrão *Factory Method* encapsula o conhecimento sobre as classes que devem ser instanciadas e move o conhecimento para fora do *framework*. Portanto, *Template Method-Factory Method* é um padrão composto simples que apresenta sinergias elementares. A semelhança está no sentido de que os padrões adiam o comportamento para subclasses da aplicação. O autor considera que *Template-Method* não é específico sobre o comportamento e *Factory-Method* é altamente específico. Um *Factory-Method* sempre resume o processo de instanciação, e onde quer que se ache um *Factory*, um *Template* não pode estar distante. O contrário também é verdade. Isso é determinado porque os padrões são semelhantes em intenção e implementação. *Factory-Mehtod* serve freqüentemente para oferecer operações primitivas a *Template-Methods*. É importante salientar que ambos os padrões são baseados em classes ao invés de serem em objetos (instâncias), o que faz os padrões serem menos flexíveis, mas simples e eficientes.

Um outro exemplo proposto é *Composite-Decorator*, onde os dois padrões complementam um ao outro. *Composite* propõe a clientes tratarem, uniformemente, objetos individuais e composições de objetos. Já *Decorator* acrescenta dinamicamente responsabilidades a objetos através de composição. Eles igualam o compartilhamento do participante da classe básica *Component,* e os mesmos mecanismos para recursão da composição representam a igualdade entre os padrões. Vlissides (1998) afirma que é muito comum os padrões trabalharem juntos quando o projetista opta pela uniformidade das operações na classe *Component*.

O trabalho apresenta, portanto, uma seqüência de explicações sobre possíveis padrões que possam trabalhar em cooperação. O autor enfatiza que a combinação de padrões não pode descaracterizar o uso real destes. Por fim, o autor argumenta que muitas características são unidas em função da combinação, o que pode acarretar dificuldades no entendimento do sistema de software desenvolvido.

3.2.6 Freitas

Os padrões de projeto apresentam-se como boas soluções para problemas de desenvolvimento de sistemas de software dentro de um contexto particular. À medida que os padrões são aplicados, aumentam-se as facilidades na implementação de detalhes do sistema. O trabalho de Freitas (2000A), intitulado Disponibilizando o uso de "Padrões de Projeto através de um *Framework* em Prolog++", tem como objetivo apresentar a porção de um *framework* que disponibiliza a utilização de padrões de projeto em um ambiente que implementa os paradigmas de programação em lógica e orientação a objetos.

O trabalho visa a incorporar facilidades, como a construção de um *framework* que permita a utilização de padrões de projeto, em um sistema híbrido que une os paradigmas de programação em lógica e orientação a objetos.

O autor acredita que cada padrão possui uma função e estrutura preestabelecidas que evitam deduzir a função das classes e dos métodos no projeto a partir das interações. Outro fator importante considerado no trabalho é a definição de métodos abstratos e *hook*, que devem ser implementados nas subclasses das aplicações conforme necessidades do projetista.

Na seqüência do trabalho, o autor apresenta um fragmento do *framework*, demonstrando alguns métodos. É apresentado um exemplo para o padrão *Observer* o qual define uma relação de dependência de um para muitos entre objetos, de modo que quando um objeto muda seu estado, todos os seus dependentes são notificados e atualizados automaticamente. A existência de um atributo denominado *dependentes* define uma lista que conterá os objetos que devem ser notificados quando propostas mudanças de estado. Na classe definida, o método *when_created* responsabiliza-se pela inicialização de tal lista como vazia. Os métodos *adiciona_dependentes* e *remove_dependentes* inserem e excluem, respectivamente, objetos da lista *dependentes*.

```
when_created :- dependentes := [].
adiciona_dependentes(X) :- dependentes:=[X|@dependentes].
                                        /* o símbolo @ representa o conteúdo de um atributo */
remove_dependentes(X) :- exclui_dependente(X,@dependentes,L),
                              dependentes := L.
exclui_dependente(X,[X|T],T).
exclui_dependente(X,[H|T1],[H|T2]) :- exclui_dependente(X,T1,T2).
```

Outros métodos para compor a utilização do padrão *Observer* são demonstrados como, por exemplo, *alerta_dependentes e dependente_alertado.* Após a especificação, o autor faz a demonstração de utilização por meio de um pequeno exemplo.

O *framework* descrito nesse trabalho encontra-se em fase de implementação, sendo importante caracterizar a necessidade de adição de novas estratégias, para completar o modelo de utilização genérica dos padrões. Finalmente, acredita-se que o trabalho desenvolvido pode ser útil para os estudos de construção de ferramentas que auxiliem o projetista em atividades de projeto de sistemas de software.

3.3 Verificação e Identificação Automáticas de Padrões de Projeto

Levando em consideração os processos de manutenção, reuso, engenharia reversa e reimplementação, verifica-se que os projetistas de sistemas de software freqüentemente precisam examinar o código fonte para entender os detalhes do sistema trabalhado. Trabalhos mais específicos, considerando a verificação e identificação, na área de padrões de projeto, também foram realizados. A seguir, apresenta-se uma descrição de alguns trabalhos considerando o processo de engenharia reversa.

3.3.1 Krämer

Os padrões de projeto representam soluções comuns para problemas de projeto orientado a objetos. A localização desses padrões em um sistema de software existente pode vir a contribuir positivamente em relação às facilidades de manutenção do sistema investigado. O trabalho de Krämer (1996), intitulado "*Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software*", apresenta uma ferramenta cujo objetivo é a investigação de padrões estruturais a partir do código fonte.

A ferramenta denominada *Pat System* executa a extração de informações pertinentes de um arquivo fonte em C++ e as armazena em um repositório de dados. Os padrões são expressados como regras Prolog e as informações extraídas como fatos. Portanto, através de consultas as informações, o autor propõe a pesquisa dos padrões.

A idéia fundamental da ferramenta é representar padrões e informações em Prolog e deixar que a máquina Prolog execute as inferências sobre a base de dados. A Figura 3.5 mostra a arquitetura do sistema.

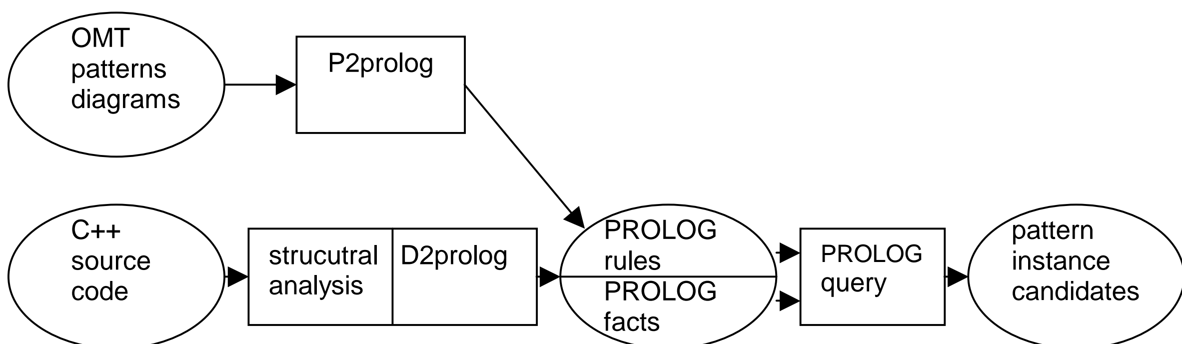

FIGURA 3.5 – Arquitetura da Ferramenta *Pat System* (KRÄMER, 1996)

Cada padrão é representado por intermédio de um diagrama estático OMT. Os diagramas constituem o repositório denominado *P*. Uma aplicação Prolog, denominada *P2prolog,* é utilizada para converter as informações do repositório em regras PROLOG, as quais representem as propriedades básicas dos padrões. O mecanismo de análise estrutural da ferramenta é utilizado para extrair as informações armazenadas em um código fonte C++ e armazená-las em um repositório denominado *D*. Uma segunda aplicação, denominada *D2prolog*, converte tais informações em fatos PROLOG. Por fim, uma *query Q* detecta as instâncias dos padrões de *P*, examinando *D.* A arquitetura conta, também, com um pós processamento que é utilizado para remover duplicações as quais, freqüentemente, ocorrem na saída PROLOG.

O trabalho apresenta como exemplo a identificação do padrão *Adapter*. Resumidamente, o padrão converte a interface de uma classe ou objeto para a interface esperada pelo cliente, proporcionando que, muitas vezes, classes incompatíveis passem

a trabalhar juntas. A seguir é apresentado um arquivo fonte em C++ e a sua representação similar em fatos Prolog.

```
Class zPane:public zChildWin {
zDisplay* curDisp;
        /* ... */
public:
        virtual void show(int=SW_SHOWNORMAL);
        /* ... */
};

class(concrete, zPane).
inheritance(zChildWin, zPane).
attribute(zPane, curDisp).
operation(virtual, selector, zPane, show, public, “int,”,”void”).
```

As regras *PROLOG* para o padrão *Adapter,* descritas a seguir, descrevem necessariamente, mas não suficientemente, as propriedades da classe. O padrão demanda a existência de delegação do método *Adapter::Request* para *Adapter::SpecificRequest*. Mas, devido ao mecanismo de análise estrutural, essas delegações não são extraídas.

```
adapter(Target,Adapter,Adaptee)
            class(_,Target)
            class(concrete, Adapter),
            class(concrete,Adaptee),
            operation(_,_,Target,Request,_,_,_)
            operation(_,_,Adapter,Request,_,_,_)
            operation(_,_,Adaptee,
                        SpecificRequest,_,_,_),
            inheritance(Targe,Adapter),
            association(Adapter, Adaptee).
```

No trabalho são apresentadas, também, algumas regras para os padrões *Bridge, Composite, Decorator e Proxy,* as quais seguem a mesma lógica do padrão *Adapter*.

Por fim, a ferramenta *Pat System* caracteriza-se como uma ferramenta fácil e simples de utilizar. Uma limitação nessa abordagem é que alguns padrões requerem muita informação semântica sobre o comportamento dos métodos e a arquitetura do sistema apresenta limitações nessas extrações. Devido à limitação considerada, a ferramenta consegue um índice de precisão de, aproximadamente, 40% de reconhecimento. Algumas construções são detectadas incorretamente devido à falta de informações suficientes.

3.3.2 Bansiya

A habilidade para entender sistemas de software a partir do código fonte é aumentada, visualizando-se os sistemas em níveis mais altos de abstração, ao invés de verificá-los como conjuntos de classes e colaborações. Neste sentido, os padrões de projeto podem representar uma ferramenta efetiva no entendimento desses sistemas, pois apresentam um grau de abstração intermediário.

Portanto, aplicações OO visualizadas como um sistema de interação entre padrões requer a descoberta, a identificação e a classificação de grupos de classes relacionadas a partir do código fonte. Essas visualizações representam qualquer padrão documentado ou agrupamentos que executam uma tarefa abstrata e não são necessariamente uma solução de padrão conhecida.

O trabalho intitulado “*Automating Design-Pattern Identification*” (BANSIYA,1998) propõe uma ferramenta que automatiza a descoberta, identificação e classificação de padrões de projeto a partir de aplicações fonte em C++.

Bansiya (1998) acredita que a solução para a identificação de um padrão está definida e representada por uma estrutura geral das classes no padrão, bem como por um assinalamento de responsabilidades entre as classes que participam da estrutura. Portanto, à estrutura associada ao conjunto de responsabilidades são personalizadas como soluções para um determinado padrão que podem ser aplicadas em situações e necessidades variadas. Em seu trabalho, afirma que o uso de padrões caracteriza-se por meio de duas alternativas: construído conscientemente em um projeto de sistema de software ou criado como resultado da solução de um problema que facilita a implementação. Isto faz com que os projetistas de sistemas estejam constantemente re-descobrindo soluções. Assim, o objetivo dos padrões é justamente reduzir estes redescobrimentos no intuito de promover o uso de padrões catalogados em repositórios de soluções.

Tipicamente, é difícil reconhecer padrões utilizados em sistemas que representam o mundo real, a menos que o projetista saiba, nitidamente, qual conjunto de especificações está observando, e então procure os indícios que representam o padrão. Isso acontece porque as formas finais da implementação do padrão são, geralmente, diferentes dos exemplos simplistas e isolados que são utilizados na bibliografia encontrada.

Estruturas limpas, como as encontradas na bibliografia de padrões, se tornam complicadas em sistemas de software do mundo real devido:

- às influências, como os domínios nos quais os padrões são implementados;
- aos fatores de modificação, como o número de classes participantes e as responsabilidades dessas que executam múltiplos papéis;
- as preferências pessoais e o estilo do projetista;
- as particularidades da linguagem de programação adotada e
- as necessidades de otimizar o desempenho.

A estratégia proposta por Bansiya (1998) é identificar o uso de padrões em aplicações fonte em C++ baseado na relação estrutural entre classes e objetos. A abordagem utiliza, também, heurísticas, informações empíricas derivadas do projeto e métricas de implementação para a identificação de padrões. Essa abordagem para a descoberta de padrões focaliza as relações estruturais fundamentais de interesse para a identificação de: herança, agregação e uso. Enquanto o mecanismo de herança é fácil de identificar, agregação e uso apresentam várias formas de implementação. Tipicamente, a relação de agregação é avaliada por meio das declarações de dados-membro, já as relações de uso são implementadas por parâmetros de métodos, em geral identificadas mediante os tipos de parâmetros.

Outra característica poderosa e efetiva de vários padrões é o polimorfismo, o qual possibilita a reescrita de classes-pai em classes-filhas e permite que uma classe-filha possa ser manipulada como uma classe-pai. Esse conceito oferece grande flexibilidade às aplicações para escolherem, em tempo de execução, os tipos de objetos que serão instanciados e manipulados. Como o real valor do polimorfismo só pode ser percebido quando a aplicação é executada, é difícil usá-lo efetivamente na descoberta de padrões pela implementação de aplicações estáticas.

Porém, métodos identificados como virtuais em C++ caracterizam informações que podem ser utilizadas como indícios para identificação de padrões. Mas isto requer uma análise gramatical, na tentativa de localizar o uso de métodos virtuais e onde esses são reescritos.

Outro componente que deve possuir tratamento especializado é a interface cuja implementação em C++ são classes abstratas que possuem uma ou mais funções membro, que são declaradas puramente virtuais. Estas funções virtuais são

implementadas pela reescrita nas classes derivadas. Resumidamente, uma interface ajuda a definir um contrato de serviços para os clientes.

Assim, para Bansiya (1998), uma lista de elementos estruturais automaticamente identificáveis de um sistema orientado a objetos deveria incluir: interfaces (classes abstratas), classes e subclasses, classes *template* (modelos), relações de herança, agregação por variáveis de instância, agregação por referência e relações de uso.

É feita uma ilustração de como pode ser desenvolvido o algoritmo de descoberta do padrão estrutural *Composite*. O padrão *Composite* é útil em soluções que requerem objetos individuais (componentes) e a composição desses (coleção) para serem tratados identicamente. A estrutura básica de um *Composite* requer uma relação de agregação entre a classe-pai e as classes-filhas. Esta relação é implementada, tipicamente, como uma referência da classe-filha composta para a classe-pai, com uma cardinalidade de 1 para N. A descoberta do padrão depende da descoberta da relação circular das classes-filhas na hierarquia, conforme Figura 3.6.

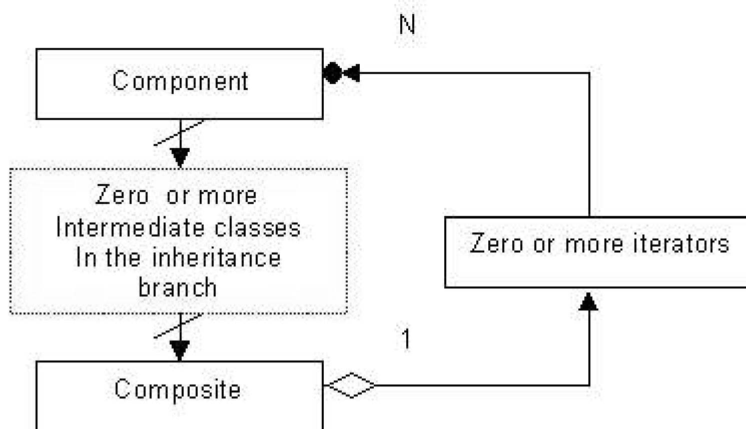

FIGURA 3.6 - Estrutura Padrão *Composite* (BANSIYA, 1998)

A ferramenta construída nesse trabalho, chamada DP++, apresenta uma arquitetura a qual consiste de três subsistemas principais, mostrados na Figura 3.7. A ferramenta faz a compilação parcial da aplicação fonte em C++ para gerar informações para o *browser* sobre o banco de dados do projeto. Os algoritmos de descoberta operam nessas estruturas de dados internas armazenadas no banco de dados. Os componentes de exibição da ferramenta permitem visualizar a estrutura da aplicação e os padrões descobertos por meio de agrupamentos de classes.

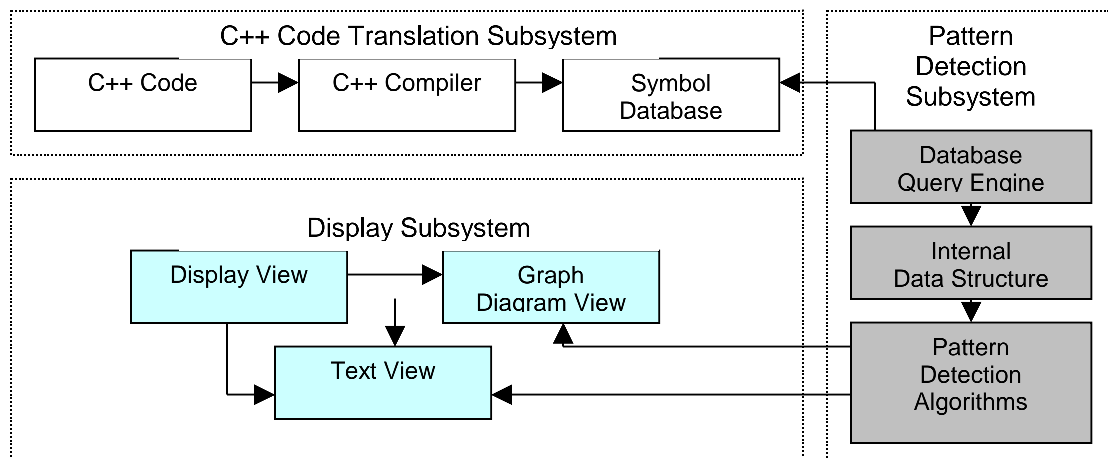

FIGURA 3.7 – Ferramenta DP++ (BANSIYA, 1998)

Segundo o trabalho, DP++ descobre atualmente a maioria dos padrões estruturais baseado na descoberta de relações estáticas. Também identifica agrupamentos de classes funcionalmente relacionadas que necessariamente podem não representar nenhum padrão conhecido, mas representam uma abstração da aplicação.

Nesse trabalho, a abordagem para identificar padrões de projeto consiste em reduzir os padrões a estruturas mínimas requeridas pelas suas soluções. Porém, uma aproximação alicerçada somente na estrutura estática não se caracteriza completa porque estas não são estruturas suficientes, pois vários padrões tendem a utilizar estruturas básicas semelhantes. Assim, reduzir as identificações errôneas é objetivo deste trabalho, mediante a construção de um repositório de informações e a criação de mecanismos para especificar limiares para a identificação de padrões.

3.3.3 Seeman

O trabalho de Seeman (1998), intitulado "*Pattern-Based Design Recovery of Java Software*", apresenta o modo de recuperar informações de projeto a partir de aplicações fonte na linguagem Java. Esse trabalho caracteriza um processo de aproximação baseada em várias camadas crescentes de abstração. O compilador coleciona informações sobre mecanismos de herança, colaborações e chamadas a métodos. O resultado desta fase é um grafo sobre o qual é aplicada uma gramática, a qual em conjunto com alguns critérios predefinidos, visa a propor o processo de identificação. Durante o processo de aproximação, a descoberta de nomes predefinidos como *Vector* ou *HashTable,* é utilizada para determinar multiplicidade entre colaborações.

A escolha pela linguagem Java se fez devido à grande migração de sistemas para esta linguagem, mas deve-se levar em conta que os sistemas de software, os quais representam o mundo real, apresentam um número considerável de pacotes, o que pode vir a prejudicar o entendimento desses sistemas.

Nesse trabalho é feita a construção de um banco de dados com o propósito de armazenar informações a respeito das descrições contidas no código, como, por exemplo:

- classe *extends* classe – herança de classe;
- interface *extends* interface - herança de interface;
- classe *implements* interface – implementação de interface;
- classe *references* classe;
- classe *references* interface – o termo *references* significa que a classe possui um atributo do tipo da segunda classe e, conseqüentemente, pode representar uma associação ou agregação;
- classe *owns* método – referências a métodos;
- método *call* método – esta referência deve ser subdividida em relações mais específicas, para seu perfeito entendimento, como:
    - *this* – o próprio objeto;
    - *super* – superclasse;
    - *attrib* – um atributo;
    - *local* – variável local de um método;
    - *param* – um parâmetro de um método;
    - *result* – o resultado de um método;
    - *static* – chamada a métodos de classe estática;
- classe *downcasts* classe – *cast* em classes;

• método *create* classe – indica que um método cria uma instância de uma classe;

Por fim, é possível construir um grafo com 3 tipos de nodos (representados pelas classes) e 8 arcos diferentes (representados pelas colaborações), conforme segue, onde $\phi(c)$ denota o nome para uma classe ou interface, e $\phi(m)$ descreve o nome, tipo de retorno e tipo de parâmetros para um método com n – 1 parâmetros:

S = (V, E, ϕ, η) como:
V = CLASS $\cup$ INTERFACE $\cup$ METHOD,
E $\subseteq$ CLASS x CLASS $\cup$ INTERFACE x INTERFACE
$\cup$ CLASS x INTERFACE $\cup$ CLASS x METHOD
$\cup$ METHOD x METHOD $\cup$ METHOD x CLASS,
ϕ : V$\rightarrow$ TEXT x TYPE$^{\eta}$
η : E $\rightarrow$ TEXT x TEXT x MULTIPLICITY

O conjunto de arcos E é subconjunto da união dos 6 produtos cartesianos descritos na lista. Todos os arcos são direcionados e podem existir vários arcos entre os mesmos nodos.

Por fim, η consiste na descrição do arco, sendo representado por três elementos: tipo do arco (*owns* ou *call*), tipo específico (*local* ou *attrib*) e a cardinalidade da relação.

Após a construção do grafo, Seeman (1998) propõe o filtro do grafo para que se possa obter uma estrutura clara a respeito do software investigado. O caminho proposto é separar o grafo de acordo com os tipos de arcos. Esse processo conduz à construção de subgrafos. Portanto, algumas regras são verificadas:

• a chamada a métodos pode ser simplificada ignorando os rótulos e considerando uma transformação para a relação *calls*, por exemplo:

ml *calls* m2 ^ cl *owns* ml ^ c2 *owns* m2 $\rightarrow$ c1 *calls* c2
(ml,m2: *methods*, cl,c2: *classes*)

Baseado no banco de dados original, um subconjunto de *calls* pode ser obtido definindo a relação *uses,* onde m1 *uses* outro método m2 se*:*

e = ml *calls* m2 $\in$ E ^ η(e)2 = *this*

• a relação *create* pode ser generalizada para a relação class-class:

$\exists$m1 : ml *creates* c2 ^ c1 owns ml $\rightarrow$ cl *creates* c2
(ml: *method*, c l, c 2: *classes*)

A criação acontece no construtor ou em outro método. A inicialização de um atributo é outra fonte para a relação *creates*:

cl *references* (init) c2 $\rightarrow$ cl *create* c2

• a interseção da relação *calls* com *references* gera a relação de associação *assoc,*não sendo consideradas associações criadas a partir de chamadas temporárias:

cl *assoc* c2 $\leftrightarrow$ cl *calls* c2 ^ cl *references* c2

Uma associação é uma agregação, se o objeto referenciado é criado pela classe, e conseqüentemente a interseção de *assoc* e *creates* é computada:

cl *aggreg* c2 $\leftrightarrow$ cl *assoc* c2 ^ cl *creates* c2

• a delegação significa a troca da responsabilidade de um método para outro objeto, que freqüentemente acontece na aplicação por contrato:

ml *delegates* m2, se
$e = m1$ *calls* $m2 \in E \wedge \eta(e)2 \neq$ *this*
$\wedge \neg\exists\, m3 \neq m2 \in E : m1$ calls $m3$
or $\forall m \in E : m1$ *calls* $m \rightarrow \phi(m1)1 = \phi(m)1$
c1 *assoc* c2 $\wedge\ \exists mi \in ci : m1$ *delegates* $m2 \rightarrow c1$ *delegates* c2

Uma delegação é assumida se somente um método é chamado ou se todos os métodos chamados possuem o mesmo nome do método de chamada, apesar de esta definição ser muito restritiva.

Após as considerações do filtro, o trabalho apresenta a verificação do padrão *Composite*. Começando por uma classe C, colecionam-se todos seus descendentes e verifica-se a caracterização de uma agregação com a raiz pela qual se delegam execuções via esse *link*. Neste caso, há altas evidências para a existência do padrão *Composite*, onde a classe C se comporta como a classe *component*. Começa-se com a relação básica e, após, derivam-se os mecanismos como agregação ou delegação e, então, recuperam-se o padrões.

SUB(C) = {D | D *extends** C }

onde extends* denota o fechamento transitivo da relação.

$\exists D \in$ SUB(C) : D *aggreg*(multiple) C $\wedge$ D *delegates* C $\rightarrow$ *composite*(C, D, A) para:
D = SUB(D) e A = SUB(C) - SUB(D)
D *inherits* C e A *inherits* C

Nesse trabalho, Seeman (1998) apresentou alguns critérios para descobrir padrões de projeto utilizando a estrutura do código fonte em aplicações Java. Esse trabalho representa uma extensão do trabalho de Krämer (1996), que deu início ao reconhecimento de informações a partir da estrutura estática. A razão desse trabalho é a proposta de colecionar mais informações como, por exemplo, chamadas a métodos. A presença de interfaces em código de Java torna isto mais fácil para definir a identificação de padrões em contraste com C++. A demonstração da abordagem, a partir de um pequeno estudo de caso foi suficiente para caracterizar um bom índice de identificação correta de padrões.

3.3.4 Freitas

Estudos de padrões estimulam facilidades durante a construção e manutenção de software orientado a objetos. O trabalho de Freitas (2000C), intitulado "*Design Patterns Automatic Identification Tool*", tem como objetivo demonstrar o protótipo de uma ferramenta desenvolvida para a identificação de padrões em aplicações *Smalltalk*. A ferramenta caracteriza-se como um protótipo de validação para um conjunto de exemplos. O desenvolvimento de exemplos permitiu a comparação e identificação de padrões, e, portanto, apresentou vantagens e tendências na utilização de um ou outro tipo de colaboração entre classes e objetos.

No trabalho foi apresentada uma especificação lógica com o objetivo de propor a construção de heurísticas para identificação de padrões de projeto em aplicações orientadas a objetos. Primeiramente, é elaborada uma aplicação utilizada como exemplo. Nesse exemplo foram identificadas as colaborações entre os objetos da aplicação e a partir dos relacionamentos foram verificadas as evidências de existência do padrão avaliado. Nesse processo, fez-se uma avaliação particular do padrão *Decorator*.

Em um segundo estudo, foi apresentado um protótipo de ferramenta que visa a propor a validação lógica das heurísticas estudadas. A ferramenta foi construída em *Smalltalk* e o autor justifica a escolha da linguagem pelo fato de que essa apresenta um conjunto de facilidades para a manipulação e avaliação de objetos e metaobjetos em tempo de execução. É importante ressaltar que a ferramenta necessita de uma série de aprimoramentos, pois ainda encontra-se em estado prematuro e, portanto, não se pode afirmar que apresente cem por cento de certeza lógica em seus resultados. Faz-se necessário, portanto, incluir um maior número de testes de aplicações reais.

A ferramenta, descrita na Figura 3.8, apresenta três blocos principais: um módulo responsável pela extração de informações, um módulo responsável pela aplicação das heurísticas e um módulo de visualização. A partir da descrição-fonte em *Smalltalk,* produzida pelo projetista, o módulo de extração de informações tem por função extrair as características dos objetos em estudo no que se refere a colaborações. Sobre estas informações serão aplicadas as heurísticas existentes. O módulo responsável por aplicar as heurísticas provê, portanto, a identificação de padrões levando em consideração características predeterminadas de cada um. O módulo de visualização permite ao projetista conferir graficamente as características do objeto e controlar a especificação.

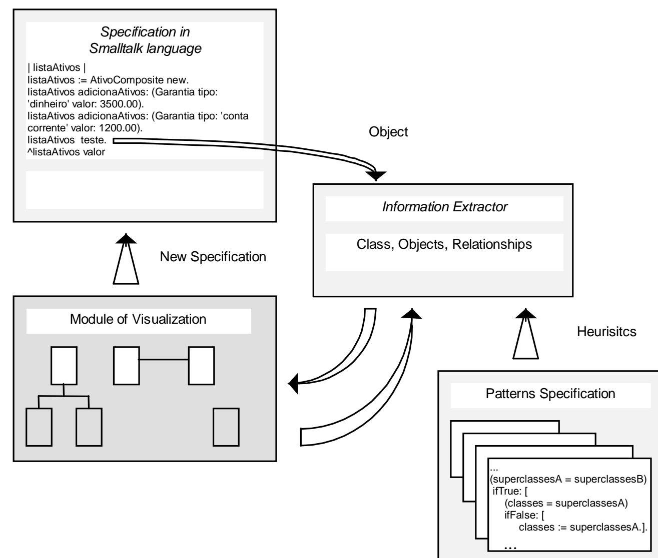

FIGURA 3.8 – Ferramenta de Detecção Automática (FREITAS, 2000C)

O autor propõe a redução de padrões conhecidos ao mínimo de estruturas necessárias e identificáveis no domínio da solução. Inicialmente, percebe-se que um estudo somente sobre as estruturas dos padrões não finaliza esse estudo, porque tais estruturas são, muitas vezes, semelhantes. Portanto, como vários padrões tendem a usar estruturas básicas aproximadas, para que se possam reduzir identificações errôneas, faz-se necessário estender essa pesquisa, procurando por heurísticas de projeto que utilizam dados empíricos para solucionar a presença de soluções padrões. As heurísticas e dados empíricos, segundo o autor, serão extraídos do projeto e métricas de implementação devem ser avaliadas sobre a estrutura e características funcionais de classes e relacionamentos.

3.3.5 Guéhéneuc

Os padrões de projeto descrevem microarquiteturas que resolvem problemas arquitetônicos de projeto periódicos dentro da OO. É importante identificar essas microarquiteturas durante a manutenção de aplicações orientadas a objetos. Porém, essas microarquiteturas aparecem freqüentemente distorcidas no código. O trabalho de Guéhéneuc (2001), intitulado "*Using Explanations for Design Patterns Identification*", apresenta uma abordagem baseada na restrição de implementação dos padrões de projeto.

A produção de aplicações é uma questão importante para a indústria de software. Um código fonte de qualidade facilita o processo de manutenção considerando a adição de novas funcionalidades, correções de *bugs*, adaptação a novas plataformas e integração com novas bibliotecas.

A programação orientada a objetos de boa qualidade deve levar em conta os seguintes aspectos:

- algoritmos eficientes e escritos de forma clara, levando em consideração as convenções da linguagem;
- uma arquitetura de classes elegante e bem estruturada.

Levando em consideração a escrita de código a partir de microarquiteturas como os padrões de projeto, é importante salientar que não é fácil escrever código fonte o qual respeite cuidadosamente os padrões.

Um padrão de projeto é descrito como um conjunto de regras lógicas. As regras lógicas unificadas com os fatos de representação formam o código fonte e identificam o padrão. Mas essa descrição é limitada, pois é difícil descobrir diretamente as microarquiteturas que representam os padrões, pois na grande maioria dos casos, a implementação é distorcida.

Nesse trabalho, Guéhéneuc (2001) propõe o uso de restrições para identificar a implementação de padrões. Para facilitar o entendimento, o autor utiliza a linguagem Java. A seguir, considera-se o problema da satisfação de restrições. As decisões são elaboradas a partir da fase de enumeração, que corresponde a somar ou remover restrições do sistema. A seguir, considera-se o problema da satisfação de restrição (*V*, *D*, *C*).

Uma restrição (C'), na verdade, representa um subconjunto do conjunto original de restrições.

$$(C' \subset C)$$
$$C \vdash \neg(C' \wedge v_1 = a_1 \wedge \ldots \wedge v_k = a_k) \quad (1)$$

Em uma explicação de restrição composta pela última decisão de restrição, uma variável é selecionada e a fórmula é reescrita:

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (vi = ai) \rightarrow vj \neq aj$$

Por outro lado, quando uma restrição é selecionada, faz-se a eliminação de uma explicação e, portanto, um valor também é eliminado (aj é eliminado do domínio de vj).

$$\text{expl}(vj \neq aj)$$

Classicamente, o trabalho utiliza técnicas de domínio-redução (remoção de valores), registrando explicações sobre os valores eliminados. Cada eliminação de valor corresponde a uma restrição de reconhecimento. Portanto, o padrão é identificado quando o domínio da variável vj é vazio. Uma explicação de restrição pode ser

facilmente computada quando feita a eliminação da explicação associada, de acordo com cada valor removido:

$$C \mid\!- \neg\left(\bigwedge_{a \in d(vj)} expl(vj \neq a)\right)$$

Um mínimo de explicações são interessantes e devem fornecer informações precisas sobre dependências entre variáveis e restrições. Infelizmente o processo de computação das explicações é demorado. Existe um compromisso entre o tamanho das informações e a computabilidade.

Nesse trabalho, o autor considera o desenvolvimento de uma aplicação para produzir representações de documentos. A superclasse da aplicação consiste em uma classe do tipo *Element,* que define uma interface comum para todos os elementos de um documento: títulos - classe *Title*, parágrafo - classe *Paragraph* e parágrafos identados - classe *ParaIndent*. Um documento - classe *Document* - compõe-se desses elementos. Esta arquitetura é semelhante ao padrão *Composite*, mas as especificações do padrão requerem que a classe *Document* se caracterize como uma subclasse de *Element* para unificar as interfaces e permitir a composição de documentos.

O padrão *Composite* é modelado associando uma variável a cada classe (*Component*, *Composite* e *Leaf*) e, por restrição, os valores dessas variáveis estão de acordo com os relacionamentos das classes: *Component* < (superclasse) *Composite*, *Component* < *Leaf* e *Composite* ⊃ (composição) *Component.*

O código fonte da aplicação demonstrada envolve os seguintes elementos:

- classes: *AbstractDocument*, *Element*, *Title*, *Paragraph*, *ParaIndent*, *Document* e *Main*;
- domínio para cada variável: *Component*, *Composite* e *Leaf* de tamanho 7 (considerando uma entrada para cada classe);
- modelo genérico:

```
PClass
        name: string,
        superclasses: list[ PClass],
        components: list[ PClass],
        componentsType: PClass,
        relatesTo: list[ PClass],
        doNotRelateTo: list[ PClass]
```

Os relacionamentos entre classes são codificados no modelo genérico, o qual é utilizado para verificar os requisitos do padrão investigado:

- *name*: representa o nome de cada classe;
- *superclasses*: representa a lista de superclasses;
- *components*: identifica a lista de todos os componentes agregados pela classe identificada;
- *componentsType:* identifica as superclasses de todos os componentes;
- *relatesTo*: representa a lista de todas as classes que possuem relacionamento com a classe identificada;
- *doNotRelateTo*: representa a lista de todas as classes que não possuem relacionamentos com a classe identificada.

A resolução da modelagem produz a representação da seguinte forma:

- < dist.sol.# > .*<Quality>.component* = <a *class*>
- < dist.sol.# > .*<Quality>.composite* = <a *class*>
- < dist.sol.# > .*<Quality>.leaf* = <a *class*>

A solução para restrição *Component* < *Composite* com peso 50, foi:

- 1.50.*component* = *Element*

- 1.50.*composite* = *Document*
- 1.50.*leaf* = *Paragraph*

Por fim, Guéhéneuc (2001) apresenta a abordagem baseada em restrições na qual é feita a identificação de padrões de projeto a partir de código fonte. São utilizadas explicações para prover ao usuário a identificação de padrões distorcidos, levando em consideração um conjunto de restrições disponíveis armazenadas em uma biblioteca. O trabalho verifica padrões e possíveis distorções de implementação a partir do código fonte, utilizando um conjunto de restrições predeterminadas.

Conclusões

Observa-se que a preocupação com os processos de desenvolvimento, documentação e manutenção de sistemas de software orientados a objetos é uma realidade. Diversos autores enfocaram estes problemas definindo especificações, formalismos e, até mesmo, ferramentas para auxiliar tais processos. O objetivo deste capítulo foi, de forma abrangente, apontar alguns trabalhos na área, bem como demonstrar a importância destes no contexto da OO.

Na seção 3.1, Automatização no Desenvolvimento de Sistemas de Software OO, verificou-se a motivação dos autores em melhorar os índices de qualidade durante o processo de desenvolvimento de um sistema. Agarwal (1995), Jiason (1996) e Freitas (1998) são unânimes em fornecer subsídios que auxiliem o projetista nesta fase, sem restrições. Todas as propostas enfocam possíveis problemas e/ou erros decorrentes durante o desenvolvimento, que se refletem na documentação e implementação do sistema. Os autores não apontam respostas definitivas para os problemas, mas sim sugestões para novos experimentos.

Uma idéia, aceita pela comunidade para apoiar o desenvolvimento, é a utilização de padrões de projeto. A utilização de padrões é, muitas vezes, feita de forma inconsciente, e não por meio do projeto do sistema. Isto pode gerar possíveis distorções na documentação e implementação do sistema. Obviamente, o processo de manutenção irá tornar-se mais complexo.

No intuito de aprimorar a especificação de padrões, a seção 3.2, Geração e/ou Formalização de Padrões de Projeto, apresenta a descrição dos trabalhos de Budinsky (1996) e Wild (1996), cujo objetivo é a geração automática de código para aplicações. Por meio de descrição textual ou gráfica, as ferramentas propostas levam à geração de padrões em código fonte. As ferramentas não representam um modo revolucionário para produção de código, mas sim um passo para facilitar o processo de implementação. Mas a geração não resolve o problema de entendimento dos sistemas de software, pois a falta de formalismos prejudica a expressividade das especificações de padrões. Neste sentido, os processos seguintes podem sofrer distorções. No sentido de melhorar tal necessidade, Eden (1997) e Lauder (1998) criam especificações não-ambíguas de padrões puros, as quais visam a permitir a verificação de projetos inconsistentes e incompletos. Mas essa realidade é problemática, pois o trabalho de Vlissides (1998) aponta possíveis composições entre padrões, o que pode vir a comprometer o entendimento geral do projeto. Já a proposta de Freitas (2000A) propõe a criação de um *framework* para utilização de padrões predeterminados, na tentativa de facilitar o entendimento de tais estruturas. Todos os trabalhos propostos visam a melhorar e auxiliar o entendimento da utilização de padrões, mas é importante caracterizar a influência de hábitos e experiências pessoais do projetista no desenvolvimento do sistema. Essas experiências podem ser maléficas, considerando-se possíveis distorções inseridas no contexto.

Avaliando sob este enfoque, a motivação dos autores é investigar a utilização dos padrões a partir do sistema proposto. Os autores citados na seção 3.3, Verificação e Identificação Automáticas de Padrões de Projeto, Krämer (1996), Bansiya (1998), Seeman (1998), Freitas (2000C) e Guéhéneuc (2001), demonstram a preocupação com o processo de manutenção de software devido a possíveis irregularidades inseridas durante o desenvolvimento e documentação. Todos os autores nessa seção têm como objetivos inspecionar padrões por meio da identificação de estruturas e colaborações do sistema. Os autores justificam que a fase de manutenção é uma fase de risco, pois na grande maioria das vezes os projetistas de sistemas precisam examinar o código fonte na busca de detalhes do sistema que não foram devidamente documentados durante o desenvolvimento do sistema de software. Os trabalhos de Krämer (1996), Bansiya (1998) e Seeman (1998) possuem o foco sobre o modelo estático da aplicação, ou seja, sobre um código fonte é realizada uma inspeção das entidades para determinar possíveis identificações. Os autores enfatizam a necessidade de adaptações considerando que o modelo não fornece informações semânticas necessárias à identificação de vários padrões. O experimento de Freitas (2000A) demonstra o processo de investigação a partir de aplicações *Smalltalk,* o que caracteriza a investigação da aplicação em tempo de execução. O modelo apresenta algumas limitações, pois vários padrões tendem a usar estruturas básicas aproximadas e, como *Smalltalk* não é tipada, estas limitações são reforçadas. Por fim, o modelo proposto por Guéhéneuc (2001) utiliza uma técnica baseada em restrições, o que limita as identificações errôneas. Mas o trabalho caracteriza-se, também, pela investigação mediante código fonte, o que acarreta uma possível falta de subsídios para investigação.

Neste capítulo, portanto, foi apresentada uma série de trabalhos que forneceram subsídios para o desenvolvimento desta tese. Acredita-se que ferramentas com a capacidade de extração de informações e interpretação de padrões, com a possibilidade de modificar situações quando necessário e diagnosticar condições para reuso de informações, representam um avanço substancial no estado da arte sobre ferramentas de apoio ao projeto, manutenção e documentação.

Apesar de um grande número de trabalhos definirem a importância da utilização dos padrões de projeto, principalmente no que tange a facilidades de reuso, o trabalho de Prechelt (2001), intitulado "*Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions*", apresenta uma visão sistêmica de tais conceitos. A proposta tem como base a comparação de cenários que apresentam soluções com padrões e cenários com soluções simples.

O autor justifica que existem duas forças: de um lado a aplicação de padrões, a qual caracteriza uma boa idéia porque as vantagens da utilização produzem boas soluções e práticas; e de outro a aplicação de padrões torna-se mais complicada do que necessária, tornando o entendimento e a manutenção do sistema complicados. A experiência de Prechelt (2001) analisa aplicações, escritas em C++, utilizando padrões versus aplicações utilizando contextos simples.

São avaliadas quatro aplicações com a implementação de padrões de projeto diferentes. O experimento analisa o desempenho entre as aplicações com ou sem os padrões. Para a realização dos testes, foram utilizados alguns padrões como: *Abstract Factory, Composite, Decorator, Facade, Observer e Visitor,* os quais o autor julga serem os mais utilizados em sistemas de software. A avaliação do desempenho dos testes é feita utilizando os seguintes quesitos:

- entendimento da tarefa de manutenção;
- procura por aspectos e partes da aplicação que sejam relevantes à realização da tarefa;
- entendimento dos aspectos relevantes da tarefa;

- entendimento de como é possível realizar modificações e
- execução de uma modificação requisitada.

Por fim, entre algumas expectativas formuladas, o trabalho apresenta, por exemplo:

- *Observer* e *Visitor* – a solução por padrões é mais complicada, a não ser que seja requerida flexibilidade, bem como são padrões difíceis de entender;
- *Decorator* – a solução do padrão é fácil de utilizar, mas pode apresentar erros de invocação a métodos;
- *Composite* e *Abstract Factory* – as duas versões são estruturalmente similares, com desempenhos diferentes.

É importante salientar que o trabalho apresenta algumas expectativas apenas qualitativas, e não resultados quantitativos, o que torna necessário, segundo o autor, prover o modelo de informações adicionais para confirmar ou rejeitar as hipóteses formuladas. Mesmo assim, a proposta fornece subsídios para justificar o quanto é importante uma ferramenta de apoio ao processo de manutenção, que possa identificar padrões os quais em muitos casos, possuem um difícil entendimento em aplicações.

Neste sentido, acredita-se que a implementação de uma ferramenta para executar aplicações Java com estas características venha a influenciar positivamente os índices de qualidade durante o processo de manutenção de software. O objetivo desta tese é propor esta ferramenta, cujas características explorem os modelos estático e dinâmico de uma aplicação.

No próximo capítulo é feita uma descrição detalhada de alguns padrões investigados pela ferramenta. Os padrões são descritos conforme a literatura, bem como são demonstradas as suas possíveis formas de representação.

4 Identificação de Padrões de Projeto

Os padrões de projeto são estruturas de colaboração entre classes freqüentemente identificadas no projeto de *frameworks* orientados a objetos. Utilizando textos e diagramas, um padrão descreve como um problema específico, que acontece repetidamente num projeto maior, pode ser resolvido (JOHNSON, 1998).

O trabalho de Gamma (1994) apresenta um catálogo de padrões cujo objetivo é relacionar os problemas de projeto mais comumente encontrados na construção de *frameworks* e o modo como estes problemas podem ser resolvidos.

Um critério de classificação de padrões é o escopo de atuação do padrão o qual especifica se a estrutura de colaboração é definida estaticamente por classes (escopo de classes) ou dinamicamente por objetos (escopo de objetos). O escopo de classes preocupa-se com a forma como as responsabilidades entre classes e subclasses estão distribuídas hierarquicamente para a realização de uma determinada funcionalidade. Em geral, o escopo de classes baseia-se em classes abstratas que definem algum protocolo, que é completado ou implementado por classes concretas. O escopo de objetos preocupa-se com as formas de combinação ou composição de objetos. Os padrões de objetos são mais flexíveis que os padrões de classe, pois composições de objetos são definidas e modificadas dinamicamente, enquanto que em relacionamentos de herança isto pode ser realizado apenas estaticamente.

Outro critério de classificação é o propósito do padrão, o qual especifica o tipo de problema do *framework* ao qual o padrão se aplica. Os padrões criacionais tratam do processo de criação dos objetos. Os padrões comportamentais caracterizam a forma pela que classes e objetos interagem e distribuem responsabilidades do sistema. Por fim, os padrões estruturais estabelecem a forma pela qual classes ou objetos são compostos e usados quando a solução do problema envolve um projeto complexo.

4.1 Detecção de Padrões de Projeto

Ao considerar como projetar aplicações para utilizar padrões de projeto, é importante lembrar que os padrões representam uma forma literária. Os padrões não são definições matemáticas rigorosas e, portanto, sofrem distorções impostas por restrições do domínio da aplicação quando considerada a sua implementação. Pensando nesta proposição, seria ingênuo refletir que uma aplicação possa compreender a intenção dos padrões de projeto.

Porém, o que pode ser detectado são as evidências de implementação que representam a solução do padrão. Estas evidências podem apontar para o padrão e apresentar argumentos convincentes de que foi utilizado. O conjunto de evidências que determinam um padrão irá ser referenciado nesta tese como a essência do padrão.

Entender como projetar os padrões é uma tarefa importante para que se possam examinar os diferentes modos nos quais é possível representá-los; portanto, é necessário entender: a hierarquia de classes, as colaborações, o protocolo das classes, bem como o fluxo de mensagens que representam particularidades e individualizam os padrões.

Muitos padrões utilizam a definição do mecanismo de herança entre as classes. Para alguns casos, deve-se definir uma classe abstrata da qual dois tipos especializados de subclasses herdam informações. Para outros, o mecanismo de herança é utilizado como uma conveniência de implementação e, necessariamente, não inclui parte da estrutura básica.

Além da herança, um projeto OO pode conter informação sobre as colaborações entre classes e/ou objetos. Em geral, as colaborações podem ser identificadas por variáveis (ou conjunto de variáveis) por meio das quais um objeto pode fazer referência a outros objetos. Em geral, considera-se esse tipo de colaboração como agregação e uma relação mais fraca, como, por exemplo uma associação pode ser determinada a partir da passagem de um parâmetro, pela instanciação ou por algum outro meio indireto.

É importante salientar que poucos padrões de projeto especificam relações de associação, pois, em grande parte, caracterizam relações específicas, o que torna o mecanismo de agregação útil nas soluções. Uma relação de agregação pode ser representada por um grafo dirigido com a adição de duas porções de informação: o dono da agregação e a cardinalidade da relação.

Muitos padrões incluem informações de como o protocolo ou conjunto de classes é estruturado. Por exemplo, alguns padrões descrevem a relação entre métodos em classes abstratas ou concretas, já outros descrevem padrões de mensagens entre objetos relacionados.

4.1.1 Tornando um Padrão Detectável

Para descobrir padrões de projeto é importante investigar o que torna possível o seu reconhecimento. Em muitos casos, os padrões são difíceis de ser descobertos em uma investigação automática.

O padrão *Interpreter*, por exemplo, permite a definição de uma gramática para a criação de uma linguagem simples e, portanto, a representação, interpretação e verificação de expressões nessa linguagem (ALPERT, 1998). Este padrão, como alguns outros, apresenta grande dificuldade para investigação automática, porque a construção de um *parser* para uma determinada linguagem implica formas de interpretação e implementação diferenciadas.

O padrão *Template Method* define o esqueleto de um algoritmo na superclasse, propiciando a definição de alguns passos pelas subclasses. Esse padrão permite que as subclasses possam redefinir certos passos de um algoritmo sem mudar a estrutura do algoritmo (GRAND, 1998). Para este caso, *Template Method* recorre a uma relação específica entre métodos virtuais e métodos concretos dentro de uma hierarquia de classes. Um método pode ser classificado como: *template method*, *hook* ou concreto examinando-se a sua definição e classificação, considerando-se o modelo estrutural sem levar em consideração as informações semânticas.

Porém, a maioria dos padrões de projeto deve manter-se entre estes dois pólos. Alguns artefatos de implementação de padrões podem ser detectados, mas a determinação do padrão pode não atingir níveis de confiança devido ao fato de esses artefatos não refletirem o uso particular de um determinado padrão.

Em geral, um padrão pode ser detectado se a sua solução for distinta e não ambígua. Se a sua solução for distinta, o padrão pode ser representado por um diagrama sem igual, o qual não é provável de ser gerado em um projeto que não o utilize. Se uma solução for não-ambígua, então esta deve ser representada de maneira única. Representações múltiplas da solução em um diagrama de projeto não devem ser possíveis.

O padrão *Adapter* converte a interface de uma classe ou objeto para a interface esperada pelo cliente, proporcionando que, muitas vezes, classes incompatíveis passem a trabalhar juntas (GAMMA, 1994). O problema básico a ser resolvido por meio do padrão é aquele cuja classe precisa utilizar outra classe, mas as interfaces delas não são semelhantes.

A solução apresenta a classe *Adapter,* como subclasse da classe *Target,* a qual contém uma instância denominada *Adaptee,* para a qual serão enviadas as mensagens. Esta solução não é ambígua, pois não existem muitas maneiras para resolver o problema. Porém, não é distinta porque a estrutura do diagrama não é única para ser identificada positivamente como um exemplo de *Adapter*. Se fosse considerado dessa maneira, então toda classe que herda informações de outra classe e contém atributos de instância poderia ser interpretada como sendo um adaptador.

4.2 Escolha de Padrões para Identificação

Escolher um conjunto de padrões de projeto que venham a servir como estudo de caso para uma ferramenta de reconhecimento automático é uma tarefa na qual devem ser observadas as características estáticas, dinâmicas e funcionais dos padrões. Sem dúvida, os padrões publicados no trabalho de Gamma (1994) são os mais bem elaborados e os mais utilizados por projetistas de sistemas de software OO.

A partir do catálogo inicial, Gamma descreve um subconjunto particular de padrões básicos, com os quais os projetistas deveriam *estar* familiarizados, quais sejam: *Composite, Decorator e Strategy.* Outro padrão importante é *Chain of Responsability,* porque está intimamente amarrado a *Composite* e *Decorator*. *Template Method* também é enfatizado pela sua proximidade com a OO e, ainda, são citados *State* e *Command,* por causa de suas familiaridades com *Strategy*. Estes padrões são considerados estruturas básicas em muitos *frameworks*. Por fim, ressalta-se a importância do padrão *Observer* devido ao controle de trocas de estados entre objetos.

A seguir faz-se um estudo de alguns padrões que serão utilizados na proposta de detecção de padrões em aplicações Java. Os padrões aqui utilizados foram avaliados sob sua ótica original conforme proposta de Gamma (1994), estudados por Johnson (1998), adaptados para utilização em *Smalltalk* por Alpert (1998) e implementados em Java por Grand (1998). Este estudo aborda uma coletânea teórica de características, bem como uma proposta de detecção destes.

4.2.1 Padrão *Composite*

O padrão *Composite* compõe objetos em estruturas de árvores para representar hierarquias todo-parte. O padrão propõe aos clientes tratarem objetos individuais e composições de objetos uniformemente. *Composite,* provavelmente, é um dos padrões de projeto mais úteis do catálogo, pois é utilizado em muitos *frameworks* e em uma variedade muito grande de domínios de problemas.

A idéia original proposta na obra de Gamma (1994), utilizando a metodologia OMT, conforme a Figura 4.1, apresenta três classes básicas: *Component, Leaf e Composite. Component* define a interface para os objetos da composição. Ela implementa o comportamento *default* para a interface comum entre esses objetos e, também, declara a interface para acessar e gerenciar seus componentes filhos. A subclasse *Leaf* define o comportamento para os objetos primitivos da composição. A classe *Composite* define o comportamento para objetos compostos, implementando as operações relacionadas de *Component*: *Add(Component)*, *Remove(Component)* e *GetChild(int).*

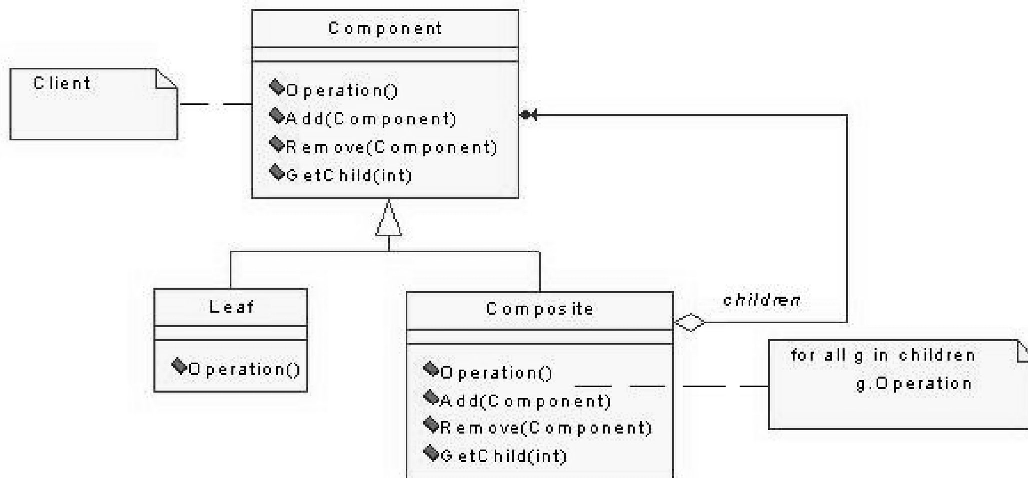

FIGURA 4.1 - Estrutura Abstrata do *Composite* para a Linguagem C++ (GAMMA, 1994)

Para manipular os objetos da composição, o cliente usa a interface da classe *Component*. Se o objeto acessado pelo cliente é um objeto *Leaf*, então o serviço solicitado é manipulado diretamente. Caso contrário, se o objeto for um *Composite*, ele repassa a solicitação de serviço para seus componentes.

No exemplo proposto por Gamma, utilizando C++ em seu trabalho, a solicitação dos serviços se dá pela execução do método *Operation()*, que por sua vez, invoca serviços por meio de uma repetição (*for*) tanto para objetos *Leaf* como para objetos *Composite*. A execução de *Operation()*, para objetos *Leaf,* ocasiona o retorno das solicitações em objetos primitivos. Já a execução para objetos *Composite* ocasiona uma nova chamada à repetição para a execução de *Operation()* em outros nodos da árvore.

A proposta de Alpert (1998) para a aplicação de padrões em *Smalltalk* é muito semelhante ao modelo definido por Gamma. Alpert utiliza o mesmo diagrama de Gamma, com a definição e representação dos métodos na linguagem *Smalltalk*. A diferença está na utilização do método *do:* pelo tradicional *for*.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, conforme Figura 4.2, o autor modifica um pouco a diagramação do padrão. Utilizando notação UML, este substitui a classe *Leaf* por *ConcreteComponent* e cria as classes *ConcreteComposite* como subclasses de *AbstractComposite*. A utilização do padrão ocorre da mesma forma.

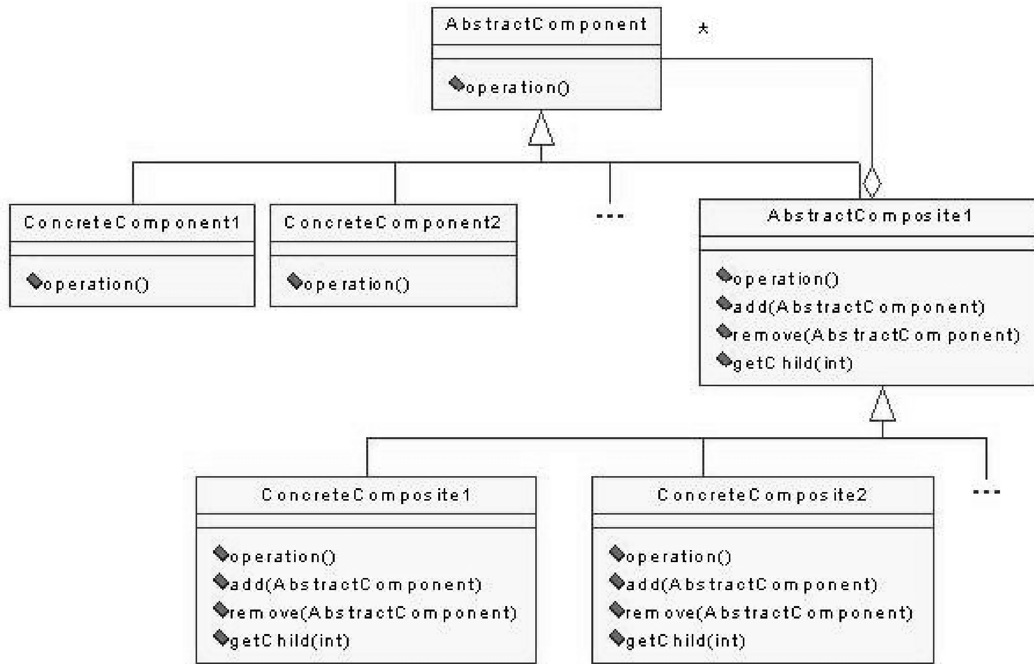

FIGURA 4.2 - Estrutura Abstrata do *Composite* para a Linguagem Java (GRAND, 1998)

A grande vantagem do padrão *Composite* é que, na estrutura hierárquica, um objeto *Composite* pode conter outros objetos *Composite* e, assim, sucessivamente. Esta forma de estrutura recursiva, apresentada na Figura 4.3, em formato de árvore, pode apresentar vários níveis, tantos quantos necessários às necessidades de projeto.

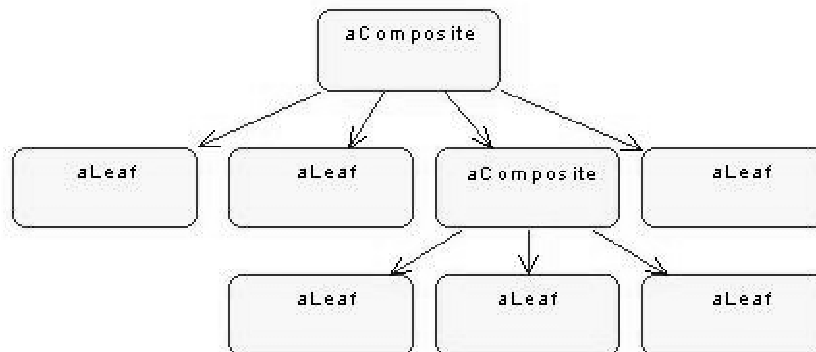

FIGURA 4.3 – Diagrama de Objetos – Padrão *Composite* (ALPERT, 1998)

Estruturas em formato de árvore representam as mais comuns e poderosas estruturas de dados em linguagens de programação. Qualquer hierarquia que represente um relacionamento de composição pode ser modelada através de uma árvore. O padrão *Composite* é um bom exemplo para tal.

Dentre as estratégias básicas para utilização e implementação do padrão *Composite,* verificam-se as seguintes:

- Utilizar uma superclasse *Component*: a classe *Composite* deve ser implementada na mesma hierarquia que a classe *Leaf,* como subclasse de *Component*. A classe *Component* deve ser única para que ocorra a utilização de uma interface comum.
- Dividir a classe *Component* em duas subclasses distintas: classes que são formadas de componentes (*Composite*) e classes que não podem ser decompostas, ou seja, primitivas (*Leaf*).

- Na hierarquia *Composite,* é possível que as classes manipulem componentes diretamente, mas é interessante prever que a superclasse *Composite* execute essa tarefa. Isto permitirá uma melhor reutilização e flexibilidade na hierarquia. A classe *Composite* delega atividades para os componentes e estende seus protocolos para adicionar comportamento.
- Objetos *Composite* devem ser aninhados. Não apenas pode um objeto *Composite* conter objetos *Leaf,* como também pode conter outros objetos *Composite*. Freqüentemente, projetistas esquecem de inserir que um objeto *Composite* deve assumir um filho como *Leaf*. Por tal descuido durante uma pesquisa, o resultado pode acusar uma falha.
- Objetos *Composite* devem possuir um código de validação para o método *Add(Component),* no intuito de manter os mesmos tipos de filhos para a estrutura *Composite*.

Resumidamente, o padrão *Composite* é a plicado quando existe um objeto complexo o qual exige a sua decomposição em uma hierarquia de objetos todo-partes. Minimiza-se a complexidade da hierarquia, pois diminui-se o número de diferentes objetos pela utilização de objetos *Leaf* mediante uma estrutura em forma de árvore.

4.2.1.1 Essência do padrão *Composite*

Todos os autores apresentam o padrão como contendo um relacionamento de agregação (composição) entre as classes *Composite* (ou *AbstractComposite*) e *Component* (*AbstractComponent*). Os exemplos propostos caracterizam um atributo na classe *Composite* em forma de coleção. Esta coleção irá armazenar, respectivamente, referências aos objetos que pertencem à composição.

Faz-se necessária, na classe *Composite*, a criação de métodos para inserção e remoção dos objetos na coleção. Deve-se atentar para que, durante o processo de inserção dos objetos, se caracterize um controle pelo qual estes pertençam à superclasse *Component*, representados respectivamente por objetos da classe *Leaf* (ou *ConcretComponent*) e da própria classe *Composite*.

Outro ponto importante que caracteriza o padrão é que os métodos abstratos na classe *Component*, como *operation,* por exemplo, devem apresentar, nos objetos da classe *Leaf,* o retorno do processamento de alguma informação e, nos objetos da classe *Composite,* devem apresentar uma repetição para o envio da mensagem *operation* a todos os objetos existentes na coleção. Isto fará com que os objetos *Composite* enviem mensagens a seus objetos *Leaf* para cada nível da árvore que for acessado, fazendo, portanto, com que se possam atingir todos os objetos envolvidos na estrutura.

A solução do padrão pode ser descoberta procurando ciclos em um grafo de herança/composição combinado, conforme figura a seguir. Sempre que uma classe tem uma relação de agregação com uma classe que descende da hierarquia de herança, existe uma forte evidência de que o padrão *Composite* pode estar sendo utilizado, levando em consideração um indicador de multiplicidade de 1 para N. A simplicidade deste padrão sugere um bom candidato para a descoberta automática.

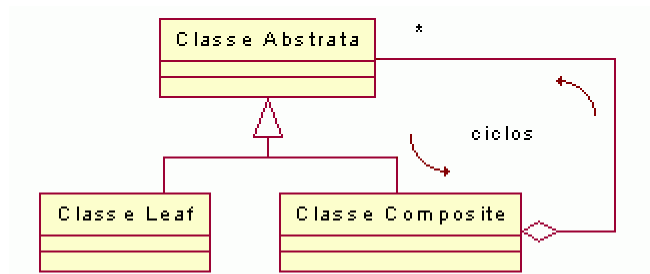

FIGURA 4.4 – Ciclos em *Composite*

4.2.1.2 Identificação do padrão *Composite*

Iniciando pela definição das classes, identifica-se a existência de duas subclasses a partir da classe abstrata A. A classe C, que demonstra a estrutura de objetos *Composite,* e a classe L, que caracteriza a estrutura dos objetos *Leaf*.

A classe C mostra, também, uma referência a objetos da superclasse abstrata. Esta referência é estabelecida via um atributo em formato de lista, identificado por *LISTATRIBS*. Cada atributo da lista, denominado *At* (*ATRIB*), representa um objeto das classes C ou L.

O mecanismo de agregação e delegação múltiplas é definido quando existe uma associação e uma referência entre o objeto em avaliação (OC – objeto *Composite*) e os objetos da lista. Cada associação deve traduzir uma relação com objetos de uma classe semelhante (outros objetos *Composite*) ou com objetos que representem subclasses (objetos *Leaf*), todos da mesma classe raiz avaliada. Para os objetos assinalados como compostos, repete-se o procedimento, recursivamente, na verificação da existência de uma hierarquia em árvore. Estas evidências, portanto, caracterizam fortes indícios para a existência do padrão *Composite*.

Adotando à notação utilizada no trabalho de Seeman (1998), pode-se deduzir o seguinte formalismo:

CLASS(C) = {C | C *extends* A ^ C *references* A}
CLASS(L) = {L | L *extends* A}
LISTATRIBS(LAs) = {LAs | LAs *attrib* C ^ LAs = LIST}

Label_Composite(OC)
ATRIB(At) = {At = LAs[n] | ∃At ∈ C v ∃At ∈ L}
∃OC ∈ C: ∀At: OC *aggreg*(*multiple*) At ^ OC *delegates* At
∀OC ∈ C: OC *agreg* At ⇔ OC *assoc* At(C) ^ OC *references* At(C) v
OC *assoc* At(L) ^ OC *references* At(L)
∀OC ∈ C: OC *delegates* At ⇔ $\forall m_1 \in$ OC: m_1 *calls* m_2 ^
OC *owns* m_1 ^ At *owns* m_2
$\Rightarrow \phi(m(OC))_1 = \phi(m(At))_1$
∀At ∈ C: Label_Composite(At)

A contribuição apresentada neste formalismo é demonstrar a especificação a qual possibilita verificar a essência do padrão a partir da estrutura dinâmica da aplicação, e não apenas leva em consideração a estrutura estática como nos trabalhos de Kramer(1996), Seeman(1998) e Bansya(1998). Portanto, além da identificação do padrão, pode-se verificar também se este está bem utilizado, levando-se em consideração as informações de entrada e saída da aplicação.

4.2.2 Padrão *Decorator*

O padrão *Decorator* acrescenta dinamicamente responsabilidades a objetos através de composição. Vários *Decorators* podem ser compostos recursivamente para acrescentar múltiplas propriedades a um objeto, evitando a criação de subclasses específicas. Basicamente, um *Decorator* realiza alguma função específica e transfere o controle ao seu componente invocando a mesma operação.

Na proposta de Gamma (1994), conforme a Figura 4.5, utilizando a metodologia OMT, a classe *Component* define a interface para objetos que podem ter responsabilidades adicionadas dinamicamente. *ConcreteComponent* representa os componentes concretos que podem ser acrescidos de comportamento. A classe *Decorator* mantém uma referência para um objeto *Component* e define uma interface padrão. Os *ConcreteDecorators*, por sua vez, implementam especificamente as responsabilidades acrescentadas para um componente.

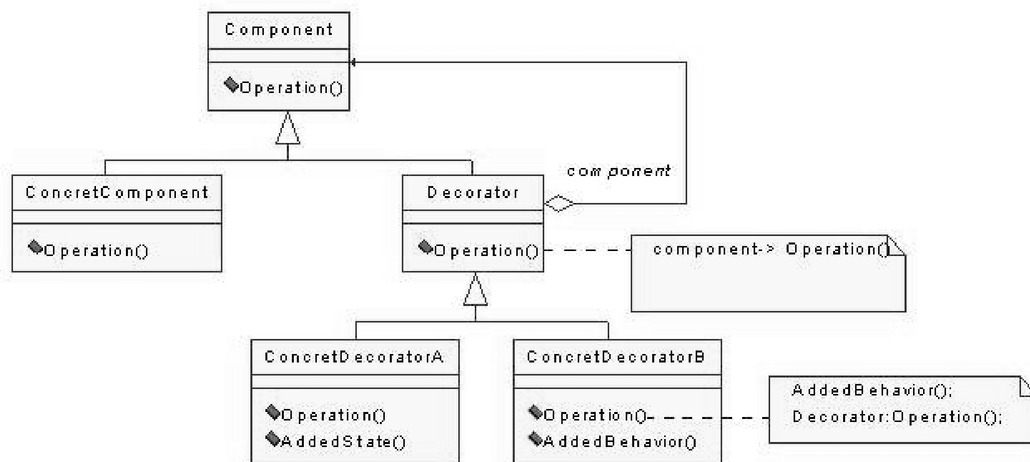

FIGURA 4.5 - Estrutura Abstrata do *Decorator* para a Linguagem C++ (GAMMA, 1994)

Decorator recebe as solicitações de serviços para seu componente referenciado (*operation*) e realiza as operações adicionais de acordo com o comportamento que está acrescentando. Após isto, repassa as solicitações para seu objeto componente, o qual pode ser outro *Decorator*.

A proposta de Alpert (1998), para a aplicação de padrões em *Smalltalk,* caracteriza-se de forma semelhante ao modelo definido por Gamma. Alpert utiliza o mesmo diagrama de Gamma com a definição e representação dos métodos na linguagem *Smalltalk*. A diferença está na nomenclatura dos métodos adaptada ao padrão *Smalltalk*.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, o autor modifica um pouco a diagramação do padrão original. Utilizando notação UML, ele substituiu a classe *Component* por *AbstractService* e criou as classes *ConcreteService e AbstractWrapper* (*Decorator*) como subclasses de *AbstractService*. Uma outra alteração proposta por Grand caracteriza-se pela utilização de um relacionamento de associação de 1 para 1, ao contrário de uma agregação utilizada nas propostas de Gamma e Alpert. A Figura 4.6 apresenta a utilização do padrão.

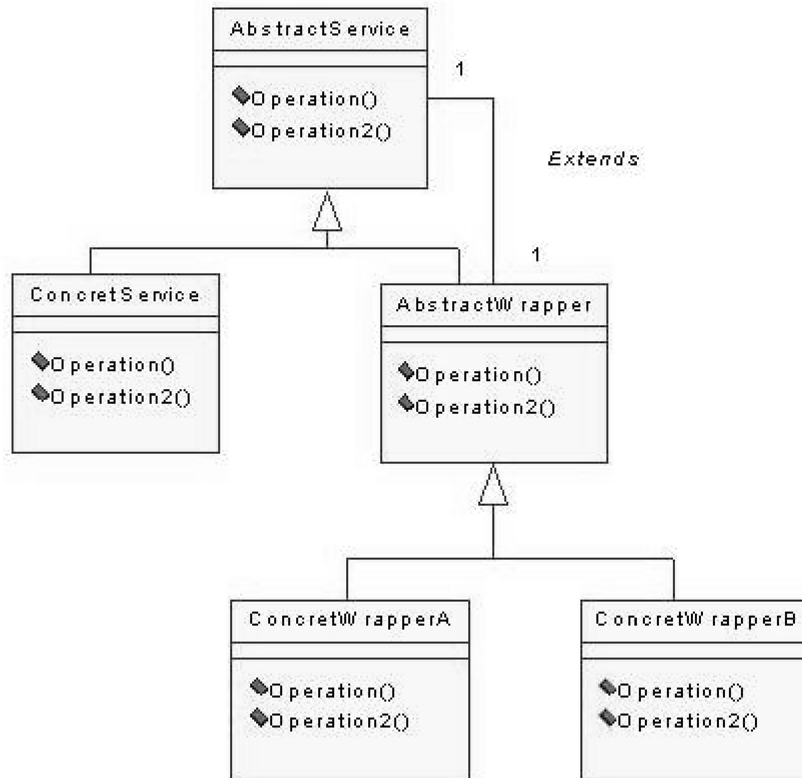

FIGURA 4.6 - Estrutura Abstrata do *Decorator* para a Linguagem Java (GRAND, 1998)

O padrão *Decorator* provê maior flexibilidade em comparação ao mecanismo de herança. O padrão acrescenta dinamicamente comportamento aos objetos individuais, adicionando e removendo funcionalidades. Já o mecanismo de herança acrescenta funcionalidades de natureza estática.

A grande vantagem do padrão *Decorator* é que, na estrutura hierárquica, um objeto *Decorator* pode conter (ou relacionar-se com) outro objeto *Decorator,* e assim sucessivamente (Figura 4.7). Utilizando combinações diferentes por meio dos objetos *Decorator,* podem-se criar muitos comportamentos diferentes. Em comparação ao mecanismo de herança, para prover vários tipos de comportamentos, faz-se necessária a utilização de várias classes predeterminadas.

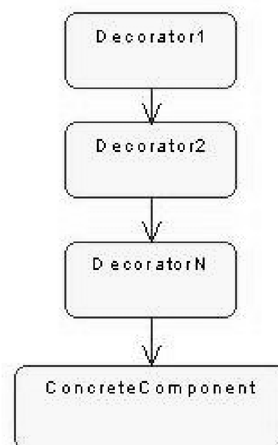

FIGURA 4.7 – Diagrama de Objetos - Padrão *Decorator*

Dentre as estratégias básicas para a implementação do padrão *Decorator,* verificam-se as seguintes:

- A utilização de uma subclasse *ConcretComponent,* em separado, dividirá a hierarquia *Component* em duas subclasses distintas: *Decorator* para as subclasses responsáveis pelo encadeamento e *ConcretComponent* para os objetos primitivos.
- Apenas *Decorator* deve delegar responsabilidades para os seus componentes. Toda subclasse *Decorator* possui um comportamento *default,* o qu al, pelo mecanismo de herança, delega responsabilidades para as suas subclasses.
- A classe *Decorator* não deve delegar responsabilidades diretamente para um *ConcretComponent*. Um *Decorator* deve delegar somente para outro *Decorator*.

4.2.2.1 Essência do padrão *Decorator*

Nas propostas de Gamma (1994), Alpert (1998) e Johnson (1998), é apresentado um relacionamento de agregação (composição) entre as classes *Decorator* e *Component*. Para Grand (1998), o relacionamento é estabelecido por meio de uma associação entre as classes *AbstractWrapper* (ou *Composite*) e *AbstractService* (ou *Component*). Os autores caracterizam tal relacionamento com cardinalidade de 1 para 1. Faz-se necessário, portanto, um atributo na classe *Decorator* (ou *AbstractWrapper*), que irá armazenar, respectivamente, uma referência a objetos que pertençam à estrutura hierárquica, no caso, objetos primitivos da classe *ConcretComponent* ou objetos *Decorator*.

Deve-se caracterizar que a classe *Decorator* contenha métodos para a inserção e remoção de um objeto, a fim de constituir o relacionamento entre os objetos da hierarquia. É importante que, durante o processo de inserção dos objetos, verifique-se um controle para que estes pertençam à superclasse *Component*, representados respectivamente por objetos da classe *ConcretComponent* e da própria classe *Decorator*.

Outro ponto importante que caracteriza o padrão é que os métodos abstratos na classe *Component*, como *operation,* por exemplo, nas propostas dos autores, devem caracterizar, nos objetos primitivos, o retorno do processamento de alguma informação e, nos objetos da classe *Decorator,* devem apresentar o envio da mensagem *operation* a outro objeto envolvido no relacionamento, adicionando ainda um possível comportamento, se necessário for. Isso fará com que um objeto *Decorator* envie mensagem a outro objeto *Decorator* ou envie mensagem a um objeto primitivo, dependendo da quantidade de objetos envolvidos na relação.

A solução do padrão, conforme Figura 4.8, é semelhante à proposta no padrão *Composite*. Portanto, a descoberta também pode ser realizada pela procura de ciclos em um grafo de herança/composição. Sempre que uma classe tenha uma relação de agregação com uma classe que descende da hierarquia de herança, há uma evidência forte de que o padrão *Decorator* pode estar sendo utilizado, levando-se em consideração um indicador de multiplicidade de 1 para 1. A simplicidade do padrão *Decorator* propõe um bom candidato para a descoberta automática.

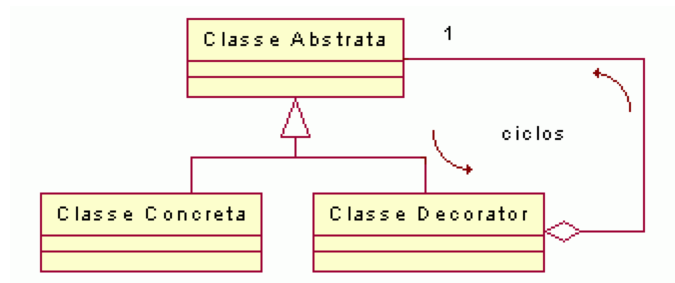

FIGURA 4.8 – Ciclos em *Decorator*

4.2.2.2 Identificação do padrão *Decorator*

A formalização para a identificação do padrão *Decorator* foi elaborada de maneira similar à do padrão *Composite*. Iniciando pela definição das classes, identifica-se a existência de duas subclasses a partir da classe abstrata A: a classe D, que demonstra a estrutura de objetos *Decorator,* e a classe C, que caracteriza a estrutura do objeto *ConcretComponent*.

A classe D demonstra, também, uma referência a objetos da superclasse abstrata. Esta referência é estabelecida via um atributo que representa um objeto das classes D ou C.

O mecanismo de agregação e delegação é definido quando existe uma associação e uma referência entre o objeto em avaliação (OD – objeto *Decorator*) e o objeto representado pelo atributo. Cada associação deve traduzir uma relação com um objeto de uma classe semelhante (outro objeto *Decorator*) ou com um objeto que represente a subclasse *ConcreteComponent*, os quais identificam objetos da mesma classe raiz avaliada. Para os objetos assinalados como *Decorator,* repete-se o procedimento, recursivamente, na verificação da existência de uma hierarquia. Estas evidências, portanto, caracterizam fortes indícios para a existência do padrão *Decorator.*

Adotando a notação utilizada no trabalho de Seeman (1998) pode-se deduzir o seguinte formalismo:

$\text{CLASS}(D) = \{D \mid D\ \textit{extends}\ A\ \wedge\ D\ \textit{references}\ A\}$
$\text{CLASS}(C) = \{C \mid C\ \textit{extends}\ A\}$

Label_Decorator(OD)

$\text{ATRIB}(At) = \{At \mid \exists At \in D\ \vee\ \exists At \in C\}$
$\exists OD \in D{:}\ \forall At{:}\ OD\ \textit{aggreg}\ At\ \wedge\ OD\ \textit{delegates}\ At$
$\forall OD \in D{:}\ \ OD\ \textit{agreg}\ At \Leftrightarrow OD\ \textit{assoc}\ At(D)\ \wedge\ OD\ \textit{references}\ At(D)\ \vee$
$OD\ \textit{assoc}\ At(C)\ \wedge\ ODC\ \textit{references}\ At(L)$
$\forall OD \in D{:}\ \ OD\ \textit{delegates}\ At \Leftrightarrow\ \forall m_1 \in OD{:}\ m_1\ \textit{calls}\ m_2$
$OD\ \textit{owns}\ m_1\ \wedge\ At\ \textit{owns}\ m_2$
$\Rightarrow \phi(m(OD))_1 = \phi(m(At))_1$
$\forall At \in D{:}\ \text{Label_Decorator}(At)$

A contribuição apresentada neste formalismo é a mesma citada para o padrão *Composite,* no qual o objetivo é demonstrar a especificação que possibilita verificar a essência do padrão a partir da estrutura dinâmica da aplicação, não apenas levando em consideração a estrutura estática.

4.2.3 Padrão *Strategy*

O padrão *Strategy* tem por função encapsular uma família de algoritmos ou comportamentos, de forma que estes algoritmos possam ser variados independentemente dos clientes que os usam (JOHNSON, 1998). A utilização do padrão se dá em situações nas quais existem diferentes algoritmos que resolvem o mesmo problema com diferentes restrições de espaço-tempo, ou para um conjunto de classes relacionadas que somente diferem numa porção do seu comportamento.

O mecanismo utilizado para obter esta funcionalidade é a herança, na qual subclasses definem as variações de comportamento (Figura 4.9). No entanto, este esquema fixa a implementação sem permitir mudar dinamicamente o comportamento.

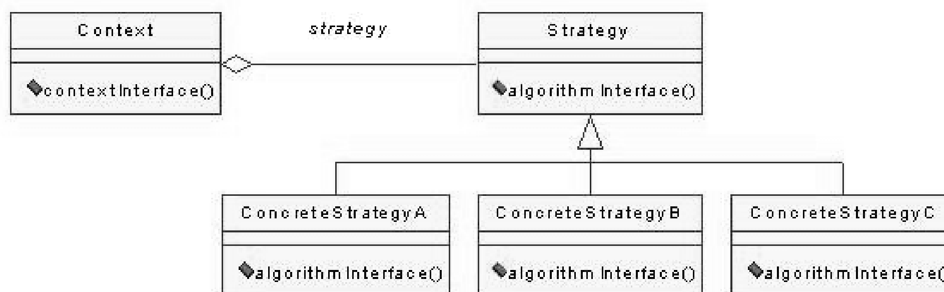

FIGURA 4.9 - Estrutura Abstrata do *Strategy* para a Linguagem C++ (GAMMA, 1994)

A classe *Strategy* declara uma interface comum (*algoritmInterface*) para suportar todos os algoritmos. Os objetos *Context* utilizam a interface para chamar um algoritmo específico definido por um objeto *ConcreteStrategy*. *Context* pode implementar uma interface para permitir aos objetos *ConcreteStrategy* acessarem seus dados. Desta forma, sempre que o cliente solicita serviços a um objeto *Context,* ele os repassa para um objeto *Strategy*. A tarefa de criação do objeto *Strategy* para um contexto é do cliente, sendo que, ao fazer isso, disponibilizam-se vários objetos *ConcreteStrategy*.

O objeto *Strategy* comporta-se como um *help* externo para *Context* e encapsula um conjunto de algoritmos. Portanto, algoritmos são encapsulados fora do escopo de *Context* e cada um é refinado como um objeto. O objeto *Strategy* é chamado periodicamente, tanto quanto necessário, para executar uma tarefa *stand-alone* em uma determinada tentativa de execução.

A proposta de Alpert (1998), para aplicação do padrão *Strategy* em *Smalltalk,* caracteriza-se muito semelhantemente ao modelo definido por Gamma.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, o autor modifica pouco a diagramação do padrão original. Utilizando notação UML, este substitui a relação de agregação por uma relação de uso entre *Client* (*Context*) e *AbstractStrategy* (*Strategy*). A Figura 4.10 apresenta o diagrama referente ao padrão.

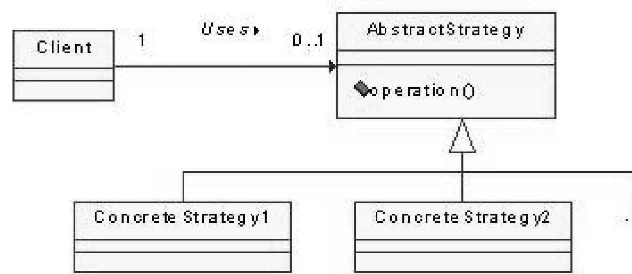

FIGURA 4.10 - Estrutura Abstrata do Strategy para a Linguagem Java(GRAND, 1998)

4.2.3.1 Essência do padrão *Strategy*

O objeto *Strategy* comporta-se como um *help* externo para *Context* e encapsula um conjunto de algoritmos. Portanto, algoritmos são encapsulados fora do escopo do cliente e cada um é refinado como um objeto. O objeto *Strategy* é chamado periodicamente, tanto quanto necessário, para executar uma tarefa *stand-alone* em uma determinada tentativa de execução.

O padrão caracteriza um objeto que faz referência a outros objetos classificados como servidores de um determinado serviço. Estes objetos servidores formam um conjunto no qual todos compõem a estrutura de herança com a classe servidora abstrata. O objeto cliente utiliza, portanto, o protocolo comum definido na classe servidora abstrata para comunicar-se com os servidores concretos. A Figura 4.11 demonstra a estrutura básica a ser avaliada, bem como a invocação de métodos definidos na classe abstrata e redefinidos nas classes concretas.

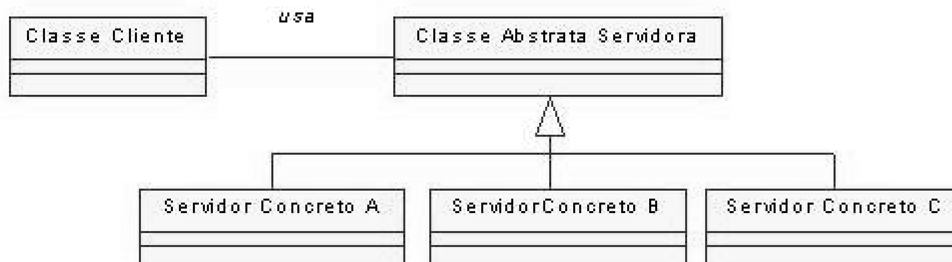

FIGURA 4.11 - Estrutura Elementar do *Strategy*

4.2.3.2 Identificação do padrão *Strategy*

A formalização para a identificação do padrão *Strategy* foi elaborada levando em consideração a definição das seguintes classes: CS, que demonstra a estrutura de objetos *ConcreteStrategy,* e C, que caracteriza a estrutura do objeto *Client*.

O mecanismo de delegação é definido quando o objeto da classe cliente faz a chamada de um método em uma classe *ConcreteStrategy*. Cada método a ser executado é comparado com os outros métodos de mesma assinatura de possíveis objetos *ConcreteStrategy* existentes. Para que se caracterize a existência do padrão, estes métodos devem ser implementados de forma diferente.

Adotando à notação utilizada no trabalho de Seeman (1998), pode-se deduzir o seguinte formalismo:

CLASS(CS) = {CS | CS *extends* S}
CLASS(C) = {C}

Label_Strategy(OS)

$\exists OC \in C$: OC(C) *delegates* CS(S) $\Leftrightarrow$ $\forall m_1 \in OC$: m_1 *calls* m_2
OC *owns* m_1 ^ CS *owns* m_2
$\Rightarrow \phi(m(OC))_1 = \phi(m(CS))_1$ ^
$\exists CSn \in CS : m_1 \neq m_2$
CSn *owns* m_1 ^ CS *owns* m_2

A contribuição apresentada neste formalismo é a mesma citada para os padrões anteriores, no qual o objetivo é demonstrar a especificação que possibilita verificar a essência do padrão a partir da estrutura dinâmica da aplicação, não apenas levando em consideração a estrutura estática.

4.2.4 Padrão *Template Method*

O padrão *Template Method* define o esqueleto de um algoritmo na superclasse, propiciando a definição de alguns passos, desse algoritmo, pelas subclasses. O padrão permite que as subclasses possam redefinir certos passos de um algoritmo sem mudar a estrutura do algoritmo (ALPERT, 1998). Trata-se de uma técnica fundamental de reutilização de código. O padrão, conforme Figura 4.12, permite generalizar o comportamento comum de várias subclasses, representando o algoritmo que elas utilizam para implementar uma operação. Os aspectos específicos do algoritmo podem ser implementados por métodos fornecidos por subclasses, os quais são invocados pelo algoritmo genérico.

A classe *AbstractClass* implementa um método *TemplateMethod(),* o qual define o esqueleto de um algoritmo. Ela também fornece a interface para as operações primitivas abstratas (*PrimitiveOperations*), invocadas pelo algoritmo. As subclasses concretas (*ConcreteClass*) redefinem as operações primitivas para implementar os passos específicos do algoritmo.

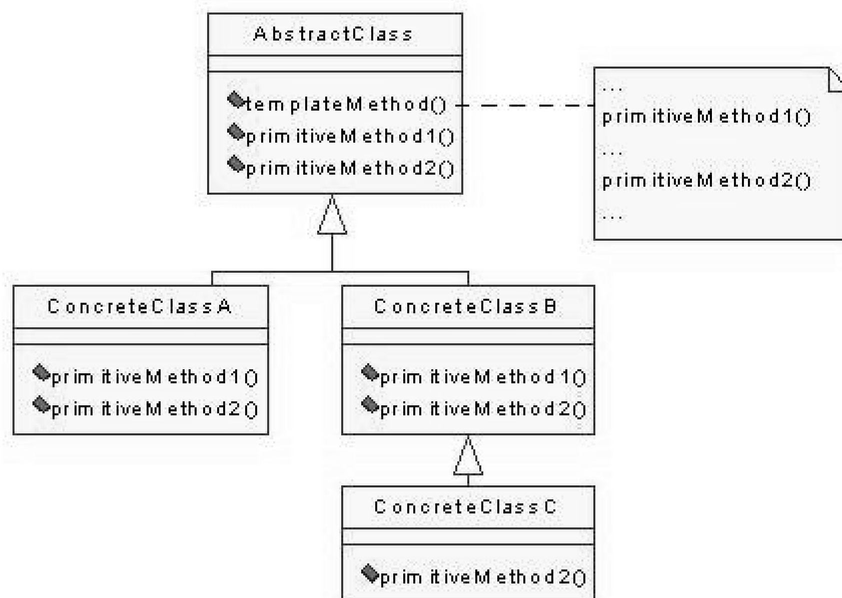

FIGURA 4.12 - Estrutura Abstrata do *Template Method* para a linguagem C++ (GAMMA, 1994)

A proposta de Alpert (1998), para aplicação do padrão *Template Method* em *Smalltalk,* constitui algo muito semelhante ao modelo definido por Gamma.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, Figura 4.13, este simplifica consideravelmente a diagramação do padrão original.

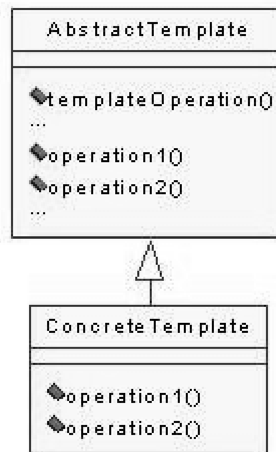

FIGURA 4.13 - Estrutura Abstrata do *Template Method* para a Linguagem Java (GRAND, 1998)

Classes abstratas e o padrão *Template Method* possuem características em comum. Uma classe abstrata é uma superclasse que define um comportamento e permite, por meio do mecanismo de herança, a redefinição de propriedades. A superclasse é abstrata porque não implementa todo o comportamento necessário ao funcionamento da aplicação, e as subclasses são concretas porque implementam particularidades do comportamento abstrato. O padrão *Template Method* utiliza a mesma técnica das classes abstratas e faz referência aos seguintes métodos: *template, hook* e concretos.

Os métodos *template* implementam algoritmos genéricos que definirão o fluxo de controle da aplicação. Métodos *template* combinam métodos concretos e *hook* em um mesmo algoritmo. Os métodos concretos são aqueles que possuem um comportamento padrão. Por fim, os métodos *hook* apresentam um comportamento padrão para alguns aspectos da aplicação. Estes métodos podem ainda ser redefinidos por subclasses concretas, a fim de fornecer um novo comportamento para a aplicação.

4.2.4.1 Essência do Padrão *Template Method*

O padrão apresenta as seguintes características:

- Implementação simples: os projetistas implementam em geral todo o código em um único método. Este método é considerado o método *template*.
- Divisão do método *template* em partes: o método *template* é dividido em partes lógicas.
- Métodos parte: a implementação separada se dá através das operações primitivas nas subclasses.
- Chamadas aos métodos parte: invocação de operações primitivas;

A identificação do padrão *Template Method* depende da classificação dos tipos dos métodos na classe abstrata. Em um primeiro momento, é importante procurar por métodos virtuais e depois encontrar as suas chamadas em um possível método *template*.

Por fim, verificam-se os métodos *hook* nas subclasses da aplicação por meio de suas possíveis implementações como métodos concretos.

É importante salientar que é muito comum o uso de métodos virtuais, sem a caracterização do padrão, mas no sentido de determinar ao projetista a redefinição destes nas subclasses da hierarquia.

4.2.4.2 Identificação do Padrão *Template Method*

A formalização para a identificação do padrão *Template Method* leva em consideração o objeto *template* concreto em avaliação (OT). A classe CT representa a classe *template* concreta, e AT, a classe abstrata.

A identificação do padrão é elaborada a partir da verificação da chamada de métodos redefinidos na própria classe, que são definidos na superclasse, em geral, como métodos abstratos. Portanto, para cada método do objeto concreto *template,* faz-se a identificação das mensagens enviadas para ele próprio, levando em consideração a execução de métodos predefinidos.

Adotando a notação utilizada no trabalho de Seeman (1998) pode-se deduzir o seguinte formalismo:

CLASS(CT) = {CT | CT *extends* AT}

Label_Template_Method(OT)
$\exists OT \in CT$: $\forall m_1 \in OT$: m_1 *calls* m_2
OT *owns* m_1 ^ OT *owns* m_2
$\Rightarrow \phi(m(OT))_1 = \phi(m(OT))_2$ ^
$\exists m_2 \in AT : (m_2(CT))1 \neq (m_2(AT))2$
CT *owns* $(m_2)1$ ^ AT *owns* $(m_2)2$

A contribuição apresentada neste formalismo é a mesma citada para os padrões anteriores, no qual o objetivo é demonstrar a especificação que possibilita verificar a essência do padrão a partir da estrutura dinâmica da aplicação, não apenas levando em consideração a estrutura estática. A partir dessa estrutura, é possível observar se as diversas opções de chamada a métodos concretos estão sendo bem utilizadas.

4.2.5 Padrão *Observer*

O padrão *Observer* define uma relação de dependência de um para muitos, entre objetos, de modo que quando um objeto muda seu estado, todos os seus dependentes são notificados e atualizados automaticamente. O relacionamento proposto no trabalho de Gamma (1994), utilizando a metodologia OMT, conforme Figura 4.14, caracteriza um objeto que detém um determinado estado, chamado *Subject*, e outro que depende de modificações propostas no estado de *Subject*, chamado *Observer*.

O padrão *Observer* tem por finalidade preservar restrições de sincronização, coordenação e consistência entre objetos. Tipicamente, o padrão é utilizado para manter consistentes as múltiplas visões de um objeto da aplicação, podendo ser aplicado também para manter consistente qualquer conjunto de objetos que apresentem dependências mútuas.

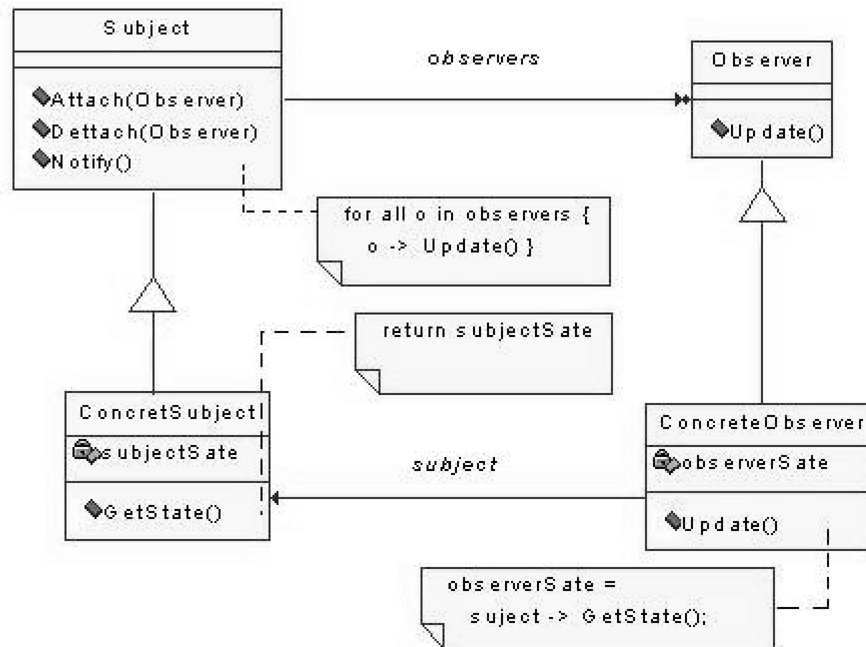

FIGURA 4.14 - Estrutura Abstrata do *Observer* para a Linguagem C++ (GAMMA, 1994)

O padrão define basicamente um mecanismo de comunicação de eventos. Estes eventos são proporcionados pelo objeto observado *Subject* por meio do método *Notify(),* quando se produz alguma mudança de interesse no seu estado interno. Os objetos dependentes (*Observers*) recebem a notificação desta mudança mediante o método *Update()* e passam a interagir com seu *Subject* para obter a informação necessária para atualizar seu próprio estado por meio de *GetSate().*

A proposta de Alpert (1998), para aplicação do padrão *Observer* em *Smalltalk,* mostra-se muito semelhante ao modelo definido por Gamma. Alpert utiliza o mesmo diagrama de Gamma com a definição e representação dos métodos na linguagem *Smalltalk*. A diferença está na nomenclatura dos métodos: *addDependent* utilizado por *Attach*, *removeDependent* utilizado por *Dettach*, *changed* utilizado por *Notify*, *sujectState* utilizado por *GetState* e, finalmente, *observerState* utilizado por *Update*. Faz-se diferença na substituição do método *do,* pelo tradicional *for* no método *changed* (*Notify*).

Na proposta definida por Grand (1998), Figura 4.15, que objetiva aplicar padrões à linguagem Java, o autor modifica a diagramação do padrão. Utilizando notação UML, o autor caracteriza as classes *ObserverIF* e *ObservableIF*. *ObserverIF* representa uma interface a qual define os métodos *notify* e *update*, tipicamente chamados. Um objeto observado envia estas mensagens para prover a notificação de que seu estado foi alterado. *Observer* representa as instâncias da classe *ObserverIF*.

ObservableIF representa a interface para os objetos observados. Esta classe caracteriza a implementação de métodos para adição e remoção dos objetos *Observers*. *Observable* representam os objetos observados. Os objetos desta classe têm por função manter o relacionamento com os objetos *Observers*, mas não implementam diretamente estas responsabilidades, passando tal controle para a classe *Multicaster*. *Multicaster*, portanto, é responsável pelas notificações aos objetos *Observers* quando da mudança de estado dos objetos observados.

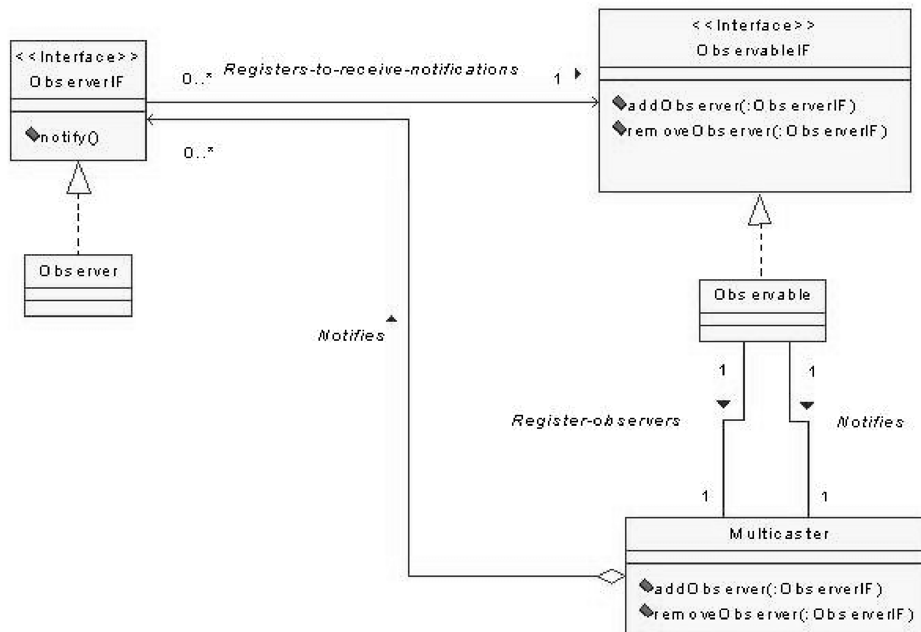

FIGURA 4.15 - Estrutura Abstrata do *Observer* para a Linguagem Java (GRAND 1998)

Apesar da aparência modificada, as funcionalidades apresentadas por Grand são muito semelhantes às de Gamma e Alpert. Dentre as estratégias básicas para implementação do padrão *Observer,* verificam-se as seguintes:

- Necessidade de métodos para o acesso a dependentes:
  Attach ou *addDependent* - adiciona um objeto como dependente.
  Dettach ou *removeDependent* - remove um objeto dependente.
- Necessidade de uma coleção de dependentes – *observers* ou *dependents*.
- Necessidade de um método para trocas de informação: o método *notify* (ou *changed*) representa o método pelo qual *Subject* (ou *Multicaster*) notifica os dependentes, quando alterações são propostas no seu estado.
- Necessidade de um método para alteração de informações – o método *update* caracteriza o método *default* para modificação do objeto *Observer*.

Para Grand *notify* e *update* possuem o mesmo sentido. Para Gamma e Alpert *notify* (ou *changed*) representa o método que caracteriza a notificação de troca de estado, e *update* representa o método que alerta a troca para os objetos *Observers*.

4.2.5.1 Essência do padrão *Observer*

Nas propostas de Gamma (1994), Alpert (1998) e Johnson (1998), é apresentado um relacionamento de associação entre as classes *Subject e Observer*. Para Grand (1998), o relacionamento é estabelecido por meio de uma composição (agregação) entre as classes *Multicaster e ObserverIF.*. Os autores caracterizam o relacionamento com cardinalidade de 1 para N, onde um objeto *Subject* pode relacionar-se com N dependentes (ou *Observers*).

Os exemplos propostos caracterizam um atributo na classe *Subject* (ou *ObservableIF*) em forma de coleção. Esta coleção irá armazenar, respectivamente, referência aos objetos que formam o relacionamento.

Na classe *Subject,* existem métodos para inserção e remoção dos objetos na coleção. Outro método, necessário, é o método *changed,* responsável em sinalizar as

trocas de estado aos objetos observadores. Deve-se atentar para que, durante o processo de inserção dos objetos à coleção, estes se caracterizem como objetos do tipo *Observer,* ou seja, objetos que contenham o método *update,* o qual caracteriza o método receptor da mensagem de notificação de troca de estado do objeto observado.

Outro ponto importante que caracteriza o padrão é que o método *changed* realiza uma repetição para todos os elementos da coleção de *Observers,* notificando as possíveis trocas de estado. Para o método *update,* deve-se prover a solicitação do estado do objeto observado.

4.2.5.2 Identificação do Padrão *Observer*

A formalização para a identificação do padrão *Observer* é elaborada a partir do objeto *Subject* concreto (OSC). A classe CS representa a classe *Subject* a qual possui os métodos para adição, remoção e notificação dos objetos *Observer*. A classe que define um objeto *Observer* concreto é definida por COC, a qual herda propriedades da classe CO (classe *Observer*).

A identificação do padrão é proposta com a verificação dos atributos pertencentes ao objeto *Subject* concreto. O relacionamento entre o objeto concreto *Subject* e possíveis objetos *Observer* é determinado pela associação entre estes. Quando *Subject* envia a mensagem *notify* para objetos *Obsever*, isso ocasiona a execução do método *update* ou até mesmo *notify* (segundo Grand) nos objetos *Observer*.

Adotando à notação utilizada no trabalho de Seeman (1998) pode-se deduzir o seguinte formalismo:

CLASS(CSC) = {CSC | CSC *extends* CS}
CLASS(COC) = {COC | COC *extends* CO}
LISTATRIBS(LAs) = {LAs | LAs *attrib* CS ^ LAs = LIST}

Label_Observer(OSC)
ATRIB(At) = {At = Las[n] | ∃At ∈ COC}
∃OSC ∈ CS: ∀At: OSC *aggreg*(*multiple*) At ^ OSC *delegates* At
∀OSC ∈ CS: OSC *agreg* At ⇔ OSC *assoc* At(COC) ^ OSC *references* At(COC)
∀OSC ∈ CSC: OSC *delegates* At ⇔ ∀notify ∈ OSC: notify *calls* update v
notify *calls* notify ^
OSC *owns* notify ^
At *owns* update v
At *owns* notify

A partir desta formalização, é possível observar se as chamadas de notificação estão sendo bem utilizadas no intuito de alterar os estados dos objetos observadores.

4.2.6 Padrão *Chain of Responsibility*

O padrão *Chain of Responsibility* permite que um objeto envie uma mensagem sabendo qual objeto ou objetos irão recebê-la. Isto acontece porque a mensagem é enviada para uma cadeia de objetos que é tipicamente parte de uma estrutura da hierarquia de classes. Cada objeto, na cadeia, possui um *handle* que envia a mensagem para o próximo objeto na cadeia (Gamma, 1994).

O padrão apresenta uma classe *Handler* que define a interface para manipular requisições de serviço (*handleRequest*) e mantém uma referência para seu sucessor. Cada *ConcreteHandler* implementa *handleRequest* de acordo com sua responsabilidade, sendo que pode repassá-lo ao seu sucessor se não for de seu interesse A Figura 4.16

apresenta o modelo de classes para o padrão.

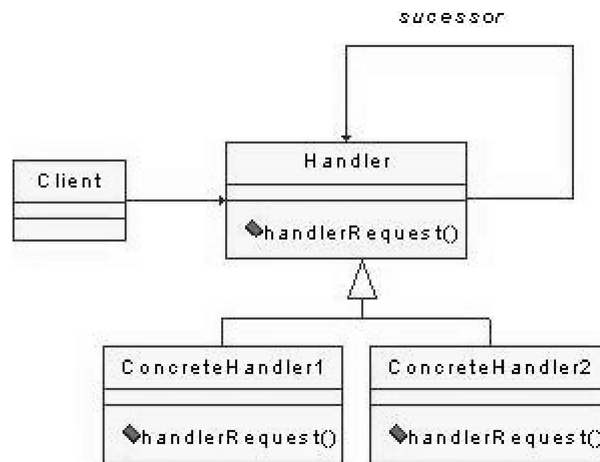

FIGURA 4.16 - Estrutura do *Chain of Responsability* para a Linguagem C++ (GAMMA, 1994)

Uma hierarquia de classes é um bom exemplo de como *Chain of Responsibility* trabalha. Na raiz apresenta-se a classe, mais geral, *Object* e subclasses são implementadas especificando o comportamento. Supondo como exemplo quatro classes A, B, C e D, onde D é subclasse de C, C é subclasse de B e B é, respectivamente, subclasse de A. Conforme a Figura 4.17, *métodox* é descrito na classe A e redefinido na classe B. Se enviada a mensagem *métodox* para uma instância de D (*objetoD*), o sistema deve prover a execução do método em D. Como não há implementação deste método em D, a execução é passada para a classe C. Como C também não dispõe da implementação do método, o controle é passado finalmente para B, que retorna o seu resultado. A implementação de *métodox* em A é irrelevante neste exemplo, pois o pedido de execução não é passado para A.

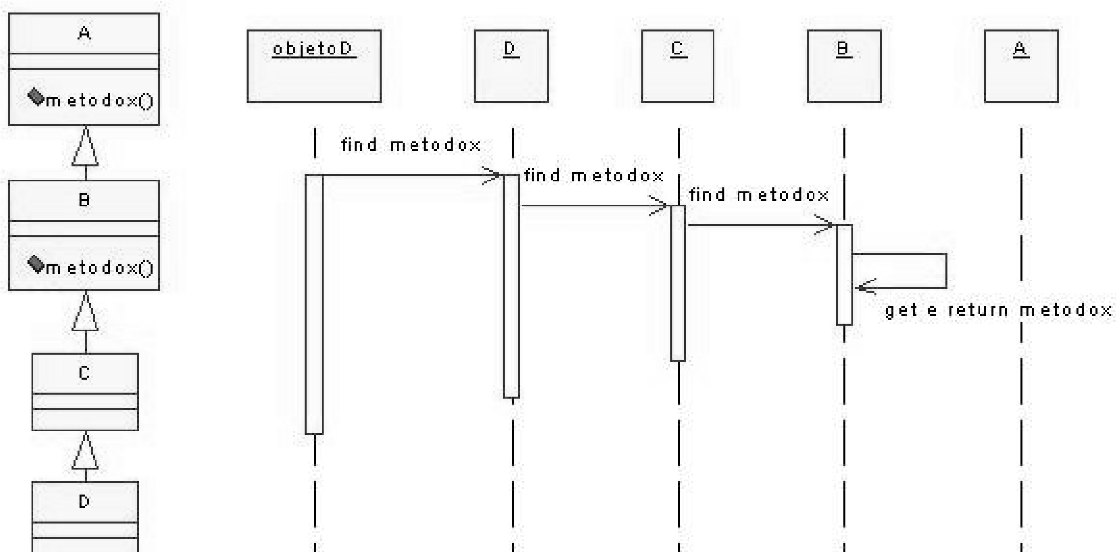

FIGURA 4.17 - Exemplo de Execução para *Chain of Responsability*

A proposta de Alpert (1998), para aplicação do padrão *Chain of Responsibility* em *Smalltalk,* apresenta-se muito semelhante ao modelo definido por Gamma.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, o autor utiliza a mesma diagramação do padrão com a nomenclatura um pouco modificada.

4.2.6.1 Essência do padrão *Chain of Responsibility*

O *padrão Chain of Responsibility* faz uso de uma cadeia de objetos que prevê *links* os quais referenciam outros elementos na cadeia. De acordo com as responsabilidades de execução de cada objeto, este pode passar novas responsabilidades para outros elementos.

O cliente que inicializa uma requisição desconhece qual objeto irá executar a tarefa, pois as responsabilidades de execução ficam a cargo da distribuição feita pela cadeia.

É importante salientar que a investigação, a partir da mensagem enviada para o objeto, deve determinar se esta mensagem pertence ao protocolo da classe verificada ou a alguma das superclasses da hierarquia.

Para que se possa otimizar e reduzir o espaço de procura pelo padrão, é importante definir algumas regras:

- limitar à procura a implementações de métodos com o mesmo nome em todos os casos. *Chain of Responsibility* não requer que o mesmo nome de método seja invocado em todos os elementos da cadeia. Com esta restrição, o problema remove dependências semânticas;
- restringir a procura a mensagens enviadas a receptores conhecidos. Uma vez que foram identificados e conhecidos atributos para armazenar valores de um conjunto particular de classes, é importante concentrar esforço nestas variáveis que podem participar de uma cadeia;
- ignorar mensagens enviadas quando os objetos são passados como argumentos ou como variáveis temporárias.

4.2.6.2 Identificação do Padrão *Chain of Responsibility*

A formalização para a identificação do padrão *Chain of Responsibility* é elaborada a partir do objeto concreto receptor da mensagem de execução. A mensagem é recebida e, caso, o objeto não a reenvie, está será executada. Caso o objeto faça o reenvio da mensagem, está será recebida por outro objeto da cadeia que poderá propor a sua execução ou o seu reenvio a outro. Adotando à notação utilizada no trabalho de Seeman (1998), pode-se deduzir o seguinte formalismo:

CLASS(CC) = {CC | CC *extends* H}

Label_Chain_of_Responsibility(OC)
$\exists OC_2 \in H$:
$\exists\, m_1 \in OC(H)$: $\neg\exists\, m_1$ *calls* m_2
OC *owns* m_1 ^ OC_2 *owns* m_2
$\Rightarrow$ return
$\exists OC \in H$: $\forall m_1 \in OC$: m_1 *calls* m_2
OC *owns* m_1 ^ OC_2 *owns* m_2
$\Rightarrow \phi(m(OC))_1 = \phi(m(OC))_2$
Label_Chain_of_Responsability(OC2)

A partir desta formalização, é possível observar o envio da mensagem a outro

objeto, desde que este não se responsabilize pela execução da mensagem e sim se comprometa a manter o fluxo da cadeia.

4.2.7 Padrão *State*

State permite um objeto mudar seu comportamento à medida que seu estado interno muda, como se tivesse trocado de classe (ALPERT 1998). Em geral, o padrão é aplicável quando a resposta de um objeto às mesmas mensagens varia em função de seu estado interno. A separação de estados em classes simplifica a implementação da classe principal, evitando a escrita de código condicional, embora que, para isso, aumente o número de classes do sistema.

A classe *Context* é responsável por definir a interface de interesse de seus clientes e por manter uma instância de um *ConcreteState* que define o estado corrente. *State* define uma interface para encapsular o comportamento associado com um particular estado do contexto. Os comportamentos específicos de cada estado são implementados por *ConcreteStates* (Figura 4.18).

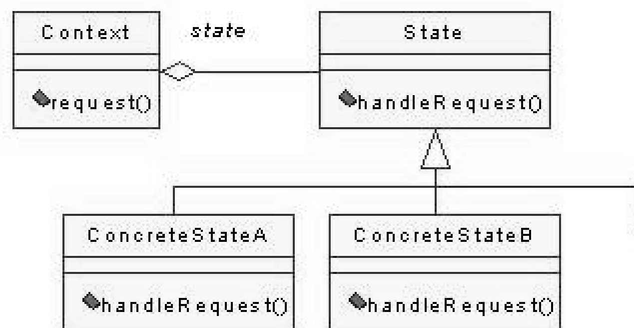

FIGURA 4.18 - Estrutura Abstrata do *State* para a Linguagem C++ (GAMMA, 1994)

A proposta de Alpert (1998) para aplicação do padrão *State* em *Smalltalk* apresenta-se muito semelhante ao modelo definido por Gamma.

Na proposta definida por Grand (1998), que objetiva aplicar padrões à linguagem Java, o autor utiliza uma diagramação estilizada como alguns métodos adicionais, conforme Figura 4.19.

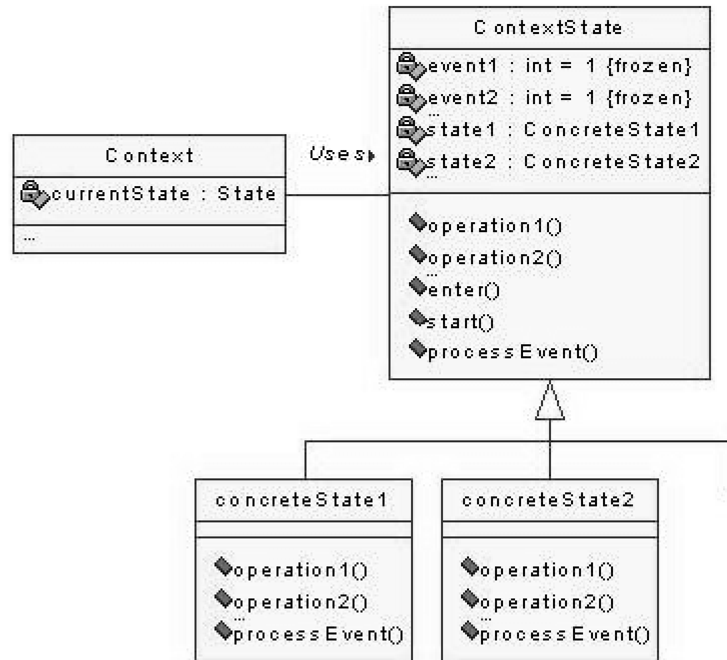

FIGURA 4.19 - Estrutura Abstrata do *State* para a Lingaugem Java (GRAND, 1998)

4.2.7.1 Essência do Padrão *State*

Dentre as estratégias utilizadas para identificação do padrão *State,* destaca-se a procura de um atributo (representado por um objeto) que apresenta diferentes definições as quais representam vários estados.

Em geral, os projetistas definem um contexto limitado onde se caracteriza com clareza a lista de estados possíveis. Portanto, o estado que define o *status* corrente possui apenas poucos valores válidos para cada um dos valores que podem ser implementados nas subclasses *State*. Se os valores de estado podem apresentar um número ilimitado de subclasses, o padrão *State* não é apropriado.

Uma vantagem do padrão *State* é que este remove o comportamento de estado/ dependência do contexto e encapsula isso em objetos *State*. Similarmente, o padrão deve extrair do comportamento das transições de estado do contexto as informações de estados e manipular os objetos conforme necessário.

Estruturalmente, o padrão pode parecer em muito com *Strategy,* mas a identificação em *State* visa, basicamente, encontrar um objeto que troca de estados, representados pelas subclasses da hierarquia. Já *Strategy* visa a encontrar algoritmos diferentes, representados nos métodos das subclasses da hierarquia.

4.2.7.2 Identificação do Padrão *State*

A formalização para a identificação do padrão *State* leva em consideração o objeto concreto avaliado e suas possíveis trocas de estados. Como se dispõe das informações que compõem o modelo dinâmico, é possível avaliar todos os estados pelos quais o objeto passa. O importante é, justamente, verificar se o estado atual é semelhante ao anterior, ou se este foi modificado para um outro estado concreto disponível na hierarquia.

Adotando à notação utilizada no trabalho de Seeman (1998), pode-se deduzir o seguinte formalismo:

CLASS(OC) {OC | OC *extends* S}
$CLASS(OC_n)$ $\{OC_n \mid OC_n$ *extends* $S\}$

Label_State(Context)

$\exists m_1$: m_1 creates OC(S) ^ Context *owns* m_1
$\exists OC \in S$:
$\exists OC_2 \in S$: $CLASS(OC) \neq CLASS(OC_2)$

A formalização do padrão *State* leva em consideração, justamente, a disponibilidade dos estados pelos quais pode passar um objeto em tempo de execução.

4.3 Conclusões

O objetivo deste capítulo é apresentar alguns padrões conhecidos na literatura e mostrar como podem ser investigados artefatos que demonstrem evidências da existência de algum padrão em um sistema de software orientado a objetos.

O mecanismo de descoberta confia, portanto, em um conjunto comum de informações que devem ser investigadas a partir do sistema. Esse conjunto de informações é referido aqui como essência do padrão.

Verifica-se, portanto, que os padrões citados apresentam boas chances de reconhecimento a partir da investigação dos elementos que compõem cada uma de suas essências. Conforme citado, *Interpreter* e *Adapter*, entre outros, não apresentam chances satisfatórias para identificação devido à diversidade de implementações. Soluções ambíguas também dificultam o processo, pois uma solução pode servir para padrões diferentes.

No caso dos padrões citados neste capítulo, todos representam importantes mecanismos de solução de projeto OO (GAMMA, 1994) apresentando soluções distintas e não-ambíguas. É importante salientar que as avaliações dos padrões não leva em consideração somente o modelo estrutural, mas também o modelo dinâmico considerando as trocas de estados dos objetos e envio de mensagens.

É importante, também, que se faça um estudo mais aprofundado para os outros padrões, na busca de elementos que possam vir a restringir possíveis ambigüidades. Este trabalho não é, portanto, definitivo em suas conclusões, pois deve-se levar em consideração a existência de padrões além dos citados no trabalho de Gamma (1994), como, por exemplo, os abordados por GRAND (1998).

Finalmente, acredita-se que estas informações, as quais representam a essência dos padrões, venham a incrementar a ferramenta proposta no sentido de facilitar o processo de manutenção de um sistema de software, demonstrando e relacionando possíveis informações que venham a contribuir para elevar os índices de qualidade do processo.

No próximo capítulo será apresentada uma descrição de como foram elaboradas as extrações estática e dinâmica das informações, a partir de uma aplicação escrita na linguagem Java para identificação de padrões de projeto.

5 Extrações Estática e Dinâmica das Informações

Um modo óbvio de prover informações a respeito de um código fonte é permitir ao projetista verificar os tipos das variáveis identificadas. Na realidade, este propósito simples permite focalizar no código as representações de projeto para a descoberta de padrões.

Porém, enquanto é simples trabalhar com este tipo de informação, torna-se, freqüentemente, difícil determinar que tipos podem assumir um atributo diretamente pelo exame do código. Um atributo, por exemplo, de uma dada superclasse, pode assumir em tempo de execução quaisquer objetos pertencentes às subclasses desta. Faz-se necessário, portanto, avaliar ao longo do código possíveis atribuições para verificar as representações do atributo.

Os padrões estruturais apresentam maiores facilidades de identificação, conforme citado nos trabalhos de Krämer (1996), Bansyia (1998), Seeman (1998) e Guéhéneuc (2001). Já os parâmetros criacionais e comportamentais incorrem na pesquisa de detalhes além do modelo estático.

Portanto, é insignificante refletir apenas sobre o modelo estático, levando em consideração que a proposta é verificar não só a existência, mas também a utilização dos padrões em sua completa, ou parcial, representação. Por fim, a idéia é verificar em tempo de execução a utilização dos recursos dos padrões em sua totalidade, e isso ser demonstrado ao projetista. Em muitos casos, devido à falta de documentação ou utilização de recursos de projeto, a implementação do sistema é construída a partir de particularidades inseridas pelo projetista, o que muitas vezes não retrata uma realidade razoável e enxuta.

Esta proposta tem por objetivo apresentar a execução de uma aplicação verificando se os recursos oferecidos pelos padrões são realmente utilizados ou não.

5.1 Extração de Informações

A identificação e a classificação de colaborações entre classes e objetos são feitas, neste trabalho, por meio dos processos de engenharia reversa e de reflexão computacional.

A engenharia reversa, conforme citado anteriormente, tem como objetivo extrair informações da especificação de um software para posterior análise, com a finalidade de identificar os componentes da aplicação e seus inter-relacionamentos. A partir do código fonte em Java, faz-se a identificação dos componentes da aplicação por meio de um gerador de referência cruzada. Um gerador de referência cruzada descreve relações e referências para entidades como classes, métodos e atributos, analisando gramaticalmente aplicações fonte em Java. Essas relações são apresentadas em um relatório, descrevendo quais entidades se referem a outras entidades. O gerador de referência cruzada foi implementado utilizando a ferramenta ANTLR, *ANother Tool for Language Recognition*, uma ferramenta que provê um *framework* para construção de reconhecedores, compiladores e tradutores a partir de gramáticas contendo descrições para aplicações C++ ou Java (PARR, 2001).

Após a construção do relatório de referência cruzada, a ferramenta de inspeção identifica, no código fonte, os atributos a serem avaliados no processo de execução. A aplicação passa a comportar-se como um pacote gerenciado pela ferramenta e, como tal, deve ser compilada. Após a compilação, a ferramenta executa a aplicação em teste e, a partir deste momento, é iniciado o processo de reflexão computacional.

Durante a execução da aplicação em teste, os estados dos objetos utilizados são extraídos e enviados para avaliação no processo de reflexão computacional. A ferramenta, portanto, avalia, além da estrutura estática, a dimensão dinâmica dos objetos durante tempo de execução. Essa avaliação permite identificar as trocas de estados em função de entradas, processamento e saídas diferentes. Localiza-se aqui um ponto de diferença, em relação às propostas descritas, as quais possuem o foco no código fonte.

O estado atual da ferramenta mostra como extrair características de objetos por meio de suas colaborações e apresenta a visualização dos objetos e classes em análise.

Mediante a identificação e a avaliação de colaborações entre classes e/ou objetos, a ferramenta propõe a identificação de alguns padrões de projeto citados na literatura. Para a identificação desses padrões, foram determinadas algumas regras baseadas na essência dos padrões, definidas no capítulo 4.

5.2 O Modelo Proposto

Esta seção visa a apresentar o modelo conceitual da ferramenta proposta, que tem por finalidade automatizar a identificação e classificação de colaborações entre classes e objetos em aplicações Java, com o objetivo de facilitar os processos de documentação e manutenção de sistemas de software orientado a objetos.

A ferramenta apresenta oito processos distintos, representados a partir da Figura 5.1 e descritos a seguir:

- 1. Geração do relatório de referência cruzada: a partir do arquivo de configuração (java.g), o qual representa a descrição gramatical de um tradutor. Com o auxílio da ferramenta ANTLR, são gerados os arquivos referentes ao processador de referência cruzada. De posse desses arquivos já compilados, a aplicação fonte do projetista é, então, submetida ao processo de referência cruzada. Como produto final desta fase, obtém-se um relatório o qual é utilizado como entrada para o processo seguinte;
- 2. Geração do arquivo fonte modificado: a geração do arquivo fonte modificado utiliza como entradas o relatório de referência cruzada e o arquivo fonte original. Modificações, como diretivas de controle, são incluídas na aplicação original para que esta possa fornecer, posteriormente, informações para a ferramenta de inspeção;
- 3. Compilação: após a transformação da aplicação do projetista, o novo código é compilado pelo compilador Java padrão por meio da chamada de uma função externa à ferramenta de inspeção. O compilador gera, portanto, os *byte-codes* para esse código. Caso o processo de compilação ocorra com sucesso, isso é indicado ao projetista; caso contrário, serão apresentadas as mensagens de erro que representam as pendências no código. Um outro aspecto importante a ser salientado é que o projetista deve, antes de utilizar a ferramenta de inspeção, compilar seu código deixando-o livre de erros. Esta observação justifica-se porque a complexidade em identificar os possíveis erros pode aumentar devido às modificações propostas no código.
- 4. Execução: a aplicação modificada comporta-se como um pacote da ferramenta de inspeção e, portanto, uma chamada ao método principal (*runprocessing*) a coloca em execução;
- 5. Extração de informações: diante da aplicação em execução, uma linha de execução da ferramenta preocupa-se com a extração das informações dos objetos utilizados na aplicação. Para cada objeto utilizado na aplicação do usuário, será gerada uma cópia de suas características;
- 6. Reflexão computacional: neste processo são avaliados todos os estados pelos quais os objetos tenham sido representados durante a execução da aplicação, bem como seus atributos, métodos, etc;

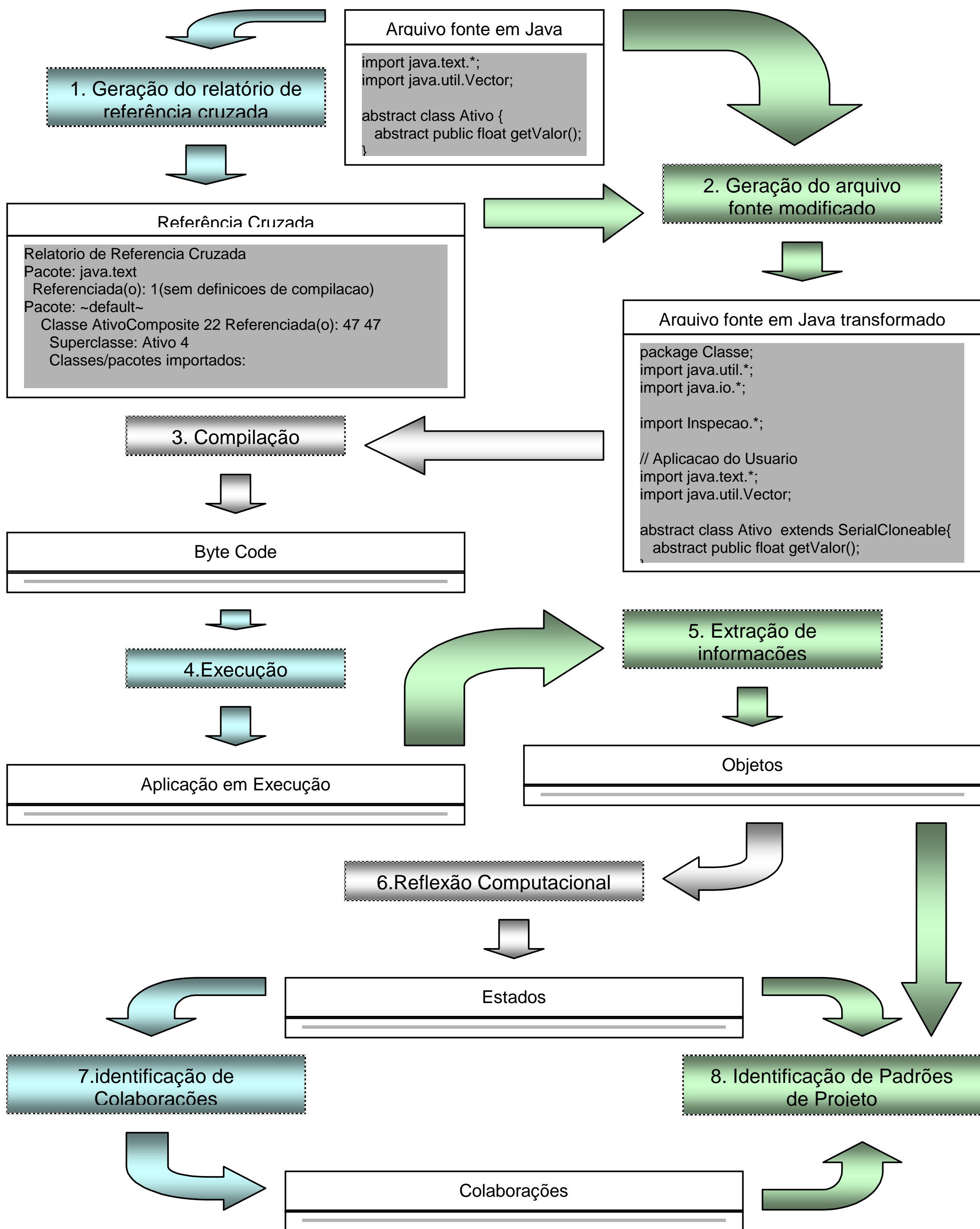

FIGURA 5.1 – Processos Executados na Ferramenta de Inspeção

- 7. Identificação de colaborações: são verificadas todas as colaborações entre classes e objetos. Neste processo, em tempo de execução, verificam-se todas as associações e agregações que se relacionam a uma classe ou objeto;
- 8. Identificação de padrões de projeto: por fim, de posse das classes e objetos envolvidos, seus estados, suas colaborações, de forma estática e dinâmica, são identificados alguns padrões de projeto utilizados na literatura.

5.3 Extração Estática de Informações

Um software orientado a objetos, em geral, consiste de definições e de referências como classes, métodos e variáveis. Estas definições incluem referências a outras definições já projetadas, como, por exemplo, classes que recorrem a outras classes ou superclasses; variáveis que fazem referência a uma classe ou a um tipo primitivo, ou ainda, declarações que podem recorrer a métodos para delegar atividades.

Uma ferramenta de referência cruzada descreve estas relações analisando gramaticalmente um ou mais códigos fonte da aplicação, determinando relações de referência entre estas definições. Estas relações, por fim, são apresentadas na saída padrão, por meio de um relatório ou arquivo.

Cada classe, método ou variável possui um identificador. O identificador é chamado de símbolo quando usado em um compilador. Um símbolo, portanto, representa uma entidade específica definida na aplicação. Múltiplos símbolos podem parecer ser o mesmo, mas as definições são baseadas no local onde estes se encontram no arquivo fonte. Portanto, uma ferramenta de referência cruzada precisa manter a localização de todos os símbolos definidos e onde estes símbolos são referenciados.

A ferramenta de referência cruzada, proposta neste trabalho, tem como objetivos analisar gramaticalmente arquivos fonte em Java, nos diretórios especificados pelo projetista, com o objetivo de criar uma tabela de símbolos que contenha uma lista de todas as definições dos arquivos, bem como suas respectivas referências. A Figura 5.2, mostra que cada elemento da tabela de símbolos é representado por uma determinada classe que possui definições a respeito de seu pacote, suas superclasses, métodos e atributos. Cada elemento apresenta o local de definição e referências.

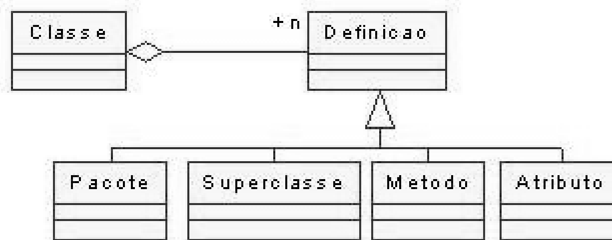

FIGURA 5.2 – Diagrama de Classes – Geração de Referência Cruzada

Existem várias maneiras possíveis de projetar um analisador para construir uma tabela de símbolos e de referências. Considerando que a linguagem Java apresenta, em geral, a definição de várias classes em um arquivo fonte, acredita-se que, para solucionar uma referência a uma classe, pode-se adotar uma das seguintes alternativas:

- *walk, then resolve*: durante a análise gramatical, são registradas as definições na tabela de símbolos e as referências são armazenadas como *strings*. Após a finalização da análise gramatical, todas as definições já foram vistas e pode-se observar a tabela de símbolos solucionando as referências pendentes (GRUNE, 2001).
- *backpatch*: esta técnica utiliza objetos *placeholder* na tabela de símbolos, ou seja, objetos que não possuam um estado definido para um tipo que ainda não foi identificado. Quando o símbolo desconhecido estiver finalmente definido, o *parser* substituirá as referências *placeholder* pela referência à real definição (AHO, 1988).

Backpatch é a técnica mais eficiente, mas acrescenta muita complexidade ao processo. Para simplificar a estrutura da ferramenta, o analisador de referência cruzada descrito no trabalho implementa a primeira estratégia.

5.3.1 *ANother Tool for Language Recognition*

A tradução de linguagens de programação tornou-se uma tarefa de interesse pela comunidade ligada à Ciência da Computação. Enquanto os *parsers* e as ferramentas para linguagens de programação tradicionais, como C e Java, ainda estão sendo construídos ou aprimorados, verifica-se o aumento gradativo do número de idiomas diferentes que estão sendo desenvolvidos para a construção de reconhecedores e tradutores. Projetistas em geral, constroem tradutores para formatos de: banco de dados, arquivos de dados gráficos (*PostScript* e *AutoCAD*), arquivos de processamento de textos (*HTML*), entre outros.

ANTLR generaliza a definição de *scanner* e *parser* em uma abstração simples por meio de uma estrutura gramatical aplicada aos *tokens* de entrada. Portanto, um arquivo de gramática em ANTLR consiste de definições que utilizam uma notação consistente e única em comparação, por exemplo, a outras ferramentas que utilizam expressões regulares para avaliar os símbolos de entrada e gramáticas BNF para descrever a estrutura da linguagem. ANTLR utiliza, portanto, a mesma notação EBNF (BNF - estendida) para descrever análise léxica, sintática e semântica.

Um analisador léxico (freqüentemente chamado scanner) subdivide uma cadeia de caracteres de entrada em um conjunto de símbolos disponíveis no vocabulário. A análise léxica isola os símbolos no fluxo de entrada e determina a sua classe e sua representação (GRUNE, 2001). Estes símbolos, por sua vez, são informados ao *parser,* o qual aplica a estrutura gramatical disponível para reconhecimento destes. A especificação para a estrutura léxica é a mesma utilizada para a estrutura do *parser*. ANTLR permite também a execução de ações durante o reconhecimento dos *tokens*.

Um *parser* é uma ferramenta cujo objetivo é a aplicação de uma estrutura gramatical a um conjunto de símbolos de entrada, no intuito de verificar se estes símbolos estão corretos ou não. ANTLR ajuda o projetista na construção intermediária de árvores de tradução mediante uma gramática de operadores, regras e ações que suportam produzir a tradução. ANTLR possibilita, portanto, construir reconhecedores para *streams* de caracteres, *tokens*, ou nodos de árvores.

5.3.2 Processos do Gerador de Referência Cruzada

O analisador gerado por ANTLR apresenta três processos distintos:

- análise gramatical do código fonte para determinar definições e referências: esta primeira fase tem por finalidade ler o código fonte do diretório especificado, e a partir de então, colecionar informações sobre os seguintes itens: pacotes, classes, interfaces, métodos, etc. Para cada definição encontrada, é criado um novo símbolo. Este símbolo pode caracterizar-se como referência de outros símbolos. Durante a primeira passagem, estas referências, muitas vezes, ainda não estão disponíveis e podem ser definidas depois. Por esta razão, qualquer referência para superclasses, interfaces, tipos de retorno para métodos e parâmetros são armazenados como *placeholders* pelos nomes. Estes *placeholders* são mantidos em uma classe denominada *DummyClass*.
- solução das referências por meio da utilização das informações da tabela de símbolos: depois de analisar o código fonte gramaticalmente, a tabela de símbolos já está pronta com as definições propostas na aplicação. Cada definição pode fazer referência a outras definições (como superclasses, tipos de variáveis e assim por diante). Esta segunda fase tem por função solucionar todas as referências na tabela de símbolos. Muitos símbolos na tabela executam um método chamado *resolveTypes,* que é usado para acertar os tipos referenciados.

- geração de relatório: esta fase verifica as informações na tabela de símbolos e gera um relatório de referência cruzada.

Por fim, para criar o analisador de referência cruzada utilizando a linguagem Java como idioma, optou-se por ANTLR como ferramenta de geração de *parsers*. A estrutura gramatical de ANTLR caracteriza-se como simples e de rápida prototipação.

5.4 Extração Dinâmica de Informações

A extração dinâmica tem por objetivos analisar o comportamento da aplicação em teste, durante tempo de execução, capturando informações a respeito dos estados assumidos pelos objetos.

A ferramenta proposta utiliza como idéia básica a execução de linhas de execução distintas, ou seja, a execução da aplicação original do projetista como uma linha de execução, e a captura das informações, por parte da ferramenta, como outra. Assim, a extração das informações se dá pela comunicação entre as linhas de execução, e faz-se necessário, portanto, diagnosticar um modo de comunicação entre as linhas de execução.

5.4.1 Linhas de Execução

Uma linha de execução (*thread* em Java) é considerada como um fluxo de controle dentro de uma aplicação. Uma linha de execução é semelhante à definição de processos, exceto pelo compartilhamento de parte do mesmo estado, ou seja, múltiplas linhas de execução concorrem no mesmo espaço de endereços (NIEMEYER, 1997).

Uma linha de execução nasce em Java quando se cria uma instância da classe *java.lang.Thread*. O objeto da classe *Thread* representa uma linha de execução no interpretador Java e serve como uma referência para controlar e sincronizar a sua execução. Por meio deste objeto, portanto, pode-se iniciar, suspender ou, até mesmo, terminar a *thread.*

5.4.2 Comunicação entre Linhas de Execução

O padrão de comunicação entre linhas de execução caracteriza uma linha denominada produtora, aquela que gera um fluxo de bytes e uma linha denominada consumidora, que lê e processa este mesmo fluxo de bytes. Se nenhum byte estiver disponível para leitura, a linha de execução consumidora é bloqueada. Se o produtor gerar dados mais rapidamente do que o consumidor pode manipulá-los, então a operação da linha de execução produtora é bloqueada (HORSTMANN, 2001).

A linguagem Java apresenta um conjunto de classes *PipedInputStream* e *PipedOutputStream* para implementar o padrão de comunicação citado. O motivo de usar *pipes* neste trabalho foi o de manter a simplicidade em cada linha de execução. A linha de execução produtora simplesmente envia seus resultados para um fluxo sem se preocupar com estes. A consumidora simplesmente lê os dados de um fluxo, sem se preocupar de onde eles vêm. Portanto, usando *pipes* podem-se conectar várias linhas de execução entre si, sem a preocupação com o sincronismo das linhas de execução.

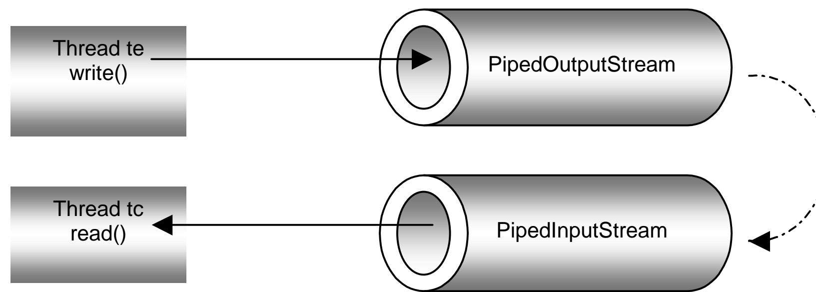

FIGURA 5.3 – *Streams* em *Pipe* (NIEMEYER, 1997)

Conforme a Figura 5.3, *te* representa a *thread* produtora que se responsabiliza pela execução da aplicação do projetista e envia as informações pertinentes via *pipe PipedOutputStream*. A *thread tc* recebe como parâmetro um objeto da classe *PipedInputStream* e, como tal, comporta-se como a *thread* consumidora lendo as informações recebidas pelo *pipe PipedInputStream*.

Quando as *threads* são postas em execução por meio do envio da mensagem *start*, passa-se efetivamente à execução balanceada mediante a produção e consumo das informações geradas.

5.4.3 Clonagem de Objetos

Cada vez que a linha de execução produtora enviar um objeto para saída estará, automaticamente, gerando uma cópia deste por meio do mecanismo de clonagem.

Ao clonar um objeto existente, obtém-se uma cópia que reflete o estado atual do objeto. Os dois objetos passam a existir independentemente, considerando que, com o passar do tempo, eles podem divergir quanto a seus estados (HORSTMANN, 2000).

Porém, o método *clone* é um método protegido da classe *Object,* e isto significa que o código não pode simplesmente chamá-lo. A classe *Object* não sabe nada sobre o objeto e, portanto, poderá apenas fazer uma cópia bit a bit. Se todos os atributos do objeto forem números ou outros tipos básicos, uma cópia só de bits seria suficiente, ou seja, o objeto clone seria outro objeto com os mesmos tipos e campos base. Mas se o objeto contiver referências a outros objetos em um ou mais de seus atributos, então uma cópia de bits irá conter exatamente cópias exatas dos campos de referência também, fazendo com que os objetos original e clonado compartilhem algumas informações.

A opção adotada neste trabalho foi implementar a classe *SerialCloneable*. A classe implementa facilidades das classes *Cloneable* e *Serializable*. Justifica-se, portanto, a modificação das superclasses na aplicação teste, automaticamente, para que estas representem classes de *SerialCloneable*.

A classe *SerialCloneable* redefine, portanto, o método clone com o modificador de acesso público. Utilizando os recursos de serialização, a classe *SerialCloneable* faz uso da classe *ByteArrayOutputStream* para objetos de *ObjectOutputStream* (fluxo de saída) e *ByteArrayInputStream* para objetos de *ObjectInputStream* (fluxo de entrada), para criar um fluxo de objetos. Por meio da serialização, é possível, portanto, salvar e restaurar, em um fluxo de objetos, todas as informações desejadas. Um fluxo de objetos apresenta uma descrição detalhada de todos os objetos que ele contém, com informações suficientes para permitir a reconstrução de objetos, referências e objetos associados.

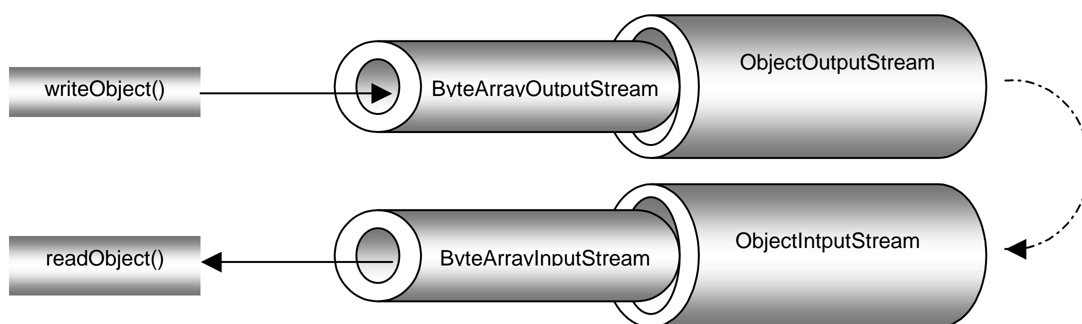

FIGURA 5.4 – Serialização e Clonagem

A serialização de objetos salva as informações dos objetos em um formato particular, por meio da utilização do método *writeObject.* Já o método *readObject* lê a classe do objeto, a assinatura da classe e os valores dos atributos, fazendo também a desserialização para permitir que múltiplas referências de objetos possam ser recuperadas. A Figura 5.5 demonstra o envio da mensagem *clone* para um objeto utilizado como exemplo, denominado *listaMonetario*, e a execução das respectivas mensagens *writeObject* e *readObject.*

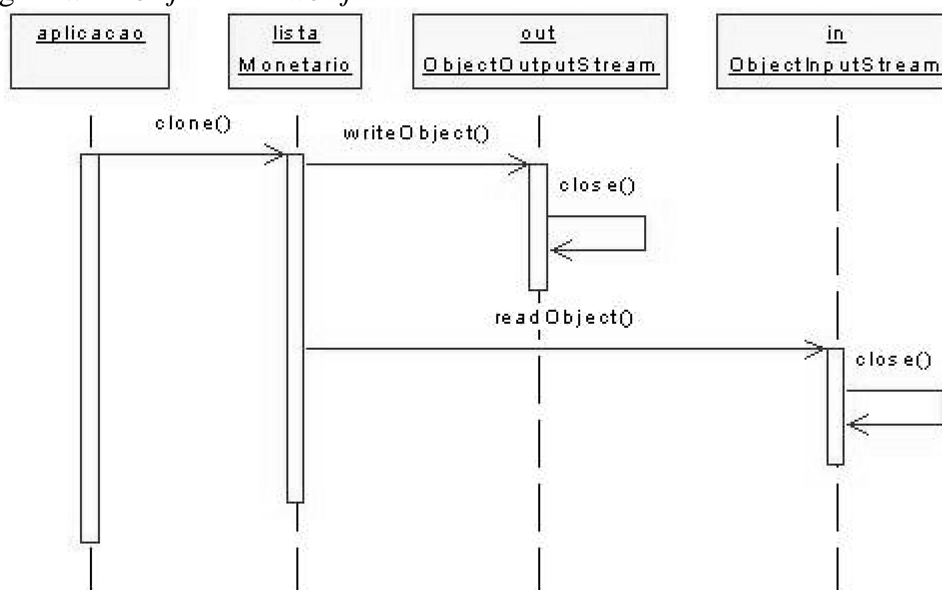

FIGURA 5.5 – Diagrama de Mensagens – *clone()*

Na classe de saída *ObjetoOutput,* o clone do objeto é recebido pelo método *writeObjeto* que instancia um novo objeto, a partir da classe *objetoTransporte,* e passa estas informações como parâmetros. Na verdade, o objeto que circulará pelo *pipe* e será recebido pela *thread* de consumo é da classe *ObjetoTransporte*.

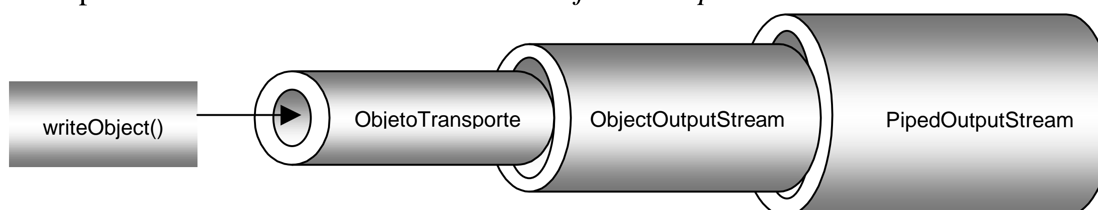

FIGURA 5.6 – Fluxo de Saída

A classe *ObjetoTransporte* simplesmente encapsula uma cópia do objeto a trafegar no *pipe* e sua descrição. Portanto, um objeto desta classe será recebido na *thread* de consumo.

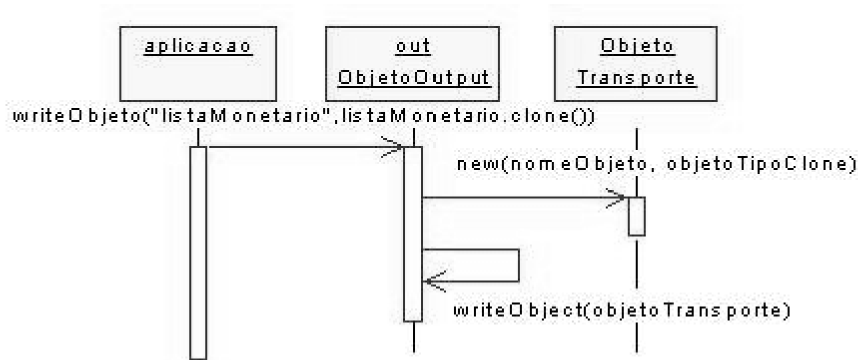

FIGURA 5.7 – Diagrama de Mensagens – *writeObjeto()*

A linha de execução responsável pela ferramenta de inspeção é considerada a linha consumidora. Identificada como *ThreadConsumo,* esta linha recebe o fluxo de objetos por intermédio do pipe. Por meio do método *readObject,* é disponibilizado um objeto da classe *Object*. Este objeto na verdade é um objeto da classe *ObjetoTransporte,* recebido pelo fluxo de entrada.

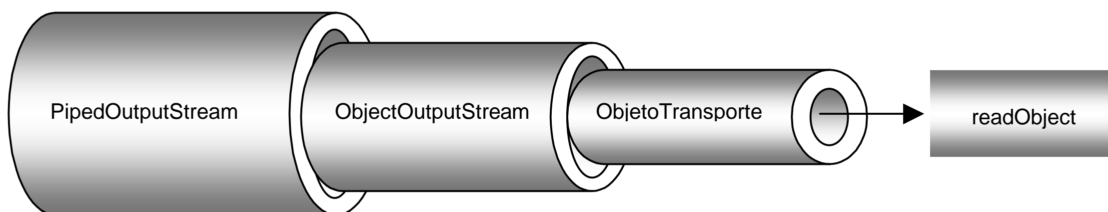

FIGURA 5.8 – Fluxo de Entrada

A linha de execução verifica se o estado atual, capturado na linha de execução do projetista, já existe na coleção de estados disponíveis. Esta verificação se dá por meio de um teste junto à coleção de estados, mediante uma comparação do estado identificado com os estados previamente armazenados.

A ferramenta de inspeção, portanto, disponibiliza o vetor denominado *VetorDeInformações,* que armazena objetos da classe *EstadosDoObjeto*. Para cada estado de objeto capturado (*ObjetoTransporte*) é feita uma comparação do nome deste objeto com os objetos armazenados no *VetorDeInformações*. Se este identificador não existe é criado um objeto da classe *EstadosDoObjeto* com as informações de definição e estado inicial. Se, porventura, este objeto for considerado como existente, será verificado o estado capturado com os estados armazenados para identificar se houve ou não modificação no estado do objeto, com relação a seu estado anterior.

A classe *EstadosDoObjeto* possui o método *addEstado,* que recebe como entrada um objeto o qual representa o estado atual capturado na aplicação do projetista. De posse desta informação, o método verifica qual o último estado disponível na lista de estados para compará-lo como o objeto recebido. Utilizando, novamente, os recursos de serialização, o método *addEstado* faz uso da classe *ByteArrayOutputStream* para objetos de *ObjectOutputStream* (fluxo de saída). Os dois objetos, estado atual (recebido) e estado anterior (último armazenado), são gerados em um fluxo de saída. Por meio da serialização, portanto, é possível salvar em um fluxo de objetos todas as informações

desejadas para os dois objetos. Cada fluxo representa uma descrição detalhada de todos os objetos/referências que cada objeto contém.

O primeiro teste é feito com relação ao tamanho do fluxo, para verificar se estes fluxos possuem o mesmo número de elementos. Caso possuam o mesmo número de informações, passa-se para a comparação elementar, ou seja, os dois fluxos são convertidos para vetores de bytes (*toByteArray*), e cada elemento do vetor de bytes de um objeto é comparado a um elemento do vetor de bytes do outro objeto. Se todas as informações forem coincidentes, acredita-se que o estado investigado não tenha necessidade de ser armazenado porque representa o estado anterior previamente armazenado. Caso contrário, faz-se o armazenamento deste objeto no vetor de estados.

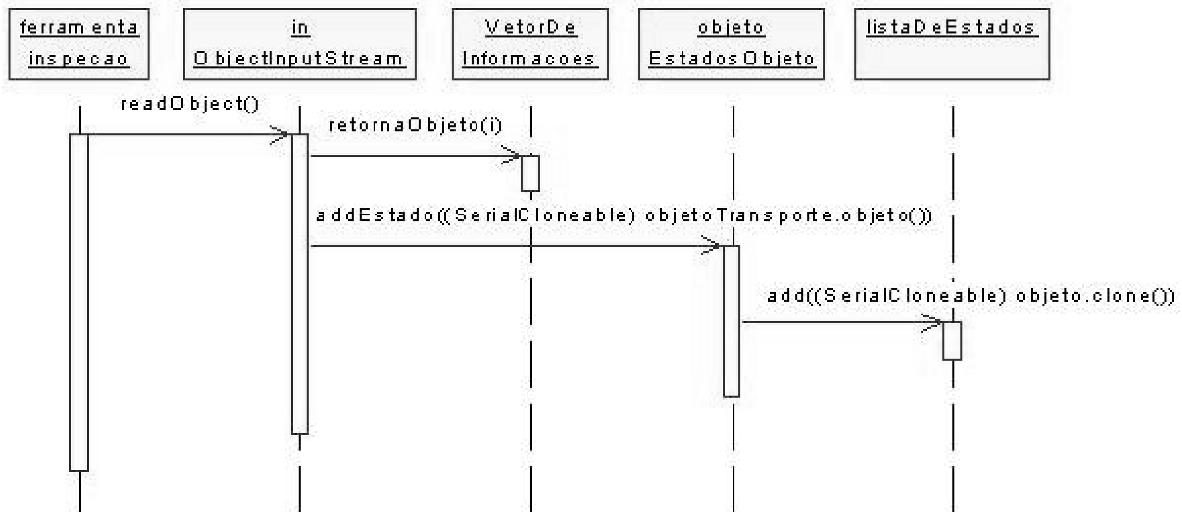

FIGURA 5.9 – Diagrama de Mensagens – *readObject()*

Por fim, para cada objeto capturado na linha de execução do projetista, a linha de execução da ferramenta procede à identificação e avaliação da existência deste na coleção de estados que denota a evolução de cada objeto na aplicação. Assim, as linhas de execução mantêm a comunicação até o momento em que não existirem mais objetos capturados, o que simboliza o fim da execução da aplicação do projetista.

5.4.4 Modelo da Extração Dinâmica

A ferramenta de inspeção adota o padrão de comunicação entre linhas de execução, o qual caracteriza uma linha denominada produtora (aquela que gera um fluxo de bytes) e uma linha denominada consumidora (que lê e processa este mesmo fluxo de bytes). A Figura 5.10 apresenta o diagrama, simplificado, de classes da estrutura de linhas de execução propostas na ferramenta.

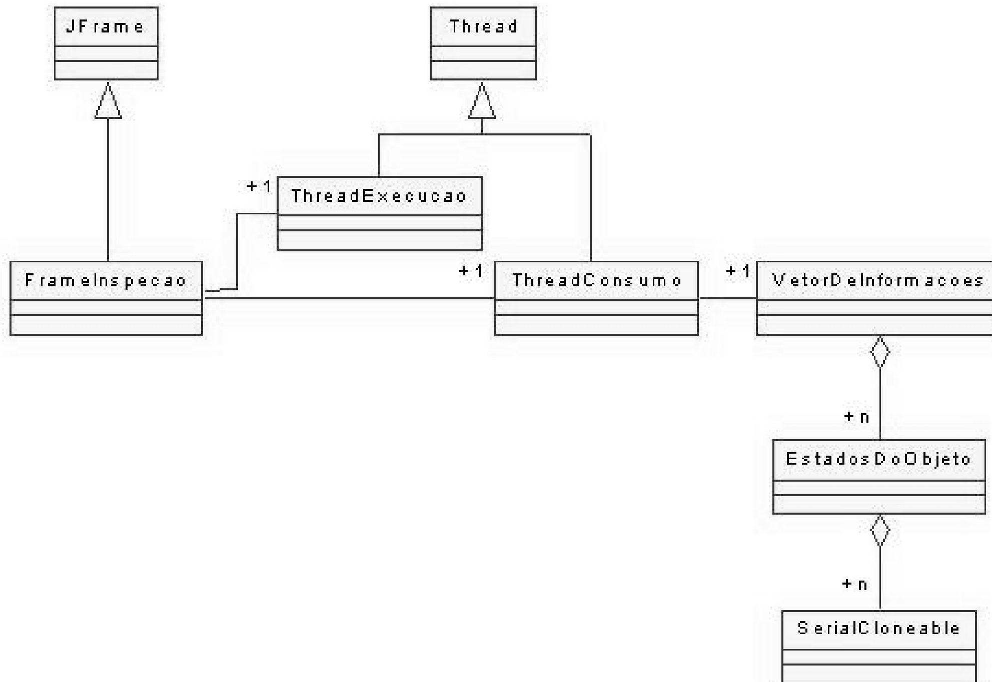

FIGURA 5.10 – Diagrama de Classes – Extração de Informações

A classe *ThreadConsumo* é responsável pelo consumo das informações fornecidas pela classe *ThreadExecucao*. Cada objeto avaliado na linha de execução do projetista é enviado pelos mecanismos de *pipe* para a linha de consumo. Optou-se pela utilização destes mecanismos levando em consideração o sincronismo e segurança para envio e consumo de informações. À medida que o *buffer* interno do *pipe* for preenchido, a linha de execução é bloqueada e espera até que haja espaço disponível. Em contrapartida, se o *pipe* estiver vazio, o consumidor é bloqueado e espera até que os dados estejam disponíveis.

A armazenagem dos objetos, na linha de consumo, é feita pela classe *VetorDeInformacoes*, a qual representa uma instância da classe *Vector* e, por sua vez, armazena os diferentes objetos avaliados na aplicação. A classe *VetorDeInformacoes* é composta pela classe *EstadoDoObjeto,* que disponibiliza os estados possíveis para cada objeto (caracterizados por objetos da classe *SerialCloneable*).

É importante salientar que os objetos armazenados em *EstadosDoObjeto* representam clones dos objetos utilizados na aplicação devido à necessidade de manter todos os estados avaliados, para que se possam verificar as modificações existentes na aplicação original.

5.5 Reflexão Computacional

A Figura 5.11 apresenta o diagrama de classes simplificado do modelo estrutural do mecanismo de interface para reflexão sobre o código inspecionado. A classe *ApresentacaoFrame* é responsável pela execução e apresentação das interfaces de consulta às informações inspecionadas.

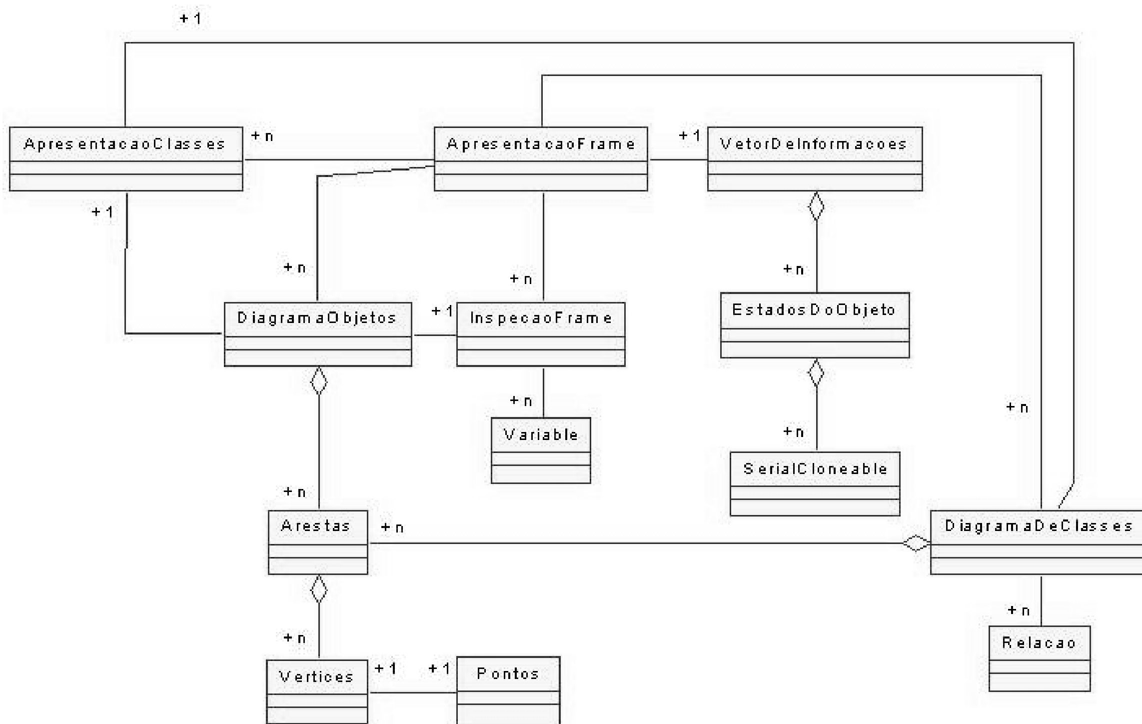

FIGURA 5.11 – Diagrama de Classes - Reflexão

A classe *ApresentacaoFrame* relaciona-se com *VetorDeInformacoes,* que dispõe de todos os estados capturados para os diferentes objetos inspecionados. A partir de então, estas informações são passadas para *ApresentacaoClasses* (responsável pela demonstração das classes e seus respectivos atributos em modo texto), *InspeçãoFrame* (responsável pela demonstração dos objetos e suas colaborações em modo texto), *DiagramaObjetos* (responsável pela demonstração dos objetos e suas colaborações em modo gráfico) e *DiagramaDeClasses* (responsável pela demonstração das classes e suas colaborações em modo gráfico).

É importante salientar que as classes que trabalham com a apresentação gráfica das informações o fazem a partir de um conjunto de arestas e vértices.

5.5.1 Modelo da Reflexão Computacional

Enquanto uma aplicação está em execução, a API Java mantém o que se chama identificação do tipo em tempo de execução de todos os objetos. Esta informação acompanha a classe à qual cada objeto pertence e é utilizada pela linguagem para selecionar os métodos corretos a executar em tempo de execução.

Portanto, para acessar estas informações, a linguagem Java provê uma classe especial denominada *Class*. Esta classe fornece um conjunto de ferramentas muito elaboradas para escrever aplicações que manipulam o código Java dinamicamente, disponibilizando a uma determinada aplicação à análise de seus recursos, caracterizando uma arquitetura reflexiva. O pacote que acrescenta estas funcionalidades à linguagem Java é denominado *java.lang.reflect.*

O mecanismo de reflexão permite (HORSTMANN, 2000):

• analisar os recursos das classes em tempo de execução – por meio do mecanismo de reflexão é permitido que se examine a estrutura de uma classe. O pacote *java.lang.reflect* disponibiliza, entre outras, três classes: *Field*, *Method* e *Constructor* que descrevem, respectivamente, os atributos, os métodos e os construtores de uma

classe. As três classes possuem o método *getName* que retorna o nome do item avaliado. A classe *Field* possui um método denominado *getType* que retorna um objeto do tipo *Class*, que define o tipo do atributo. Já as classes *Method* e *Constructor* possuem métodos para informar o tipo de retorno e os tipos dos parâmetros utilizados nos métodos avaliados. Todas as três classes possuem o método *getModifiers,* que retorna um tipo inteiro com vários bits ativados e desativados, os quais descrevem os modificadores tais como *public* e *static* (MCCLUSKEY, 1999);

- inspecionar objetos em tempo de execução - o mecanismo de reflexão computacional permite examinar os atributos de um objeto que não são determinados em tempo de compilação. O método responsável por este procedimento é o método *get,* da classe *Field.* Considerando *f* um objeto do tipo *Field* (um objeto obtido por meio de *getDeclaredFields*) e *obj* um objeto da classe da qual *f* é um atributo, então *f.get(obj)* retorna um objeto cujo conteúdo é o valor atual do atributo de *obj* (HORSTMANN, 2000). Mas nem sempre esta afirmativa é correta, pois em muitos casos o atributo pode ser definido como privado, e o método *get* tende a lançar uma exceção denominada *IllegalAccessException.* O mecanismo de segurança da linguagem Java permite que sejam identificados os atributos de um objeto, mas não permite que sejam lidos os seus conteúdos, a menos que se tenha permissão de acesso para tal. O comportamento padrão do mecanismo de reflexão é o respeito ao controle de acesso. Na ferramenta de inspeção, faz-se necessário mudar este controle e, como tal, utiliza-se o método *setAccessible* disponível para objetos: *Field*, *Method* e *Constructor.* O método *setAccessible* é disponível na classe *accessibleObject,* que representa uma superclasse comum de *Field*, *Method* e *Constructor*;

- implementar código genérico de manipulação de arrays e coleções - a API de reflexão permite criar e inspecionar arrays de tipos básicos usando a classe *Array* disponível no pacote *reflect.* O comprimento é obtido chamando-se *Array.getLength(a).* O método estático *getLength* retorna o tamanho de qualquer *array.* Portanto, para obter os elementos de um componente do tipo array, primeiro faz-se necessário obter o objeto *a*, confirmar se realmente é um *array* e, então, propor uma repetição até o número máximo de elementos dados por *getLength.* Nesta repetição para inspecionar cada elemento, faz-se uso do método *get* identificando qual elemento do *array* pretende-se inspecionar. Mas, a interface fundamental para classes de coleção em Java é a interface *Collection.* Esta possui dois métodos fundamentais: *add* e *iterator.* O método *add* é responsável pela inclusão de objetos em uma coleção. Já o método *iterator* retorna um objeto que implementa a interface *Iterator.* A interface *Iterator* possui três métodos fundamentais: *next*, *hasnext* e *remove* (HORSTMANN, 2001). Portanto, chamando repetidamente o método *next,* pode-se examinar uma coleção considerando elemento a elemento. Caso, se chegue ao final da coleção, o método *next* lançará uma exceção *NoSuchElementException.* Para evitar tal problema, é importante invocar o método *hasNext* antes de *next.* O método *hasNext* retorna *true* se o objeto iterador ainda tiver mais elementos para visitar. Caso seja necessário inspecionar toda uma coleção, deve-se solicitar um iterador com uma repetição a invocação do método *next* enquanto *hasNext* retornar *true.* Deve-se considerar, conforme figura a seguir, que os iteradores na linguagem Java são entre elementos, ou seja, quando executado o método *next* o iterador pula para o próximo elemento e retorna uma referência para o elemento que acabou de passar.

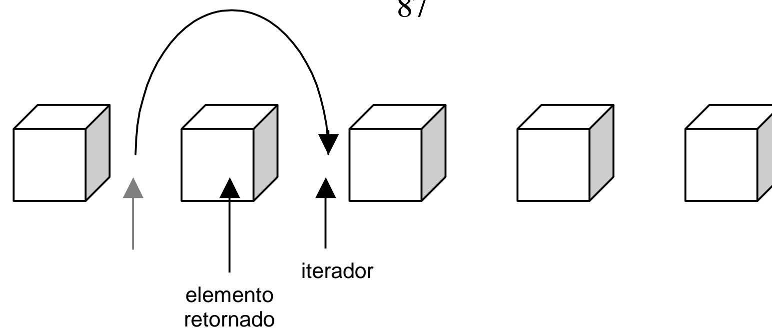

FIGURA 5.12 – Avanço em um Iterador (HORSTMANN, 2001)

Portanto, como as interfaces *Collection* e *Iterator* são genéricas, podem-se escrever métodos utilitários para operarem sobre qualquer tipo de coleção.

• aproveitar os objetos *Method* que funcionam exatamente como ponteiros de funções em linguagens com C++.

5.6 Identificação de Padrões de Projeto

A ferramenta proposta vale-se da procura dos mecanismos de associação ou agregação entre classes e objetos por meio da representação de atributos, pela identificação de interfaces como *Collection* e *Iterator* (mediante coleções ou *arrays*). Assim, a inspeção dos atributos pode ser feita com o auxílio de um iterador ou diretamente, acessando seus elementos.

A ferramenta de inspeção faz a verificação das colaborações para todos os estados que o objeto apresentar em tempo de execução, sendo possível a avaliação da evolução de seus estados. Assim pode-se determinar se as estruturas estão sendo bem utilizadas ou, até mesmo, subutilizadas. A identificação nesta fase conta com o auxílio do processo de reflexão computacional citado na seção anterior.

Considerando-se as classes e objetos envolvidos, seus estados, suas colaborações, de forma estática e dinâmica, esta fase tem por objetivo identificar alguns padrões de projeto utilizados na literatura.

A identificação de padrões leva em consideração a essência dos padrões comentada no capítulo 4. Para cada aplicação, são investigadas as evidências que levam a crer na existência da utilização dos padrões no comportamento da aplicação. Em cada caso, há um conjunto de características específicas a serem avaliadas. Dependendo do tipo do padrão, a verificação das colaborações caracteriza-se como de maior importância, para outros a troca de estados define a estrutura básica ou, ainda, em alguns casos, o mais importante para a avaliação são as modificações de estados.

Tomando como base o padrão *Composite*, o qual será abordado no próximo capítulo como exemplo, faz-se necessário avaliar todos os estados do objeto em estudo, na procura de relacionamentos que venham a diagnosticar a existência do padrão. Neste caso, todos os atributos são avaliados e aqueles que pertencem à mesma hierarquia de herança podem caracterizar objetos *leaf,* ou até mesmo outros objetos *Composite*. Para certificar se os objetos da relação realmente caracterizam-se como objetos *Composite,* faz-se a procura pelos atributos deste novo objeto, identificando as suas colaborações. Neste sentido, por meio de um procedimento de pesquisa recursiva é necessário avaliar estas colaborações em toda a estrutura.

Para cada comparação prevista, verifica-se a utilização de métodos das classes avaliadas em relação à superclasse. Outro ponto importante, no caso, é o diagnóstico da utilização ou subutilização do padrão de acordo com as suas características básicas, ou seja, em algumas situações o exemplo pode demonstrar uma conexão recursiva, ou seja,

relacionar-se com objetos do mesmo tipo. Para isso são avaliados todos os estados do objeto e, dentro de cada estado, todas as suas colaborações.

Em função da apresentação, no capítulo 4, das regras para demonstrar a identificação e classificação dos padrões de projeto, será omitida, neste capítulo, uma descrição a respeito.

5.7 Conclusões

Este capítulo tem por finalidade apresentar, resumidamente, os processos de extração de informações que compõem a ferramenta de inspeção de aplicações Java. Inicialmente foi abordado um resumo das atividades implementadas na ferramenta.

O objetivo da ferramenta é disponibilizar dois grupos de informações: estáticas e dinâmicas. Para compor as informações referentes ao modelo estático, a aplicação teste do projetista é submetida ao processo de referência cruzada o qual extrai um relatório de informações a respeito das entidades utilizadas na aplicação. Esse processo de referência cruzada é construído com o apoio da ferramenta ANTLR, *ANother Tool for Language Recognition,* que dispõe de um *framework* para a construção de tradutores a partir de gramáticas predefinidas. O relatório de referência cruzada é formado pelo conjunto de classes identificadas na aplicação. Para cada classe são caracterizados seus métodos, atributos, superclasses, entre outros.

Para representar as informações do modelo dinâmico, faz-se a execução da aplicação teste em uma linha de execução separada, considerada produtora, e a ferramenta é executada em uma outra linha de execução, denominada consumidora. O fluxo de informações entre as linhas produtora e consumidora é feito através de *pipes*. Portanto, cópias dos estados dos objetos avaliados são realizadas de uma linha para a outra.

Sobre os objetos avaliados, aplica-se o processo de reflexão computacional o qual extrai informações sobre as classes, atributos, métodos, entre outras. A partir do processo de reflexão computacional, são verificadas as colaborações levando em consideração todos os estados pelos quais os objetos passam durante execução da aplicação. Considerando-se estas informações, verificam-se as evidências que levam a definir a existência dos padrões na aplicação inspecionada. O termo essência, definido no capítulo 4, vem propor as características básicas a serem encontradas na aplicação. Caso estas características venham a ser atendidas, na quase totalidade, verificam-se fortes indícios da existência dos padrões estudados.

No próximo capítulo será detalhada a ferramenta de inspeção construída, com a definição de seu modelo físico. Após, faz-se a apresentação de um exemplo bem detalhado, levando em consideração todos os mecanismos aplicados pela ferramenta de inspeção.

6 Demonstração da Ferramenta

Uma ferramenta projetada para investigar a essência dos padrões de projeto deve, como mencionado no capítulo 4, investigar tanto o modelo estático como dinâmico da aplicação construída. O leitor poderia argumentar que, para a investigação em aplicações Java, seria necessário apenas avaliar o modelo estático, levando em consideração que, para alguns padrões, observando-se apenas estas características seria satisfatório, principalmente porque se trata de uma linguagem fortemente tipada. Os trabalhos citados no capítulo 3, como os de Krämer (1996), Bansyia (1998), Seeman (1998) e Guéhéneuc (2001), possuem o foco principal no modelo estático da aplicação cuja observação faz-se a partir do código fonte.

Mas a realidade desta tese é caracterizar uma ferramenta que possa diagnosticar se o padrão se faz bem utilizado dentro da sua estrutura, ou se está ocioso em sua capacidade de trabalho. Levando em consideração tal argumento, é importante avaliar o modelo dinâmico que fará uma inspeção durante o ciclo de vida dos objetos envolvidos. Portanto, a verificação das colaborações não será efetuada sobre o modelo estático semelhante aos trabalhos descritos, mas estas serão avaliadas sobre o modelo dinâmico. Os trabalhos citados forneceram fortes subsídios para o desenvolvimento da ferramenta e fortaleceram a idéia da necessidade de trabalhar sobre o modelo dinâmico. A proposta visa a atender, justamente, a identificação da essência dos padrões a partir das aplicações em tempo de execução. O trabalho "*Heuristics for Automatic Detection of Design Patterns in Object-Oriented Software*" (FREITAS, 2000D) justifica a necessidade de observar informações além do modelo estrutural, levando em consideração a execução da aplicação.

A ferramenta proposta visa, portanto, a encontrar as informações descritas como essência dos padrões por meio de uma inspeção nas aplicações. A inspeção consiste em uma verificação tanto na dimensão estrutural (no código) como na dimensão dinâmica (execução da aplicação).

6.1 Geração do Arquivo Fonte Modificado

Um ponto a ser observado pela ferramenta é a não-possibilidade de prever qual classe *main* será, exatamente, instanciada até que se passe a informação para tal. Em uma situação como esta, é importante poder criar um objeto por algum identificador mediante o nome da classe ao qual pertence dinamicamente. Esta é um das habilidades que a API *Reflection* de Java disponibiliza.

A ferramenta processará a aplicação a ser analisada levando em consideração a instanciação a partir da classe abstrata, predefinida, chamada *ProcessadorMensagem*. Considere-se, por exemplo, o fragmento abaixo, de uma aplicação a ser inspecionada.

```
class Exemplo {
        public static void main(String[] args) { ...
        }
}
```

A aplicação do projetista é modificada com o objetivo de representar uma classe que herda propriedades da superclasse *ProcessadorMensagem*. A superclasse *ProcessadorMensagem* disponibiliza o método *runProcessing* que será ativado posteriormente. O objetivo é permitir à ferramenta executar a aplicação do projetista por uma chamada ao método *runProcessing* (SINGH, 2001), a seguir identificado.

```
public  class Exemplo extends ProcessadorMensagem  {
        public void  runProcessing(ObjetoOutput out)    { ...
        }
}
```

Para a ferramenta de inspeção ativar a execução da aplicação em análise, é criada uma instância da classe *ProcessadorMensagem* a partir da seleção do arquivo de entrada a ser inspecionado. Em um segundo momento, é enviada a mensagem *runProcessing* para o objeto instanciado, que invoca a execução da aplicação transformada do projetista. Portanto, a transformação exigida para a aplicação do projetista é a modificação do método *main* para o método *runProcessing.* Além disso a classe principal deverá caracterizar o mecanismo de herança da superclasse *ProcessadorMensagem.*

Outra alteração proposta pela ferramenta é determinar que as classes da aplicação possam herdar características da classe *SerialCloneable* (classe responsável pela serialização e clonagem de objetos). Este mecanismo faz-se necessário para que os objetos, em tempo de execução, possam ser copiados para posterior processo de inspeção.

Este trabalho, em sua proposta inicial, admite que alterar a aplicação do projetista venha a aumentar a complexidade da aplicação, e este fator pode apresentar efeitos colaterais em tempo de execução. Acredita-se que se possam encontrar meios mais eficientes para executar tal processo, mas, até o presente momento, testes realizados não caracterizaram nenhum problema oriundo dessas alterações.

Um outro ponto a ser levado em consideração é a avaliação de objetos, somente no método *main* da aplicação. Optou-se por um modelo mais simples, que possa demonstrar as funcionalidades da ferramenta. Tem-se como meta estender essa avaliação para partes da aplicação à medida que a ferramenta possa garantir melhores índices de robutez.

Resumidamente, o processo de transformação utiliza como entradas o arquivo código Java, representando a aplicação e o relatório de referência cruzada, vindo a gerar como saída a aplicação em análise modificada.

6.2 Interfaces e Execução

Estas seções têm por objetivos demonstrar a execução da ferramenta de inspeção, levando em consideração uma pequena aplicação utilizada como exemplo. A aplicação será, portanto, detalhada, no intuito de possibilitar ao leitor associar as características propostas a esse exemplo.

Após o detalhamento da aplicação, serão identificadas as características e interfaces sobre os processos de:

- referência cruzada;
- execução e extração de informações;
- reflexão computacional;
- identificação de colaborações e padrões.

6.2.1 Interface de Apresentação

A interface inicial apresentada pela ferramenta é demonstrada na Figura 6.1 e caracteriza um menu de seleção o qual permite ao projetista a escolha da aplicação fonte que será investigada. Portanto, por meio da opção *Abrir*, faz-se chamada à interface de seleção de aplicações. Como qualquer aplicativo, o projetista pode deslocar-se pelos diversos subdiretórios disponíveis para visualização e procurar a aplicação que julgar conveniente para avaliação.

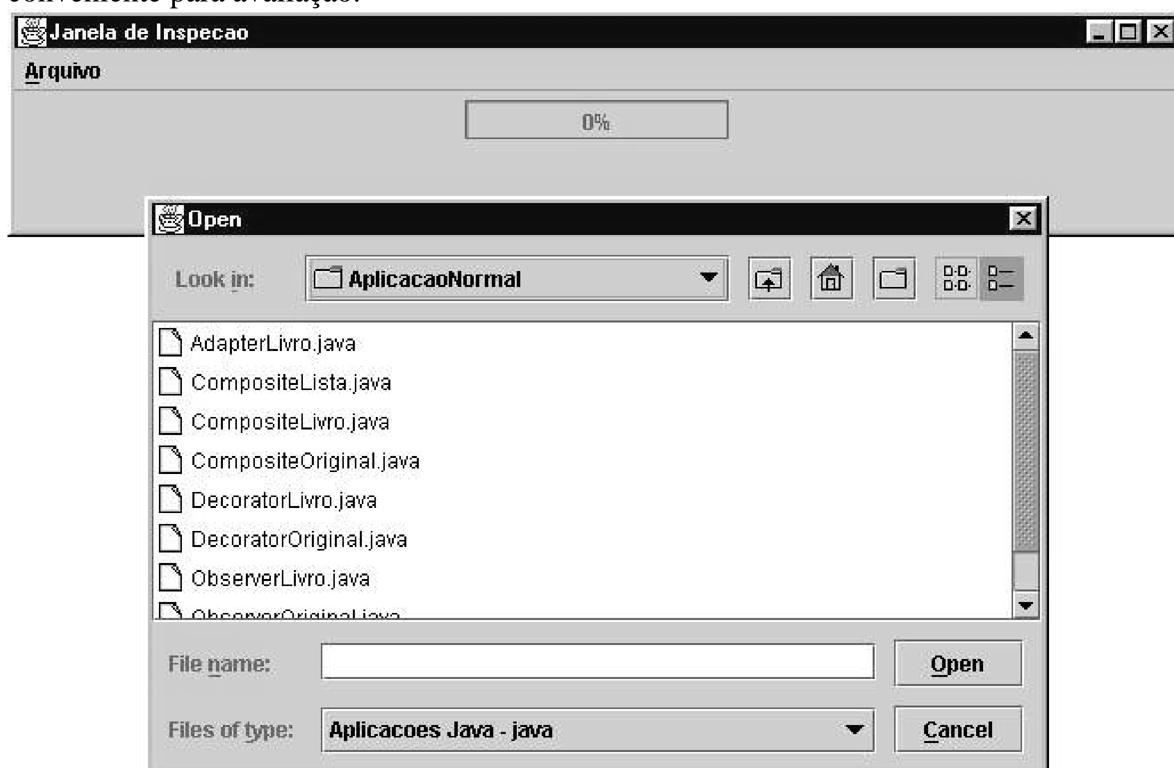

FIGURA 6.1 – Interface de Seleção de Aplicações

A interface de seleção caracteriza uma janela padrão, disponível na classe *JFileChooser*, a qual permite exibir uma caixa de diálogo de arquivo semelhante à maioria dos aplicativos nativos. Os diálogos *JFileChooser* são modais e permitem, portanto, a seleção da unidade de disco, diretórios, subdiretórios e arquivos (HORSTMANN, 2000). O projetista deve, diante desta janela, encontrar o arquivo fonte que será analisado.

Estas seções apresentarão todos os passos aos quais a aplicação escolhida é submetida e os respectivos resultados obtidos em cada uma dessas situações. Portanto, para que as apresentações se tornem menos abstratas optou-se pela utilização de um pequeno exemplo que será investigado.

6.2.2 O Exemplo Proposto

Conforme Figura 6.2, o exemplo apresenta um pequeno conjunto de classes e seus relacionamentos. A hierarquia de classes representa o ativo patrimonial de uma determinada pessoa. *Ativo* implementa a classe que oferece a interface para a estrutura, *AtivoComposite* define o comportamento para os objetos compostos e *Garantia,* para objetos primitivos (ALPERT, 1998).

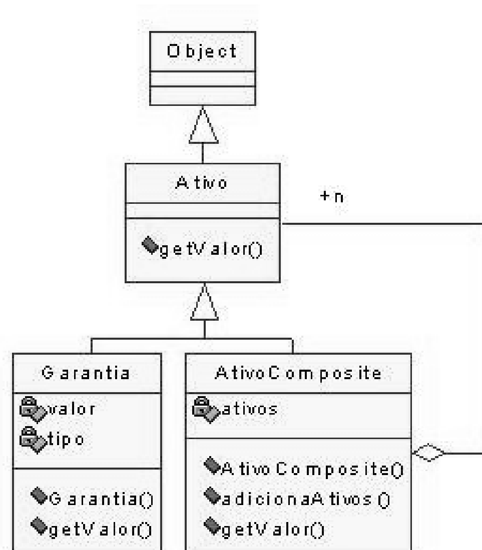

FIGURA 6.2 - Diagrama de Classes da Aplicação (ALPERT, 1998)

Na aplicação faz-se a criação de um objeto *AtivoComposite*, denominado *listaMonetario,* que inicializa o atributo *ativos* (do tipo *Vector*) por meio da mensagem *new*. Em seguida, é enviada a mensagem *adicionaAtivos* para *listaMonetario,* que procede à inserção dos elementos primitivos (*Garantia*) ao vetor. Quatro elementos: *aplicacoes*, *conta corrente*, *poupanca* e *acoes* são inseridos no vetor.

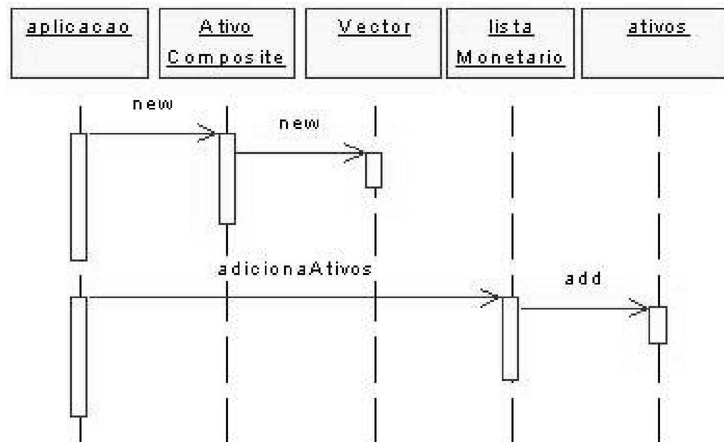

FIGURA 6.3 – Diagrama de Mensagens

A estrutura utilizada para a criação de *listaBens* é a mesma de *listaMonetario*. O mecanismo de inserção em *AtivoComposite* permite a inserção de *listaMonetario* como elemento do vetor *listaBens*. A estrutura a seguir demonstra uma árvore de objetos.

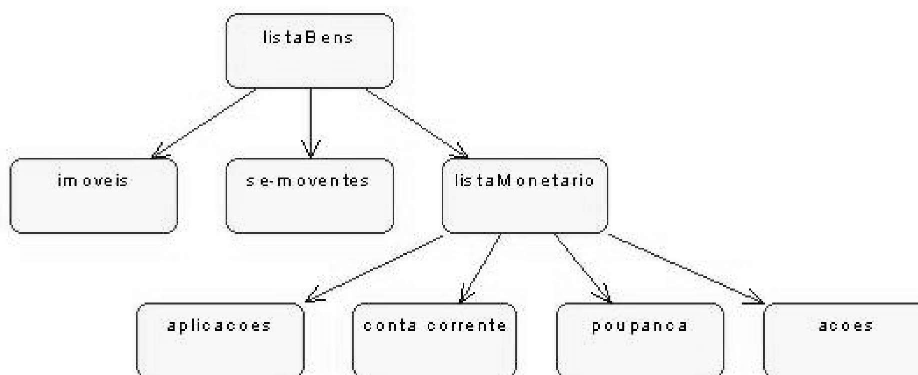

FIGURA 6.4 – Diagrama de Objetos

Ao final, a aplicação procede à pesquisa de uma informação na árvore, utilizando o método *getValor* que executa o somatório junto aos nodos primitivos ou junto a possíveis vetores na estrutura, de maneira recursiva. O resultado do somatório é representado pela soma do atributo valor de todos os objetos primitivos.

6.2.3 Extração Estática de Informações

Conforme já descrito anteriormente, o primeiro processo a ser realizado sobre a aplicação do projetista é o de referência cruzada. O código fonte escolhido é submetido, então, ao processo. O projetista pode visualizar em uma janela texto a execução do processo onde são apresentadas as classes investigadas no momento.

O relatório de referência cruzada é salvo em arquivo texto, para posterior utilização pelo processo de modificação da aplicação do usuário. Conforme descrição a seguir, o relatório é formado pelo conjunto de classes identificadas na aplicação. Para cada classe são caracterizados seus métodos, atributos, procedimentos, entre outros.

A descrição a seguir demonstra, por exemplo, que a classe *AtivoComposite* é definida na linha 22 e referenciada na linha 47 (duas vezes). Identifica que *AtivoComposite* é subclasse da classe *Ativo,* referenciada na linha 4. Os seguintes métodos são identificados: construtor na linha 23, *getValor* na linha 31 e *adicionaAtivos* na linha 27. Para cada método são referenciados os atributos utilizados, como, por exemplo, as variáveis *i* e *valor* do método *getValor*.

E assim para todas as classes, conforme mostrado a seguir, são identificadas todas as entidades investigadas a partir da aplicação fonte.

```
Relatorio de Referencia Cruzada
Pacote: java.text
  Referenciada(o): 1(sem definicoes de compilacao)
Pacote: ~default~
    Classe AtivoComposite 22 Referenciada(o): 47 47
      Superclasse: Ativo 4
      Classes/pacotes importados:
        Vector
        java.text
      Metodo ~construtor~ 23
          Variavel elementos 23  Tipo int Referenciada(o): 24
      Variavel ativos 41  Tipo Vector Referenciada(o): 24
      Metodo getValor 31
        Tipo retorno: float
          Bloco: 34
            Variavel objetoTempo 35  Tipo Ativo 4
          Variavel i 34  Tipo int Referenciada(o): 35 34 34
          Variavel valor 32  Tipo float Referenciada(o): 36 36 38
      Metodo adicionaAtivos 27
        Tipo retorno: void
          Variavel objetoAtivo 27  Tipo Ativo 4 Referenciada(o): 28
    Classe Garantia 8
      Superclasse: Ativo 4
      Classes/pacotes importados:
        Vector
        java.text
      Metodo ~construtor~ 9
          Variavel t 9  Tipo String Referenciada(o): 10
          Variavel v 9  Tipo float Referenciada(o): 11
      Variavel tipo 19  Tipo String Referenciada(o): 10
      Variavel valor 18  Tipo float Referenciada(o): 11 15
      Metodo getValor 14
        Tipo retorno: float
    Classe CompositeOriginal 44
      Classes/pacotes importados:
        Vector
        java.text
      Metodo main 45
        Tipo retorno: void
          Variavel args 45  Tipo String
          Variavel listaBens 47 Tipo AtivoComposite 22 Referenciada(o):55
          Variavel listaMonetario 47  Tipo AtivoComposite 22 Referenciada(o): 50 58
    Classe Ativo 4 Referenciada(o): 35 35 27 22 8
      Classes Derivadas:
        AtivoComposite
        Garantia
      Classes/pacotes importados:
        Vector
        java.text
      Metodo getValor 5 Referenciada(o): 36
        Tipo retorno: float
```

A partir do relatório de referência cruzada, faz-se a apresentação destas informações em uma janela de visualização para que o projetista possa inspecionar a aplicação fonte. A seção a seguir apresenta detalhes desta janela.

6.2.4 *Browser* de Visualização

A janela de visualização, denominada interface de visualização de referência cruzada, utiliza a técnica MVC (*Model-View-Controller*) para carregar e gerenciar o arquivo de entrada, representado pelo relatório de referência cruzada. MVC é uma técnica de desenvolvimento que tem por objetivo separar os dados (*Model*) da interface de apresentação (*View*) e do controle do fluxo do processo (*Control*). Por manter separados os dados da lógica e da apresentação, esta técnica proporciona grande modularidade e escalabilidade (BOOCH, 1996).

O relatório de referência cruzada é lido por meio do método *readInTags,* o qual cria uma árvore para especificar as entidades identificadas e a quem estas entidades estão vinculadas.

O método *readInTags* identifica, na leitura do arquivo de entrada, os tipos dos *tokens*, como, por exemplo, tipo *Classe*, *Método*, *Variável,* etc. Para cada tipo é verificada a linha na qual essa entidade se encontra e esse tipo é vinculado de acordo com a sua abrangência. Por exemplo: quando identificado um tipo *Classe,* esse passa a ser definido como corrente e, então, os tipos *Métodos* identificados, a partir desse momento, tornam-se vinculados a essa classe. No momento em que for identificado um outro tipo *Classe,* esse passará a ser o corrente. Isto acontece, também, com os atributos e variáveis. O método cria, portanto, uma árvore com as entidades identificadas no arquivo fonte, de acordo com as classes envolvidas.

Em um segundo momento, é lido o arquivo fonte original por meio do método *diplayFile*. Cada linha do arquivo é lida através do dispositivo de entrada via método *readLine,* disponível na classe *DataInputStream*. Para cada linha lida é incrementado um contador que identifica esta linha. É por meio do identificador de linha que será feita a relação com o número de linhas onde se encontram as entidades armazenadas na árvore mencionada anteriormente.

A Figura 6.5 demonstra a interface de visualização de referência cruzada. O projetista pode, a partir da seleção das entidades, por meio de uma barra de rolagem, verificar no código fonte onde esse elemento se encontra. O exemplo caracteriza a seleção do objeto *listaBens* (da classe *AtivoComposite*) e sua respectiva definição e utilização no código fonte associado. O modelo, justamente, caracteriza o controle a partir da localização, na árvore, de entidades e respectiva linha no código fonte.

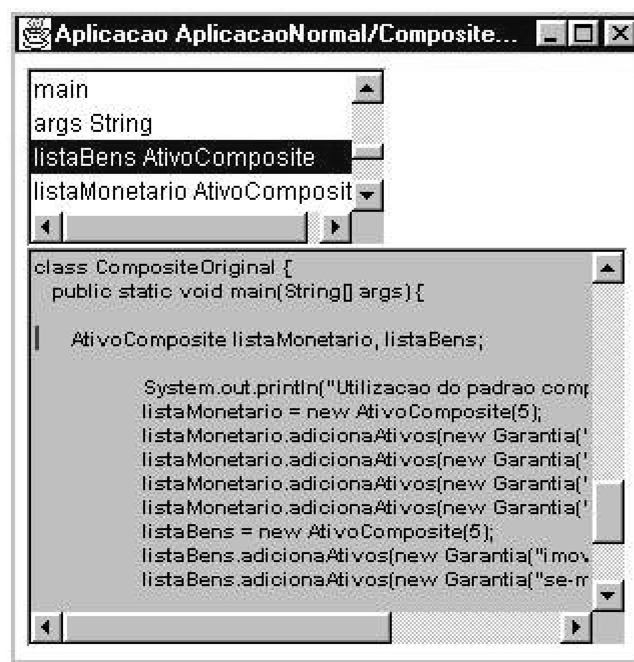

FIGURA 6.5 – Interface de Visualização Referência Cruzada

6.2.5 Extração Dinâmica de Informações

Durante a execução da aplicação do projetista, os objetos identificados para avaliação são devidamente enviados para o fluxo de saída. Conforme o exemplo a seguir a linha de execução invoca o método *writeObjeto*, que envia como parâmetros para o objeto *out* (da classe *ObjetoOutput* subclasse de *ObjectOutputStream*) o nome do objeto avaliado e uma cópia desse, executada por meio do método *clone().*

```
out.writeObjeto("listaMonetario",listaMonetario.clone());
```

62.6 Inspeção de Objetos

A classe *ApresentacaoFrame*, responsável pela interface com o usuário, faz uma verificação na estrutura *VetorDeInformacoes,* que possui as características e estados de todos os objetos avaliados e identificados na linha de execução produtora. O objeto *VetorDeInformacoes* é composto de objetos da classe *EstadosDoObjeto*, a qual possui a identificação do objeto e uma lista de estados para cada.

A partir desta estrutura, *ApresentacaoFrame* coloca a identificação de cada objeto em um componente do tipo *JcomboBox,* propiciando ao projetista proceder à escolha mais adequada para inspeção. Para cada objeto selecionado no componente *JcomboBox,* a ferramenta inspeciona *EstadosDoObjeto* e disponibiliza os estados que o objeto apresentou durante a execução da aplicação, em ordem crescente numérica, por meio de um componente *JList*.

A ordem crescente numérica não caracteriza nenhuma diferença de importância entre os estados, mas sim caracteriza os estados assumidos durante o ciclo de vida do objeto inspecionado.

Por fim, a visualização dos objetos é feita por meio da interface demonstrada na Figura 6.6. A janela também disponibiliza dois componentes do tipo *button* que serão comentados a seguir. Conforme já mencionado, o exemplo caracteriza a existência dos dois objetos *listaBens* e *listaMonetario,* bem como a seqüência de estados (*estado 0*, *estado 1*, ... , *estado 3*) do objeto em avaliação *listaBens*.

FIGURA 6.6 – Interface de Representação dos Objetos Identificados

Para cada objeto selecionado, a interface deve alterar os estados disponíveis para o novo objeto selecionado. Esta operação é executada pelo método *eventoCombo.* Neste

método, é avaliada a seleção do projetista e removidos da janela os estados anteriores e acrescentados os do objeto selecionado.

A classe *ApresentacaoFrame,* além da janela de apresentação dos objetos e respectivos estados, faz a chamada à execução da interface responsável pelas informações sobre as classes dos objetos identificados.

Esta janela, denominada *ApresentacaoClasses*, tem por finalidade demonstrar em formato de árvore, a classe à qual o objeto pertence, respectivas superclasses, bem como os atributos identificados nessas classes.

A Figura 6.7 demonstra o exemplo do objeto *listaBens*, que caracteriza um objeto da classe *AtivoComposite*, subclasse de *Ativo* e *Object*. Ao lado é apresentado o atributo *ativos,* que simboliza uma coleção (classe *Vector*). Caso o projetista queira observar os atributos das superclasses, basta que as selecione.

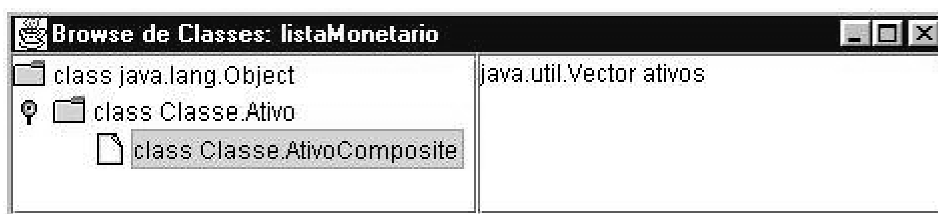

FIGURA 6.7 – Interface de Representação de Classe/Superclasses do Objeto

A classe *ApresentacaoClasse* utiliza o componente *JTree,* o qual permite a utilização de árvores e disponibiliza o serviço de processamento de pedidos do projetista para expandir e contrair nós. Em um primeiro momento a implementação de árvores pode ser atingida por meio da utilização das interfaces *TreeModel* ou *DefaultTreeModel* (TOPLEY, 1998). Neste trabalho optou-se em utilizar *DefaultTreeModel,* pois esta é fornecida pela biblioteca *Swing*.

A construção desta interface tira proveito de que os objetos usuários dos nós da árvore podem ser de qualquer tipo. Como os nós descrevem classes, nestes serão armazenados objetos do tipo *Class*. Para que não seja incluído o mesmo objeto duas vezes, precisa-se verificar se uma classe já existe ou não na árvore. O método *findUserObject* é responsável por encontrar um nó objeto usuário.

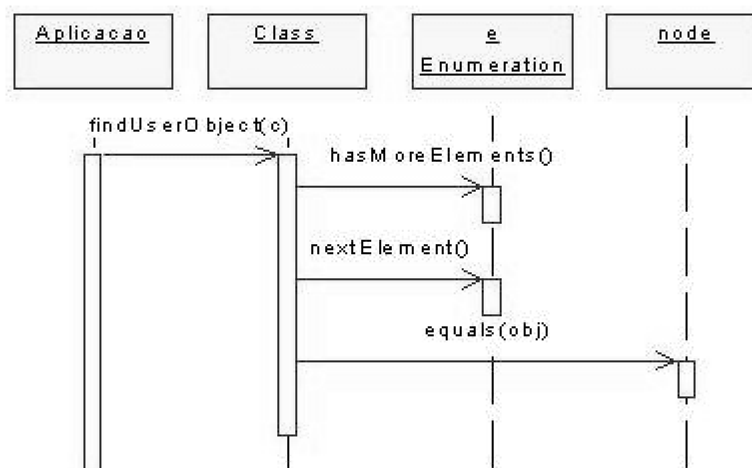

FIGURA 6.8 – Diagrama de Mensagens – *findUserObject()*

Um componente de árvore corresponde a algum outro componente. Quando o projetista seleciona alguma classe representada por um nó da árvore, as informações respectivas da classe serão demonstradas, ou seja, quando selecionada uma classe, os

atributos de instância e estáticos devem ser apresentados. Para caracterizar tal comportamento, deve-se instalar um ouvinte de seleção caracterizado pelo método *valueChanged* da interface *TreeSelectionListener*.

Esse método é chamado quando o usuário seleciona ou anula a seleção de nós da árvore. Por meio do método *getSelectionPath,* faz-se o retorno do caminho selecionado. Caso o caminho se apresente como nulo, o método retorna vazio; caso contrário, pesquisa-se o último nó do caminho na intenção de encontrar o objeto usuário. Por fim, é invocado o método *getFieldDescription,* que utiliza reflexão para criar uma *string* com todos os campos da classe selecionada, *string* esta que será apresentada na área de texto.

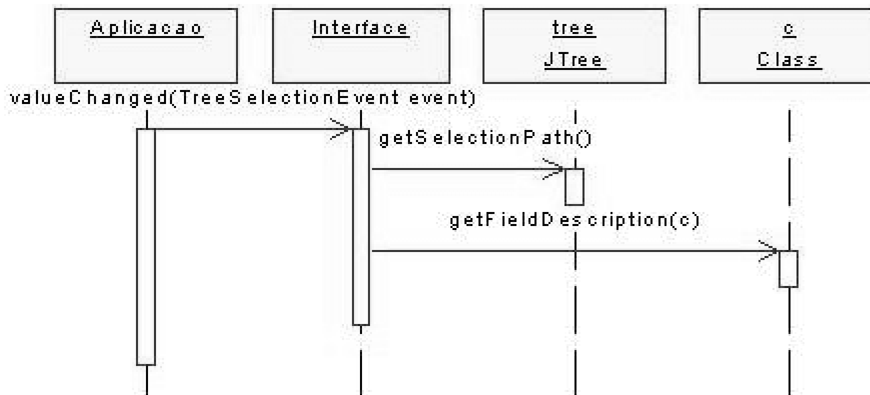

FIGURA 6.9 – Diagrama de Mensagens – *valueChanged()*

O método *getFieldDescription,* invocado por *valueChanged,* especifica tipos e nomes de campos. O método *getDeclaredFields*, sobre uma classe em observação, retorna uma coleção de atributos pertencentes à classe. Esta coleção é do tipo *Field* e, como tal, faz-se uma repetição para identificar cada um dos atributos, que vão compor a descrição da classe, mediante seu tipo e nome.

Considerando, portanto, o mecanismo de reflexão para examinar a estrutura de uma classe, o pacote *java.lang.reflect* possui as seguintes classes: *Field*, *Method* e *Constructor.*

Os métodos *getFields*, *getMethods* e *getConstructors* da classe *Class* retornam *arrays* de campos, operações e construtores que a classe suporta como *arrays* de objetos da classe apropriada *java.lang.reflect.* Já os métodos *getDeclaredFields*, *getDeclaredMethods* e *getDeclaredConstructors* retornam *arrays* contendo todos os campos, operações e construtores da classe.

6.2.7 Verificação de Objetos

As Figuras 6.10 e 6.11 demonstram, respectivamente, exemplos do objeto *listaBens* e suas devidas colaborações apresentadas de modo gráfico, nos estados 1 e 3. O objeto *listaBens* (da classe *AtivoComposite*), por meio do atributo *ativos*, relaciona-se, no estado 1, com um único objeto Garantia; e no estado 3, com dois objetos do tipo *Garantia* e um outro do tipo *AtivoComposite*. O segundo objeto *AtivoComposite,* no estado 3, relaciona-se com outros quatro objetos *Garantia.* É importante salientar que o projetista pode selecionar cada um dos objetos para proceder a inspeção sobre a evolução dos estados deste.

A classe *DiagramaObjetos* é responsável pela construção desta interface. Para cada objeto identificado nas colaborações, a partir do objeto inspecionado, é gerado um

conjunto de arestas e vértices que irão possibilitar a representação gráfica destes elementos. Optou-se aqui em utilizar uma representação bidimensional para traçado dos objetos.

FIGURA 6.10 – Diagrama de Objetos – Estado 1

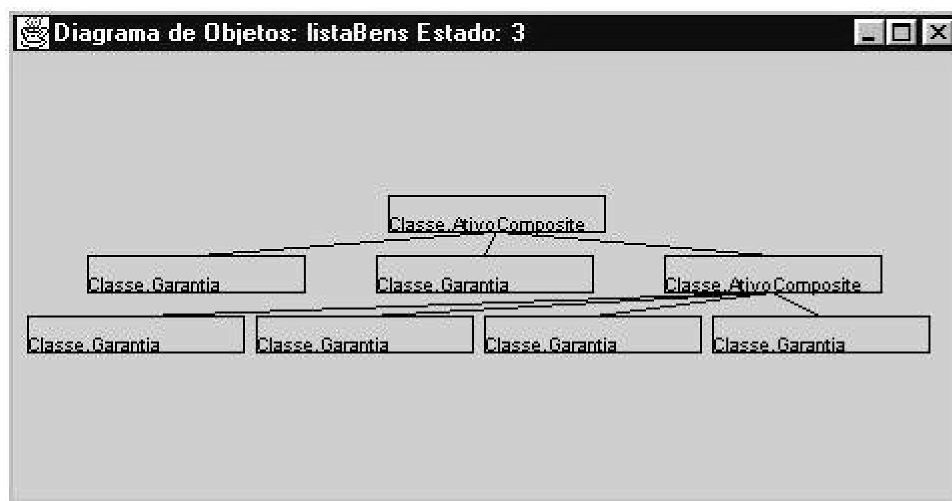

FIGURA 6.11 – Diagrama de Objetos – Estado 3

O método *criaArestasVertices* demonstra como os objetos são identificados e representados. O método *criaArestasVertices,* em um primeiro instante, testa se os objetos representam objetos dos tipos: primitivos, *strings* ou numéricos. Caso positivo, não será feita a representação destes elementos; caso contrário, serão identificados os atributos deste objeto, no intuito de procurar os relacionamentos com outros objetos. Neste sentido, também serão inspecionadas as superclasses para verificar possíveis atributos que existam nestas, por meio da repetição no método *getDeclaredFields*.

Para cada classe identificada é criado um vetor de atributos (do tipo *Field*), o qual será inspecionado pela ferramenta. Cada atributo é, então, testado individualmente para verificar se este compõe uma colaboração com outros objetos ou não.

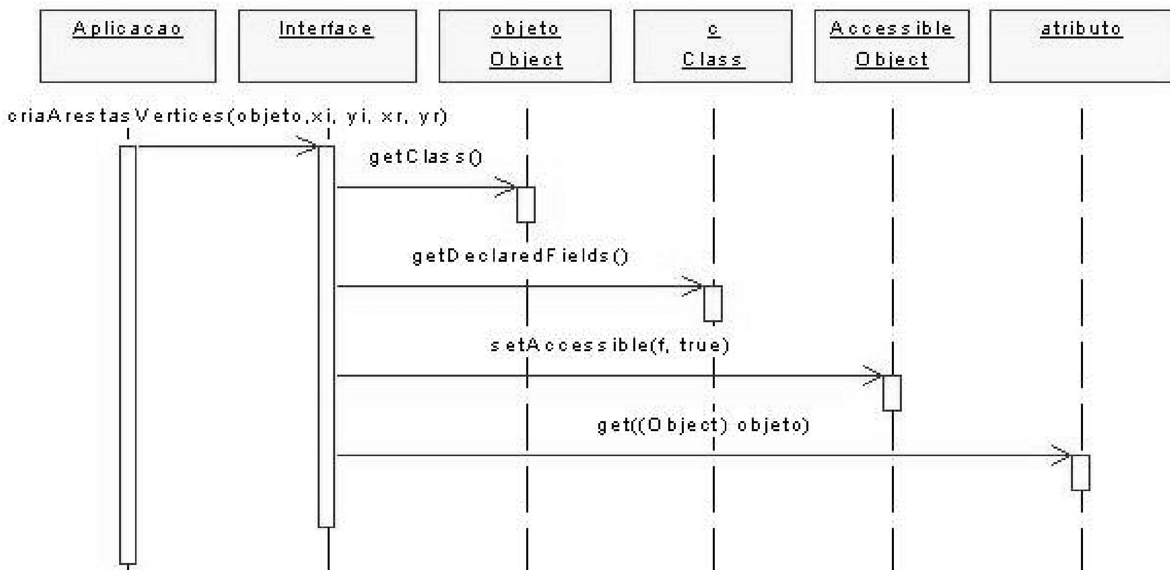

FIGURA 6.12 – Diagrama de Mensagens – Inspeção de Atributos

Primeiro, faz-se a verificação se estes atributos fazem parte da classe *Collection*. Caso representem atributos como vetores, listas, conjuntos, entre outros, serão tratados por meio de iteradores. Os iteradores, conforme já discutido anteriormente, proporcionam a inspeção dos elementos da coleção por meio dos métodos *next* (verifica o próximo elemento) e *hasnext* (verifica se existe um próximo elemento).

Caso os atributos pertençam à classe *Array,* serão tratados de maneira diferente, ou seja, para tipos da classe *Array* serão utilizados os métodos *getLength* (verifica o tamanho do *array*) e *get* (lê os elementos do *array*).

Por fim, caso a aplicação em análise apresente uma colaboração via um atributo de valor único, ou seja, uma associção para um elemento, este será também verificado após os testes de coleções e *arrays*.

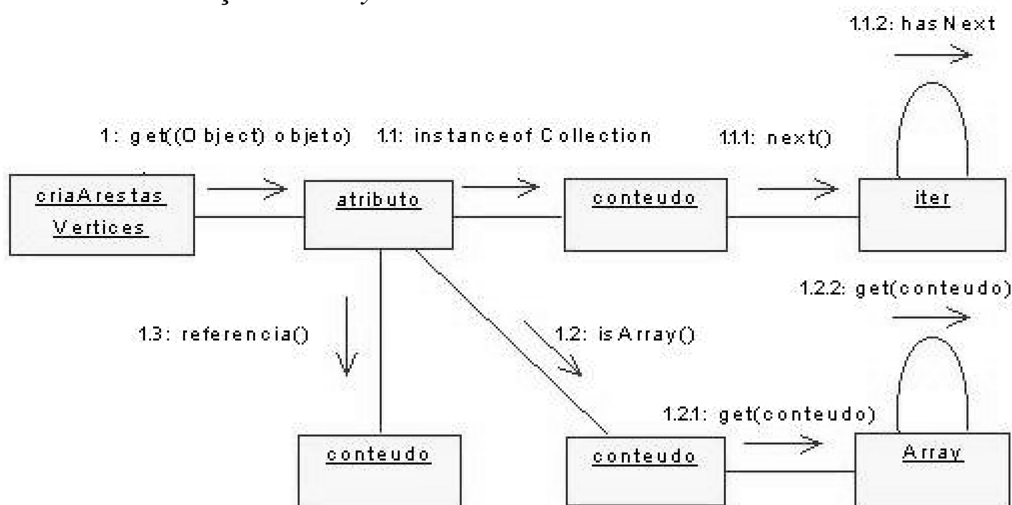

FIGURA 6.13 – Diagrama de Colaborações – *criaArestasVertices()*

É importante salientar que cada objeto pode relacionar-se com outros via um determinado atributo, e estes outros podem possuir relações com outros, e assim sucessivamente. A idéia, aqui apresentada, propõe a chamada recursiva do método *criaArestasVertices.* É por meio da chamada recursiva que serão verificadas todas as colaborações possíveis entre os objetos envolvidos. Para a chamada recursiva, são

identificados objetos que possivelmente já tenham sido avaliados e, portanto, evita-se que a execução entre em uma repetição infinita. Isto se dá pelo armazenamento dos objetos avaliados em uma tabela temporária.

Na interface *DiagramaObjetos*, os objetos são apresentados por meio de caixas e representados internamente por coordenadas cartesianas a serem visualizadas de forma bidimensional. Cada objeto identificado possui quatro arestas, compostas por quatro vértices (cordenadas: x e y). Optou-se, neste trabalho, por utilizar coordenadas de universo para definir os objetos a serem plotados. Como a janela de interface possui uma resolução diferente optou-se por utilizar a transferência das coordenadas de localização para uma *ViewPort*. Uma *ViewPort* corresponde a uma região retangular que representa as coordenadas de tela (ou dispostivo gráfico). Portanto, todas as coordenadas de universo devem ser mapeadas para as coordenadas da *ViewPort* (FOLEY, 1995)

```
// Exemplo colecao de arestas
// arestas=(nome1 (1 2) (2 3) (3 4) (4 1) nome2 (5 6) (6 7) (7 8) (8 5))
// Exemplo colecao de vertices
// vertices=((3 2) (3 7) (6 7) (6 2) (3 -4) (3 -2) (6 -2) (6 -4))
```

As colaborações também são consideradas na mesma tabela, ocupando apenas uma aresta representada por dois vértices. Acredita-se que esta representação possa caracterizar certa fragilidade de representação e tem-se como objetivo, no futuro, viabilizar melhor esta forma de apresentação, utilizando entidades internas da própria linguagem Java. É importante citar que se faz uma repetição para a coleção de arestas disponíveis e para cada aresta verificam-se os seus vértices. Os vértices são, portanto, convertidos para as coordenadas *ViewPort*, para que depois seja traçada a aresta.

Outra interface disponível, representada na Figura 6.14, demonstra o exemplo do objeto *listaBens* e suas devidas colaborações apresentadas de modo textual. Note-se que cada objeto *Garantia* demonstra o seu conteúdo por meio da identificação de seus atributos. O objeto *AtivoComposite* também se relaciona com quatro objetos *Garantia*, devidamente identificados mediante seus atributos.

A construção da interface de inspeção textual é muito semelhante à interface de inspeção de classes/superclasses (Figura 6.7). A classe responsável pela construção desta interface é a classe *InspecaoFrame*. A classe *InspeçãoFrame,* como a classe *ApresentacaoClasse*, também utiliza o componente *JTree* o qual permite a utilização de árvores e disponibiliza o serviço de processamento de pedidos do projetista para expandir e contrair nós. Neste trabalho optou-se em utilizar *ObjectTreeModel* que herda propriedades da classe *TreeModel,* ao contrário de utilizar *DefaultTreeModel.* Esta utilização justifica-se pelo fato de que os objetos inspecionados já estão vinculados entre si por meio de referências de objetos, o que acarreta a não-necessidade de duplicação da estrutura de vinculação.

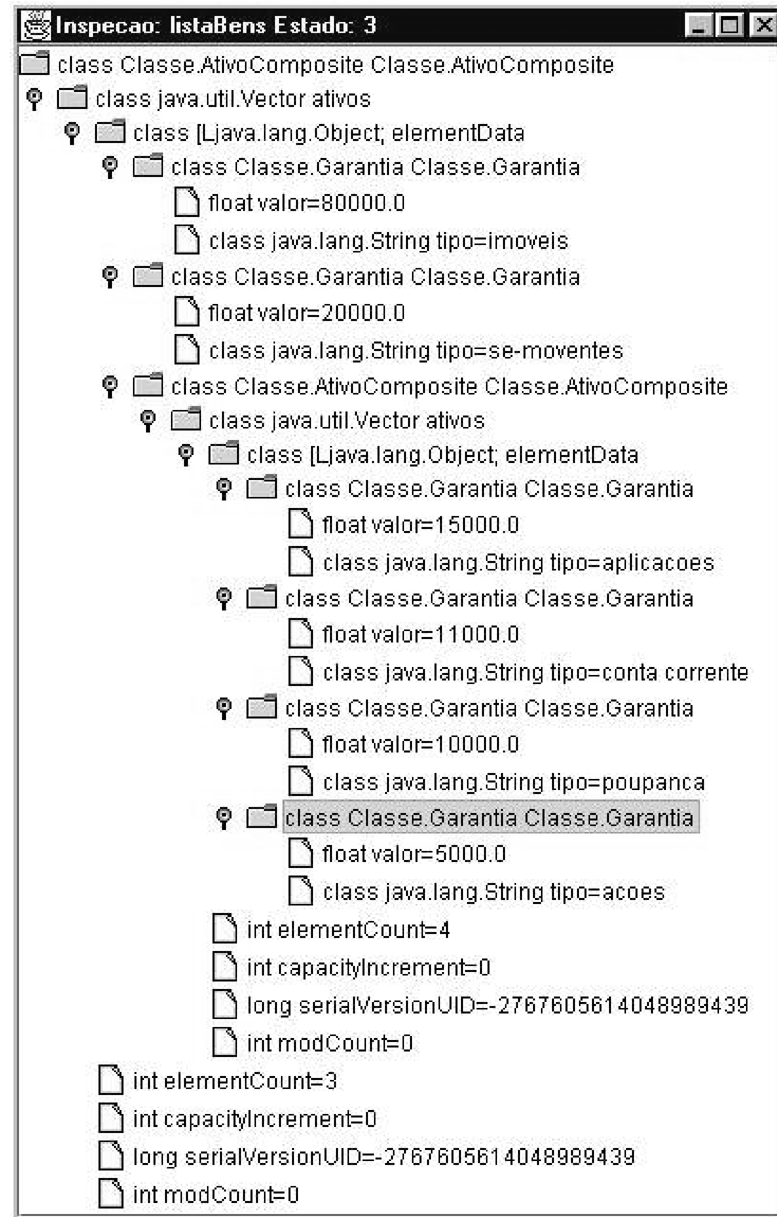

FIGURA 6.14 – Interface de Inspeção de Objetos

O método *getChild,* da classe *InspeçãoFrame,* especifica os atributos identificados para um determinado objeto, ou seja, por meio do método *getFields* são verificados os atributos do objeto informado na entrada do método. O método *getFields*, sobre uma classe em observação, retorna uma coleção de atributos pertencentes à classe. Esta coleção é do tipo *Field* e, como tal, faz-se uma repetição para identificar cada um dos atributos, que vão compor a descrição da classe, mediante seu tipo e nome. Caso o objeto não possua atributos, mas se caracterize como um objeto do tipo *array,* faz-se necessário examinar o objeto do *array* indexado pelo índice de entrada. Neste sentido, o método devolverá as informações deste objeto em um outro do tipo *Variable*. Caso contrário, ele retornará um objeto do tipo *Variable* com as informações encontradas. A classe *Variable*, na verdade, oferece uma interface de comunicação para árvore, pois, quando solicitada alguma pesquisa, esta será respondida via classe *Variable,* a qual informará o conteúdo de um determinado atributo por meio do método *getValue().*

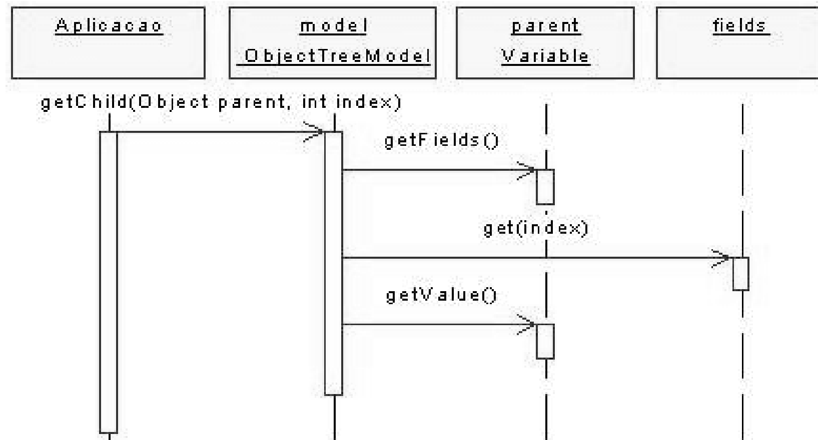

FIGURA 6.15 – Diagrama de Mensagens – *getChild()*

6.2.8 Identificação de Colaborações e Padrões

Finalmente, a interface *DiagramaDeClasses*, representada na Figura 6.16, é construída de maneira semelhante à interface *DiagramaDeObjetos* (Figura 6.11). Inicialmente são verificadas todas as colaborações dos objetos envolvidos durante a inspeção. É importante salientar que, para a construção do diagrama de classes, são verificados todos os estados disponíveis capturados para um determinado objeto. O exemplo a seguir demonstra a avaliação do objeto *listabens*, o qual representa um objeto da classe *AtivoComposite*. A classe *AtivoComposite* relaciona-se com ela mesma por meio da relação dos objetos *listabens* e *listamonetário*. Para representar a relação dos objetos *imoveis* e *se-moventes* com *listabens* e *aplicacoes*, *contacorrente*, *poupanca* e *acoes* com *listamonetario,* faz-se a relação *AtivoComposite Garantia* com cardinalidade n.

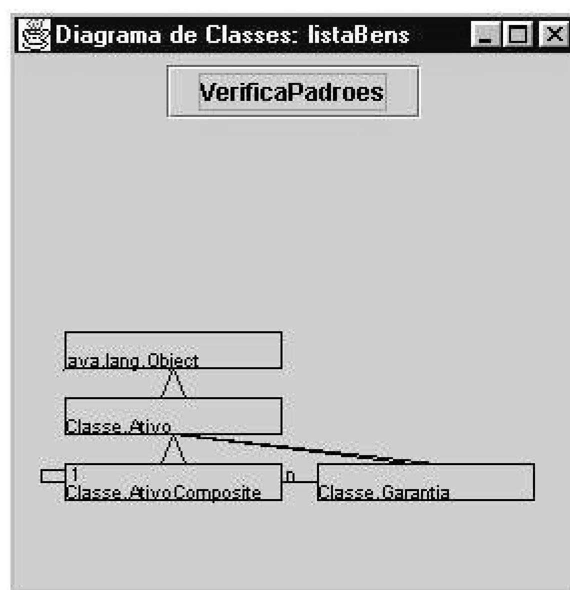

FIGURA 6.16 – Diagrama de Classes da Aplicação

O método *procuraRelacoes,* da classe *DiagramaDeClasses*, faz uma repetição considerando a *listaDeEstados* disponíveis para um determinado objeto. Neste método inicializa-se um tipo da classe *Vector* denominado *relacoes,* o qual conterá todas as colaborações entre classes construídas a partir da inspeção dos estados. Para que se concretize tal verificação, é feita a invocação do método *procuraClassesRecursivo*.

O método *procuraClassesRecursivo* é implementado de maneira semelhante ao método *criaArestasVertices*, já citado, ou seja, é feita uma verificação nos estados de cada objeto, considerando seus atributos e relações nos respectivos estados. A diferença concentra-se na chamada ao método *procuraRelacionamentos,* que verifica a existência ou não, no vetor *relacoes,* de uma colaboração identificada. Se esta colaboração não existe, será criada; caso contrário, será incrementado o seu indicador de cardinalidade. Um outro fator considerado é o grau de profundidade da relação, ou seja, se a relação se dá do objeto inspecionado diretamente com seus colaboradores, ou se de seus colaboradores com outros, e assim sucessivamente. Considerando o exemplo tratado até aqui, a avaliação do vetor *relacoes* conteria a estrutura abaixo.

```
Vetor de Cardinalidades
Classe.AtivoComposite x Classe.Garantia
Estado: 1 Prof:1 Numero:1
Estado: 2 Prof:1 Numero:2
Estado: 3 Prof:1 Numero:2
Estado: 3 Prof:13 Numero:4
Vetor de Cardinalidades
Classe.AtivoComposite x Classe.AtivoComposite
Estado: 3 Prof:1 Numero:1
```

O objeto *listabens* no estado 1, relação direta, demonstra uma relação com 1 objeto da classe *Garantia*. Já no estado 2, existe uma relação com 2 objetos de *Garantia*. No estado 3, mantém a relação com 2 objetos *Garantia*, mas por relação indireta, verifica-se a relação com 4 objetos *Garantia*, ou seja, como *listabens* relaciona-se com *listamonetario,* verifica-se na estrutura uma relação indireta.

A construção do diagrama se dá de maneira semelhante à já comentada para o diagrama de objetos, onde as classes a partir do vetor *relacoes* são consideradas como caixas e construídas por meio de pontos identificados por um conjunto de arestas e vértices.

A partir deste momento o projetista pode escolher a opção *VerificaPadrões,* definida na interface. Esta opção coloca em prática um conjunto de regras para testar as diferentes características encontradas na aplicação, em comparação à essência do padrão que está sendo procurado. A seguir, procede-se à avaliação para comparação com o padrão *Composite*.

Para cada relacionamento previsto na estrutura *relações,* primeiro é verificado se as classes possuem uma estrutura de superclasses em comum. Faz-se, portanto, uma comparação de cada superclasse da primeira, verificando se esta se encontra na estrutura da segunda. Quando as classes são iguais (representam um relacionamento entre objetos da mesma classe), é a partir destas que será representada a colaboração com a superclasse. Caso sejam diferentes, é importante verificar se estas classes representam os objetos folha. É verificado, no caso do exemplo, que *listabens* se relaciona com *listamonetario,* o que representa cardinalidade 1. Mas tanto *listabens* como *listamonetario* relacionam-se com objetos do tipo *Garantia* com cardinalidade n. Neste caso é importante que a relação com a superclasse *Ativo* apresente cardinalidade n. Por este motivo, é alterada a cardinalidade de *AtivoComposite,* pois é a partir desta que será construída a relação com a superclasse, por meio da verificação de objetos folha que venham a modificar a cardinalidade.

Caso as classes sejam diferentes mas possuam a mesma estrutura de superclasses, faz-se a verificação no intuito de analisar se existe só uma relação com filhos ou se existe uma relação recursiva. Caso exista somente uma relação com objetos filhos, verifica-se que o padrão *Composite* não está sendo usado em sua totalidade e então é apresentada uma mensagem informando que não há relação entre objetos do tipo *Composite*, mas sim que só há entre *Composite* e folhas.

É importante salientar que o conjunto *relacoes* é alterado nas cardinalidades à medida que se identifica qual relacionamento será construído com a superclasse, conforme já comentado anteriormente. Para tal, a indicação da colaboração com a classe que representa os objetos filhos deve ser suprimida. Caso não exista a estrutura recursiva, esta relação não deve ser suprimida. Por este motivo, alteram-se as indicações para mostrar ou não as relações em função das relações entre objetos da mesma classe.

O padrão *Composite* demonstra a existência de métodos com a mesma assinatura na superclasse e nas classes que compõem a estrutura dos relacionamentos. São avaliados todos os métodos da superclasse com os métodos da primeira classe, após todos os métodos da superclasse com os métodos da segunda classe e, finalmente, comparam-se os métodos das duas classes.

Finalmente, considerando todas as condições, atributo de relacionamento, relação com objetos da mesma classe, relação de objetos de classe diferente com uma mesma superclasse em comum aos de mesma classe, assinatura de métodos em comum, passa-se à verificação da cardinalidade. Conforme já comentado, se a relação está marcada implicitamente, verifica-se a sua cardinalidade.

A construção do diagrama acontece de maneira semelhante às já comentadas. O projetista pode visualizar as características citadas e mensagens de diagnóstico na interface da Figura 6.17.

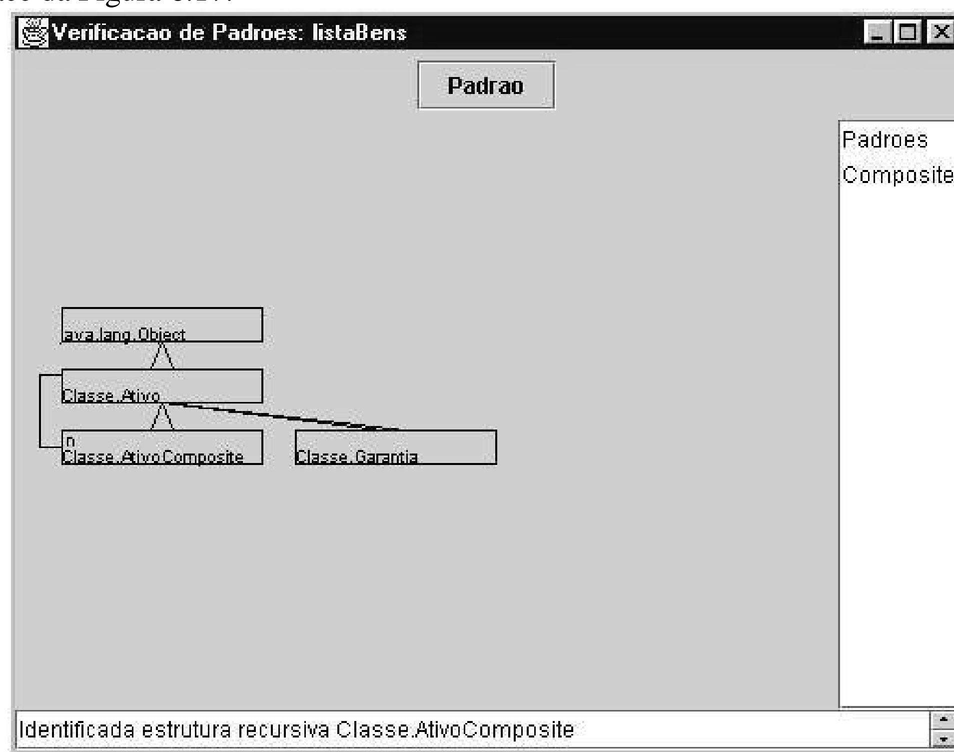

FIGURA 6.17 – Diagrama de Classes c/Identificação de Padrões

Quando identificado um padrão na estrutura, o projetista pode verificar uma interface de ajuda, mostrando o padrão na sua essência. No caso, é apresentada a interface do padrão *Composite* com seus respectivos detalhes (Figura 6.18).

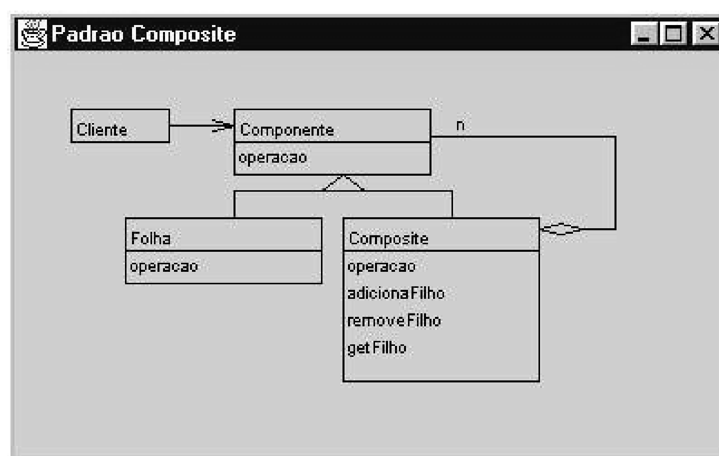

FIGURA 6.18 – Interface de Ajuda de Reconhecimento

6.3 Conclusões

Este capítulo tem por função demonstrar os processos administrados pela ferramenta de inspeção, bem como apresentar as interfaces disponíveis para interação com o projetista.

O objetivo foi demonstrar, a partir de uma aplicação exemplo, as características básicas da ferramenta, desde o processo inicial de referência cruzada, até o processo de identificação de padrões em tempo de execução.

A ênfase da ferramenta de inspeção concentra-se, portanto, em fornecer subsídios ao projetista, a respeito da execução da aplicação. O projetista dispõe de mecanismos de visualização das informações a respeito dos estados dos objetos avaliados. A ferramenta tem condições de dispor as características dos objetos ao longo de seu ciclo de vida.

Os mecanismos de reflexão computacional permitem o acesso às diferentes informações dos estados de um objeto. É a partir destas informações que são verificadas as colaborações, estabelecendo as relações entre objetos ou classes. A partir destas relações, são estabelecidas regras para verificar o quanto este conjunto de informações disponíveis caracteriza-se como a essência dos padrões estudados.

O trabalho de Freitas (2002A), intitulado "Uma Ferramenta de Inspeção para Aplicações Java Utilizando Reflexão Computacional", caracteriza que a ferramenta ainda se encontra em fase de construção, mas já conta com todos os processos aqui mencionados. A ferramenta captura um ou mais objetos, a partir do código fonte, e identifica as colaborações desses objetos por meio da inspeção de seus atributos em tempo de execução. Essas informações são apresentadas, portanto, em janelas apropriadas, onde o projetista pode diagnosticar e interagir com as informações contidas nos objetos.

O projetista pode visualizar, por meio de diagramas, as colaborações dos objetos com relação a outros objetos relacionados. As colaborações não se limitam somente ao objeto em teste, mas sim a todos os objetos relacionados a esse, e assim sucessivamente. Por meio de um método com chamada recursiva, pesquisam-se todos os atributos dos objetos envolvidos nos relacionamentos. A partir dessas informações, procede-se a verificação dos padrões de projeto.

Resumidamente, a ferramenta descrita apresenta quatro processos principais: referência cruzada, execução e extração de informações, reflexão computacional e identificação de colaborações e padrões. Todos os processos apresentam, portanto, interfaces de acesso para que o projetista possa visualizar as informações disponíveis.

7 Conclusões

Com o objetivo de auxiliar os processos de manutenção e documentação de software orientado a objetos, este trabalho apresenta um estudo que possibilite tornar mais automáticas as atividades de entendimento de sistemas de software.

Estratégias de produção de software como padrões de projeto visam a tornar mais automáticas as atividades de projeto de sistemas. Os padrões de projeto são classificados como padrões criacionais, estruturais e comportamentais. Um conhecimento aprofundado sobre os padrões de projeto facilita a construção de software orientado a objetos, proporcionando a definição de projetos de boa qualidade. Observando sob esta ótica verifica-se a grande utilização, por parte de projetistas, de estruturas como padrões de projeto.

Mas, ao mesmo tempo que os padrões de projeto representam um recurso positivo para aumentar os índices de qualidade do sistema gerado, significam, também, em muitos casos, dificuldades de entendimento em relação às estruturas utilizadas para resolver um determinado problema. Esta dificuldade de entendimento muitas vezes é aumentada com a inserção de recursos que se valem da experiência humana.

7.1 Contribuições

Neste sentido, esta pesquisa teve como objetivo propor a automatização da identificação de padrões de projeto por meio da inspeção de aplicações Java em tempo de execução. Inicialmente, foram descritos alguns trabalhos desenvolvidos na área de padrões, que serviram como subsídios para as idéias aqui citadas. Estes trabalhos enfatizam a utilização e a importância dos padrões. Ao mesmo tempo, foram caracterizados trabalhos que demonstram a preocupação na utilização e reconhecimento dos padrões em sistemas de software.

Visando a alcançar este objetivo, apresentou-se um conjunto de regras que estabelecem a redução dos padrões conhecidos ao mínimo de estruturas necessárias e identificáveis no domínio da solução. Preliminarmente, percebe-se que um estudo somente sobre as estruturas dos padrões, conforme citado nos trabalhos de Krämer (1996), Bansyia (1998), Seeman (1998) e Guéhéneuc (2001), não finaliza o estudo porque tais estruturas são, muitas vezes, semelhantes. Portanto, como vários padrões tendem a usar estruturas básicas aproximadas, para que se possam reduzir identificações errôneas, fez-se necessário estender a pesquisa procurando por detalhes no modelo dinâmico da aplicação. A procura por detalhes em tempo de execução exige avaliar a aplicação, extraindo características pertinentes exigidas. A diferença deste trabalho em relação aos apresentados é justamente no que tange à avaliação dos padrões sobre uma aplicação em tempo de execução.

A essência dos padrões, conforme descrito no capítulo 4, é a determinação de um conjunto de estruturas e colaborações entre os objetos envolvidos na aplicação. A partir das relações, foram verificadas as evidências de existência dos padrões propostos. Neste processo, fez-se uma avaliação de alguns padrões como: *Composite*. *Decorator*, *Strategy*, *Template Method*, *Observer*, *Chain of Responsibility* e *State*. Justifica-se o trabalho com tais padrões, pois estes representam modelos extremamente utilizados em sistemas de software, conforme trabalhos de Gamma (1994) e Alpert (1998), além de outros. Tem-se como meta estender a proposta a um maior número de padrões possível, integrando as regras representativas dos diversos padrões existentes.

Em um segundo momento, apresentou-se o protótipo de uma ferramenta que visa a propor a validação das regras estudadas. A ferramenta está sendo construída em

Java. A escolha por tal linguagem justifica-se pelo fato de que esta apresenta um conjunto de facilidades para manipular a avaliação de objetos, utilizando reflexão computacional, além de caracterizar-se como uma linguagem de grande utilização nos meios acadêmicos e profissionais. Acredita-se que a ferramenta necessite de uma série de aprimoramentos, pois ainda se encontra em estado prematuro e, portanto, não se pode afirmar que apresente 100 por cento de certeza lógica em seus resultados (FREITAS, 2002A). Faz-se necessário, portanto, incluir um maior número de testes e ajustes referentes a aplicações reais. Até o momento, o reconhecimento dos padrões pela ferramenta alcança um índice regular de identificação, levando em consideração aplicações de pequeno porte.

O objetivo da ferramenta é avaliar as informações estáticas e dinâmicas. A avaliação do modelo estático da aplicação em análise é efetuada através de um processo de referência cruzada o qual extrai um relatório de informações a respeito das entidades utilizadas na aplicação. O processo foi construído com o apoio da ferramenta ANTLR, *ANother Tool for Language Recognition,* que dispõe de um *framework* para a construção de tradutores a partir de gramáticas predefinidas. O relatório de referência cruzada é formado pelo conjunto de classes identificadas na aplicação e suas respectivas informações.

A representação das informações referentes ao modelo dinâmico é feita a partir da execução da aplicação em análise, em uma linha de execução separada, considerada produtora, e a ferramenta é executada em uma outra linha de execução, denominada consumidora. Portanto, cópias dos estados dos objetos avaliados são realizadas de uma linha para a outra.

Sobre os objetos avaliados, aplica-se o processo de reflexão computacional o qual extrai informações referentes ao modelo dinâmico. A partir do processo de reflexão computacional, são verificadas as colaborações levando em consideração todos os estados pelos quais os objetos passam durante a execução da aplicação. Considerando-se estas informações, verificam-se as evidências que levam a definir a existência dos padrões na aplicação inspecionada. Caso essas características venham a ser atendidas na quase totalidade, verificam-se fortes indícios da existência dos padrões comentados.

7.2 Trabalhos Futuros

O estudo realizado representa um esforço na direção da pesquisa para a criação de técnicas que a visam otimizar as tarefas de manutenção e documentação de sistemas de software OO. Acredita-se que outros aspectos precisam ser estudados, como investigar a estrutura do sistema quando identificado um padrão e conferir se esta inserção é realmente necessária. É muito importante avaliar se realmente é necessária a utilização de um padrão em uma aplicação, justificando tal fato, pois muitas vezes o projetista pode utilizar uma outra técnica de implementação. Em muitos casos, a complexidade gerada pela utilização de um padrão não representa um esforço significativo se comparado à pouca utilização dos benefícios fornecidos. Pode-se observar que, neste trabalho, a ferramenta identifica características que demonstram a pouca utilização dos recursos oferecidos pelo padrão, mas não faz opção pela sua utilização ou não.

Tem-se como meta para trabalhos futuros armazenar as informações examinadas em uma base de dados, para posterior análise após a execução da mesma aplicação, várias vezes, levando em consideração a diversidade de entradas e saídas.

Ainda existem muitas subáreas abertas ao estudo, relacionadas a esta tese. Neste trabalho foram verificados apenas alguns padrões de projeto e ainda existem muitos

outros na literatura, que poderiam ser identificados potencialmente por meio de ferramentas automatizadas. A literatura de padrões é consideravelmente extensa e trabalhos como o de Grand (1998) comprovam o acréscimo de novos padrões aos já definidos por Gamma (1994).

A ferramenta de inspeção descrita nesta tese servirá como objeto para a geração de mais estudos relacionados à área em questão. A avaliação em tempo de execução propõe uma rápida aproximação à identificação dos padrões, no intuito de determinar colaborações entre classes e objetos. É importante ressaltar a necessidade de estudos empíricos no que tange à ocorrência de padrões de projeto vinculados à subjetividade no desenvolvimento do software.

Por fim, este trabalho faz parte do projeto de pesquisa vinculado à FAPERGS – Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul, intitulado "Uma Ferramenta para Avaliação de Heurísticas na Detecção e Inserção Automáticas de Padrões de Projeto em Software Orientado a Objetos" (FREITAS, 2002B). Este projeto tem como objetivos não só a identificação de padrões de projeto, mas também a inserção destes em uma aplicação já desenvolvida. Para tal, é importante ressaltar a investigação de mudanças na estrutura do sistema quando for necessário incluir um padrão e conferir se esta inserção é realmente necessária.

Finalmente, acredita-se que o trabalho desenvolvido poderá ser útil para os estudos de construção de ferramentas que auxiliem o projetista em atividades de manutenção e documentação de sistemas de software, ou seja, ferramentas capazes de transformar uma aplicação em execução, em abstrações de alto nível, visando a facilitar as modificações necessárias no sistema diante de alternativas disponíveis.

Anexo 1 Descrição da Análise Léxica e Sintática de ANTLR

A descrição da análise léxica em ANTLR começa pela definição da classe *JavaLexer,* que herda propriedades da super classe *Lexer*, disponível no pacote de ANTLR. Entre algumas opções disponíveis, estão: o vocabulário do qual será gerado o código fonte, a identificação de tipos longos, teste de literais e número de caracteres avaliados à frente.

```
class JavaLexer extends Lexer;
options {
    exportVocab=Java;
    longestPossible=true;
    testLiterals=false;
    k=4;
}
```

Após a declaração da classe, faz-se a descrição dos operadores que estarão disponíveis para avaliação. Optou-se aqui em demonstrar apenas parte dos símbolos disponíveis.

```
LPAREN          :   '('     ;
RPAREN          :   ')'     ;
LCURLY          :   '{'     ;
RCURLY          :   '}'     ;
COLON           :   ':'     ;
EQUAL           :   "=="    ;
NOT_EQUAL       :   "!="    ; ...
```

Os caracteres separadores também devem ser identificados. É importante ressaltar a necessidade de controle destes caracteres, que demonstram os caracteres para leitura do próximo *token* ou geração de nova linha.

```
WS  :   (   ' ' | '\t' | '\f' |
            (   "\r\n"   // DOS
            |   '\n'     // Unix
            )
            { newline(); }
            )
        { _ttype = Token.SKIP; }
    ;
```

Quanto à identificação para o reconhecimento de comentários, faz-se a especificação para comentários descritos em um única linha (//) ou a partir de várias linhas (/* */).

```
SL_COMMENT
    :   "//" (~'\n')* '\n'
         { _ttype = Token.SKIP; newline(); }
    ;
ML_COMMENT
    :   "/*"
        (   { LA(2)!='/' }? '*'
        |   '\n' { newline(); }
        |   ~('*'|'\n')
        )*
        "*/"
         { _ttype = Token.SKIP; }
    ;
```

Para o controle de caracteres, define-se a especificação para identificação de apenas um caracter ou uma seqüência (string). Além de um caracter normal, as regras podem apresentar a seqüência *ESC*, descrita a seguir.

```
CHAR_LITERAL
    :   '\'' (ESC|~'\'') '\''
    ;
STRING_LITERAL
    :   '"' (ESC|~'"')* '"'
    ;
```

A seqüência *ESC* é definida protegida, ou seja, pode apenas ser chamada por outra regra léxica. Esta regra não retorna diretamente valores para o *parser*.

```
protected
ESC
    :   '\\'
        (   'n'
        |   'r'
        |   't'
        |   'b'
        |   'f'
        |   '"'
        |   '\''
        |   '\\'
        |   ('u')+ HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
        |   ('0'..'3') ( ('0'..'9') ('0'..'9')? )?
        |   ('4'..'7') ('0'..'9')?
        );
```

A representação de símbolos hexadecimais é feita separadamente pela regra a seguir.

```
protected
HEX_DIGIT
    :   ('0'..'9'|'A'..'F'|'a'..'f')
    ;
```

Para o controle de identificadores, é utilizada a representação a seguir. Os identificadores são formados por caracteres alfabéticos e sublinha, seguidos de alfanuméricos. A regra *testLiterals* é inicialmente aceita como verdadeira. Isso significa que depois, da execução da regra se verifica o literal na tabela de símbolos para avaliar se esse realmente é um identificador.

```
IDENT
    options {testLiterals=true;}
    :   ('a'..'z'|'A'..'Z'|'_'|'$')
       ('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'$')*;
```

O controle de *tokens* numéricos é feito utilizando a regra a seguir. A regra contempla a verificação da parte inteira, bem como a parte fracionária, caso o *token* caracterize um elemento com ponto flutuante. Caso identificado o "." (ponto), a regra procede ao reconhecimento do número de casas decimais possíveis (para elementos com ponto flutuante sem a parte inteira), verificando, após, a existência dos elementos representativos da exponenciação e sufixo (para identificação de *tokens* do tipo *float* ou *double*). Se considerada a parte inteira, a regra avalia apenas o zero, verifica se o *token* contém elementos para formação hexadecimal ou octal, ou, ainda propõe a verificação numérica decimal normal. Por fim, se o *token* for um inteiro, existe também a representação para um *large*, por exemplo. Caso a regra identifique um "." após a

parte inteira, a regra procede à verificação da parte fracionária, de maneira semelhante à citada acima, agora para elementos com ponto flutuante com a parte inteira associada.

```
NUM_INT
    {boolean isDecimal=false;}
    :   '.' {_ttype = DOT;}
            (('0'..'9')+ (EXPONENT)? (FLOAT_SUFFIX)? { _ttype =
                                                NUM_FLOAT; })?
    |   (   '0' {isDecimal = true;}
        (   ('x'|'X') (HEX_DIGIT)+
            |  ('0'..'8')+
        )?
        |   ('1'..'9') ('0'..'9')*  {isDecimal=true;}
        )
        (   ('l'|'L')
        |   {isDecimal}?
            (   '.' ('0'..'9')* (EXPONENT)? (FLOAT_SUFFIX)?
            |   EXPONENT (FLOAT_SUFFIX)?
            |   FLOAT_SUFFIX
            )
            { _ttype = NUM_FLOAT; }
        )?
    ;
```

A seguir são apresentadas as regras para formação de *tokens* com expoentes e respectivo sufixo, para elementos do tipo ponto flutuante. As regras são definidas protegidas, pois são utilizadas exclusivamente pela regra *NUM_INT*.

```
protected
EXPONENT
    :   ('e'|'E') ('+'|'-')? ('0'..'9')+
    ;
protected
FLOAT_SUFFIX
    :   'f'|'F'|'d'|'D'
    ;
```

A descrição da análise léxica gera, portanto, por meio da ferramenta ANTLR o arquivo *JavaLexer.java* com todas as definições informadas. A seguir é apresentada a definição do método *FLOAT_SUFFIX* em função de sua definição acima.

```
protected final void mFLOAT_SUFFIX(boolean _createToken) throws
RecognitionException,CharStreamException,TokenStreamException {
        int _ttype; Token _token=null; int begin = text.length();
        _ttype = FLOAT_SUFFIX;
        int _saveIndex;

        switch ( LA(1)) {
               case 'f': {
                      match('f');
                      break;
               }
               case 'F': {
                      match('F');
                      break;
               }
               case 'd': {
                      match('d');
                      break;
               }
               case 'D': {
                      match('D');
                      break;
               }
               default: {
               throw new NoViableAltForCharException((char)LA(1),
                      getFilename(), getLine());
               }
        }
        if ( _createToken && _token==null && _ttype!=Token.SKIP ) {
               _token = makeToken(_ttype);
               _token.setText(new String(text.getBuffer(), _begin,
                             text.length()-_begin));
        }
        returnToken = _token;
  }
```

A descrição da análise sintática em ANTLR começa pela definição da classe *JavaXref,* que herda propriedades da super classe *Parser*, disponível no pacote de ANTLR. Entre algumas opções disponíveis, estão: número de caracteres avaliados à frente, o vocabulário do qual será gerado o código fonte, níveis de otimização e a variável para verificação de erros em *handlers*.

```
class JavaXref extends Parser;
 options {
     k = 2;
     exportVocab=Java;
     codeGenMakeSwitchThreshold = 2;
     codeGenBitsetTestThreshold = 3;
     defaultErrorHandler = false;
 }
```

A descrição do método *main*, responsável pela ativação da aplicação de referência cruzada, também é feita a partir da classe *JavaXref*. Entre alguns identificadores globais, é definida a tabela de símbolos que armazenará a definição dos *tokens* encontrados. O método *main* faz a identificação de possíveis arquivos fontes a serem lidos para executar o processo de geração de informações. O processo resume-se à execução dos métodos *doFile, resolveTypes* e *report*. O método *doFile* é responsável pela leitura do(s) arquivo(s) fonte(s) e execução do *parser*. Já o método *resolveTypes* tem como objetivo resolver as pendências encontradas na tabela de símbolos por meio de uma análise final sobre os *tokens* investigados. E, por fim, *report* fará a geração do relatório de saída a partir da tabela de símbolos.

```
public static void main(String[] args) {
   try {
     SymbolTable symbolTable = new SymbolTable();
     if (args.length > 0 ) {
        System.err.println("Compilando...");
        for(int i=0; i< args.length;i++)
           doFile(new File(args[i]), symbolTable);
        System.err.println("Resolvendo tipos...");
        symbolTable.resolveTypes();
     }
     else
            ...
     IndentingPrintWriter ipw = new
                           IndentingPrintWriter(saida);
     System.err.println("Preparando Relatorio...");
     symbolTable.report(ipw);
     ipw.close();
     saida.close();
   }
   catch(Exception e) {...}
}
```

O método *doFile* é responsável pelo gerenciamento das entradas e execução do *parser*. Este método possibilita o gerenciamento de vários arquivos fontes, levando em consideração o diretório destes. Caso seja informado um diretório, este verificará todos os arquivos do diretório e tornará a chamar o método *doFile* para cada arquivo ou diretório informado. Caso seja encontrado um arquivo, este fará a verificação da extensão *java* para chamar, após, o método *parseFile* para este arquivo.

```
public static void doFile(File f, SymbolTable symbolTable)
                                  throws Exception {
     if (f.isDirectory()) {
        String files[] = f.list();
        for(int i=0; i < files.length; i++)
            doFile(new File(f, files[i]), symbolTable);
     }
     else if ((f.getName().length()>5) && f.getName().Substring
                          (f.getName().length()-5).equals(".java")) {
                  symbolTable.setFile(f);
                  System.err.println("   "+f.getAbsolutePath());
                  parseFile(new FileInputStream(f), symbolTable);
             }
}
```

O método *parseFile* tem por finalidade criar o analisador léxico da classe *JavaLexer*, cuja função é ler os dados da entrada padrão, informando ao *scanner* para criar uma associação à estrutura dos *tokens* do pacote *Cruzada*, classe *JavaToken*. Após, é feita a criação do *parser* que terá como entrada os *tokens* do analisador léxico, sendo inicializado o processo de verificação a partir do método *compilationUnit*.

```
public static void parseFile(InputStream s, SymbolTable symbolTable)
                                             throws Exception {
        try {
            JavaLexer lexer = new JavaLexer(s);
            lexer.setTokenObjectClass("Cruzada.JavaToken");
            JavaXref parser = new JavaXref(lexer);
            parser.setSymbolTable(symbolTable);
            parser.compilationUnit(); }
        catch (Exception e) {...}
}
```

A partir deste ponto, serão descritas algumas regras sintáticas utilizadas neste trabalho. Em função de a descrição destas regras caracterizar-se como muito longa, preferiu-se omitir algumas. A regra a seguir define a avaliação de tipos nativos em Java.

```
builtInType returns [JavaToken t]
    {t=null;}
    :   bVoid:"void"       {t = (JavaToken)bVoid;}
    |   bBoolean:"boolean" {t = (JavaToken)bBoolean;}
    |   bByte:"byte"       {t = (JavaToken)bByte;}
    |   bChar:"char"       {t = (JavaToken)bChar;}
    |   bShort:"short"     {t = (JavaToken)bShort;}
    |   bInt:"int"         {t = (JavaToken)bInt;}
    |   bFloat:"float"     {t = (JavaToken)bFloat;}
    |   bLong:"long"       {t = (JavaToken)bLong;}
    |   bDouble:"double"   {t = (JavaToken)bDouble;}
    ;
```

Já a regra *identifier* faz a validação dos identificadores.

```
identifier returns [JavaToken t]
    {t=null;}
    :   id1:IDENT      {t=(JavaToken)id1;}
        (   DOT
            id2:IDENT  {t.setText(t.getText() + "." + id2.getText());}
        )*
    ;
```

A regra *modifier* faz a verificação dos modificadores de acesso.

```
modifier
    :   "private"
    |   "public"
    |   "protected"
    |   "static"
    |   "transient"
    |   "final"
    |   "abstract"
    |   "native"
    |   "threadsafe"
    |   "synchronized"
    |   "const"
    ;
```

As regras *classDefinition* e *interfaceDefinition* fazem a validação dos identificadores que representam classes e interfaces em Java. Para cada uma das regras, são criados na tabela de símbolos os identificadores com as suas respectivas informações.

```
classDefinition
    {JavaToken superClass=null; JavaVector interfaces=null;}
    :   "class" id:IDENT
            ("extends" superClass=identifier )?
            (interfaces=implementsClause)?
            {defineClass((JavaToken)id, superClass, interfaces);}
            classBlock
        {popScope();}    ;

interfaceDefinition
    {JavaVector superInterfaces = null;}
    :   "interface" id:IDENT
            (superInterfaces=interfaceExtends)?
            {defineInterface((JavaToken)id, superInterfaces);}
            classBlock
        {popScope();}    ;
```

A regra *classBlock* caracteriza as informações que possam existir no escopo de uma classe delimitadas pelas chaves de abertura e fechamento. *ClassBlock* faz referência à existência de elementos do tipo *field,* seguidos de ";".

```
classBlock
    :   LCURLY
            ( field | SEMI )*
        RCURLY
    ;
```

A regra *field* tem por função avaliar para uma determinada classe, as declarações de métodos e atributos nessa contidos.

```
Field {JavaToken type;}
    :   modifiers
        (   methodHead[null
                compoundStatement[BODY]
        |   classDefinition
        |   interfaceDefinition
        |   type=typeSpec  // method or variable declaration(s)
            (   methodHead[type]
                    ( compoundStatement[BODY] | SEMI {popScope();})
            |   variableDefinitions[type] SEMI
            )
        )
    |   "static" compoundStatement[CLASS_INIT]
    |   compoundStatement[INSTANCE_INIT]
    ;
```

As regras a seguir propõem, na seqüência, a definição e declaração de atributos(variáveis), métodos e parâmetros de entrada e saída.

```
variableDefinitions[JavaToken type]
    :   variableDeclarator[type]
        (COMMA variableDeclarator[type] )*    ;
variableDeclarator[JavaToken type]
    :   id:IDENT (LBRACK RBRACK)* ( ASSIGN initializer )?
        {defineVar((JavaToken)id, type);}     ;
arrayInitializer
    :   LCURLY
            (   initializer ( COMMA initializer )*
                (COMMA)?
            )?
        RCURLY    ;
initializer
    :   expression
    |   arrayInitializer    ;
methodHead[JavaToken type]
    {JavaVector exceptions=null;}
    :   method:IDENT
        {defineMethod((JavaToken)method, type);}
        LPAREN (parameterDeclarationList)? RPAREN
        (LBRACK RBRACK)*
        (exceptions=throwsClause)?
        {endMethodHead(exceptions);}    ;
parameterDeclarationList
    :   parameterDeclaration ( COMMA parameterDeclaration )*    ;
parameterDeclaration
    {JavaToken type;}
    :   ("final")? type=typeSpec id:IDENT (LBRACK RBRACK)*
        {defineVar((JavaToken)id, type);}
    ;
```

Já a regra *compoundStatement* é responsável pela identificação de blocos de comandos fazendo a chamada da regra *statement*.

```
compoundStatement[int scopeType]
    :   lc:LCURLY
            {
                switch(scopeType) {
                    case NEW_SCOPE:
                        defineBlock((JavaToken)lc);
                    break; ...
                }
            }
            (statement)*
            {popScope();}
        RCURLY
    ;
```

Já a regra *statement*, definida a seguir, faz a verificação sintática dos comandos *if-else*, *for*, *while*, *do-while*, *break*, *continue*, *return*, *case* e opções *throw* e *synchronized*.

```
statement
    {int count = -1;}
    :   compoundStatement[NEW_SCOPE]
    |   (declaration)=> declaration SEMI
    |   id:IDENT COLON statement  {defineLabel((JavaToken)id);}
    |   expression SEMI
    |   "if" LPAREN expression RPAREN statement
        ( "else" statement )?
    |   "for"
            LPAREN
                (forInit)?        SEMI    // initializer
                (expression)?     SEMI    // condition test
                (count=expressionList)?   // updater
            RPAREN
            statement                     // statement to loop over
    |   "while" LPAREN expression RPAREN statement
    |   "do" statement "while" LPAREN expression RPAREN SEMI
    |   "break" (bid:IDENT {reference((JavaToken)bid);})? SEMI
    |   "continue" (cid:IDENT {reference((JavaToken)cid);})? SEMI
    |   "return" (expression)? SEMI
    |   "switch" LPAREN expression RPAREN
            LCURLY
                ((("case" expression | "default") COLON)+ (statement)*
                )*
            RCURLY
    |   tryBlock
    |   "throw" expression SEMI
    |   "synchronized" LPAREN expression RPAREN statement
    |   SEMI
    ;
```

As regras *forInit* e *tryBlock* definem, respectivamente, a incialização para comandos *for* e a verificação dos comandos de exceção *try-catch*.

```
forInit
    {int count = -1;}
    :   (declaration)=> declaration
    |   count=expressionList    ;
tryBlock
    :   "try" compoundStatement[NEW_SCOPE]
        (handler)*
        ( "finally" compoundStatement[NEW_SCOPE] )?    ;
handler
    : "catch" LPAREN parameterDeclaration RPAREN
            compoundStatement[NEW_SCOPE]    ;
```

A seguir, são definidas as regras para expressões. Conforme demonstrado abaixo, uma expressão pode conter outras subexpressões, e assim sucessivamente.

```
expression
    :   assignmentExpression
    ;
expressionList returns [int count]
    {count=1;}
    :   expression (COMMA expression {count++;})*
    ;
assignmentExpression
    :   conditionalExpression
        (   (   ASSIGN
            |   PLUS_ASSIGN
            |   MINUS_ASSIGN
            |   STAR_ASSIGN
            |   DIV_ASSIGN
            |   MOD_ASSIGN
            |   SR_ASSIGN
            |   BSR_ASSIGN
            |   SL_ASSIGN
            |   BAND_ASSIGN
            |   BXOR_ASSIGN
            |   BOR_ASSIGN
            )
            assignmentExpression
        )?
    ;
conditionalExpression
    :   logicalOrExpression
        (QUESTION conditionalExpression COLON conditionalExpression )?
    ;
```

Na seqüência são definidas as regras para expressões lógicas. Para simplificar este trabalho, optou-se em apresentar apenas a expressão para verificar o OR lógico. As outras seguem a mesma analogia.

```
logicalOrExpression
    :   logicalAndExpression (LOR logicalAndExpression)*
    ;
```

As regras abaixo verificam a igualdade em expressões a partir da regra *equalityExpression*.

```
equalityExpression
    : relationalExpression ((NOT_EQUAL | EQUAL) relationalExpression)*
    ;
relationalExpression
    :   shiftExpression
        (   (   LT
            |   GT
            |   LE
            |   GE
            )
            shiftExpression
        )*    ;
shiftExpression
    : additiveExpression ((SL | SR | BSR) additiveExpression)*    ;
      additiveExpression
    : multiplicativeExpression ((PLUS | MINUS)
                multiplicativeExpression)*    ;
      multiplicativeExpression
    : castExpression ((STAR | DIV | MOD ) castExpression)*    ;
```

A regra *postfixExpression* verifica nomes de classes e/ou objetos, expressões que envolvam *arrays* e operadores de incremento e decremento.

```
postfixExpression
    {JavaToken t; int count=-1;}
    :   t=primaryExpression // start with a primary
     (DOT ( id:IDENT {if (t!=null)
           t.setText(t.getText()+"."+id.getText());}
    | "this"   {if (t!=null) t.setText(t.getText()+".this");}
    | "class"  {if (t!=null) t.setText(t.getText()+".class");}
     )
        |   LBRACK expression RBRACK
        |   LPAREN
                ( count=expressionList
                | {count=0;}
                )
            RPAREN
            {
                if (t!=null)
                    t.setParamCount(count);
            }
        )*
        {if (t != null) reference(t);}
        (   INC
        |   DEC
        |
        )    ;
```

Na seqüência, a regra *primaryExpression* identifica os elementos básicos de uma expressão, disponibilizando os elementos: *true*, *false*, *this* e *null*.

```
primaryExpression returns [JavaToken t]
    {t=null;}
    :   id:IDENT {t = (JavaToken)id;}
    |   t=builtInType DOT "class" {t.setText(t.getText()+".class");}
    |   t=newExpression
    |   constant
    |   s:"super"        {t = (JavaToken)s;}
    |   "true"
    |   "false"
    |   th:"this"        {t = (JavaToken)th; setNearestClassScope();}
    |   "null"
    |   LPAREN expression RPAREN     ;
```

A regra *newExpression* faz a validação para criação de novas instâncias de objetos.

```
newExpression returns [JavaToken t]
    {t=null; int count=-1;}
    :   "new" t=type
        (   LPAREN
                ( count=expressionList
                | {count=0;}
                )
            RPAREN
                {
                    t.setText(t.getText()+".~constructor~");
                    t.setParamCount(count);
                }
                (classBlock)?
        |   (   LBRACK
                    (expression)?
                RBRACK
            )+
            (arrayInitializer)?
        )    ;
```

E, por fim, a regra *constant* realiza a verificação de constantes a partir de números inteiros, com ponto flutante e literais.

```
constant
    :   NUM_INT
    |   CHAR_LITERAL
    |   STRING_LITERAL
    |   NUM_FLOAT    ;
```

A descrição da análise sintática gera, portanto, por meio da ferramenta ANTLR o arquivo *JavaXRef.java* com todas as definições informadas. Abaixo é apresentada a definição do método *constant* em função de sua definição acima.

```
public final void constant() throws RecognitionException,
                                     TokenStreamException {
     Switch ( LA(1)) {
            case NUM_INT:
            {
                   match(NUM_INT);
                   break;
            }
            case CHAR_LITERAL:
            {
                   match(CHAR_LITERAL);
                   break;
            }
            case STRING_LITERAL:
            {
                   match(STRING_LITERAL);
                   break;
            }
            case NUM_FLOAT:
            {
                   match(NUM_FLOAT);
                   break;
            }
            default:
            {
            throw new NoViableAltException(LT(1), getFilename());
            }
     }
}
```

Este anexo tem por finalidade demonstrar a construção da ferramenta de referência cruzada utilizando como base a ferramenta ANTLR. Foram descritas algumas regras léxicas e sintáticas para demonstrar como se procede à geração de um analisador para códigos Java.

A submissão das regras à ferramenta ANTLR gera arquivos fonte em Java que detêm o código criado a partir dessas regras.

Anexo 2 Métodos para Identificação de Padrões

Com a utilização de *pipes,* podem-se conectar várias linhas de execução entre si, sem a preocupação com o sincronismo das linhas de execução. O fragmento de código abaixo apresenta tais características.

```
private void eventoInspecao() {
            ...
            try {
                 ativaMonitor.start();
                 PipedOutputStream pout = new
                                            PipedOutputStream();
                 PipedInputStream pin = new
                                            PipedInputStream(pout)
                  ...
                 te = new ThreadExecucao(processorName, pout);
                 tc = new ThreadConsumo(pin);
                 ...
                 te.start();
                 tc.start();
            }
            catch (IOException e) {
                   ...;
            }
}
```

A opção adotada neste trabalho foi implementar a classe *SerialCloneable*. Esta classe implementa facilidades das classes *Cloneable* e *Serializable*. A classe *SerialCloneable* redefine, portanto, o método clone com o modificador de acesso público, conforme definido a seguir.

```
public class SerialCloneable implements Cloneable,
                                            Serializable {
    public Object clone() {
       try {
         ByteArrayOutputStream bout = new ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream(bout);
         out.writeObject(this);
         out.close();
         ByteArrayInputStream bin = new
                           ByteArrayInputStream(bout.toByteArray());
         ObjectInputStream in = new ObjectInputStream(bin);
         Object ret = in.readObject();
         in.close();
         return ret;
       }
       catch (Exception e) {
         return null;
       }
    }
}
```

Na classe *ObjetoOutput,* o clone do objeto é recebido pelo método *writeObjeto,* que instancia um novo objeto a partir da classe *objetoTransporte* e passa estas informações como parâmetros.

```
public class ObjetoOutput extends ObjectOutputStream { ...
    public void writeObjeto(String nomeObjeto, Object objeto) {
        try {
            SerialCloneable objetoTipoClone = (SerialCloneable)
                objeto;
            ObjetoTransporte objetoTransporte = new
                ObjetoTransporte(nomeObjeto, objetoTipoClone);
            this.writeObject(objetoTransporte);
            this.flush();
        }
        catch (Exception e) {
        }
    }
}
```

A classe *ObjetoTransporte,* descrita a seguir, simplesmente encapsula uma cópia do objeto a trafegar no *pipe* e sua descrição. Portanto, um objeto desta classe é que será recebido na *thread* de consumo.

```
public class ObjetoTransporte extends SerialCloneable {
   public ObjetoTransporte(String nomeObjeto,
                           SerialCloneable objeto) {
      nome = nomeObjeto;
      objetoEmAvaliacao = (SerialCloneable) objeto.clone();
   }   ...
   private String nome;
   private SerialCloneable objetoEmAvaliacao;
}
```

A linha de execução, demonstrada a seguir, verifica se o estado atual, capturado na linha de execução do projetista, já existe na coleção de estados disponíveis.

```
public void run() {   ...
        while(repeticao) {
          try {
            Object objetoExecutado = in.readObject();
            ObjetoTransporte objetoTransporte =
                   (ObjetoTransporte) objetoExecutado;
            boolean existe = false;
            for (i=0; i < VetorDeInformacoes.tamanho(); i++) {
              EstadosDoObjeto objeto = (EstadosDoObjeto)
                   VetorDeInformacoes.retornaObjeto(i);
              nomeObjetoTransporte = objetoTransporte.nome();
              if (nomeObjetoTransporte.equals
                   (objeto.getNomeObjeto())) {
                     objeto.addEstado((SerialCloneable)
                          objetoTransporte.objeto());
                     existe = true;
                  }
            }
            if (! existe) {
                 nomeObjetoTransporte = objetoTransporte.nome();
                 EstadosDoObjeto estado = new EstadosDoObjeto
                     (nomeObjetoTransporte, (SerialCloneable)
                     objetoTransporte.objeto());
                 VetorDeInformacoes.adiciona(estado);
              }
              corrente++;
          }
          catch(Exception e) {   ...
          }
        }
    }   ...
  }
```

Por fim, a descrição parcial da classe *EstadosDoObjeto* é feita a seguir. O método *addEstado* recebe como entrada o objeto que representa o estado atual capturado na aplicação do projetista. De posse desta informação, o método verifica qual o último estado disponível na lista de estados para compará-lo como o objeto recebido.

```
public class EstadosDoObjeto {
   public EstadosDoObjeto(String nomeObjeto, Object objeto) {
      listaDeEstados = new Vector(10);
      nome = nomeObjeto;
      addEstado((SerialCloneable) objeto);
   }
   ...
   public void addEstado(SerialCloneable objeto) {
         if (listaDeEstados.size() > 0) {
              SerialCloneable objetoTemp = (SerialCloneable)
                   listaDeEstados.get(listaDeEstados.size() - 1);
              try {
                    ByteArrayOutputStream bout1 = new
                         ByteArrayOutputStream();
                    ObjectOutputStream out1 = new
                         ObjectOutputStream(bout1);
                    out1.writeObject(objetoTemp);
                    ByteArrayOutputStream bout2 = new
                         ByteArrayOutputStream();
                    ObjectOutputStream out2 = new
                         ObjectOutputStream(bout2);
                    out2.writeObject(objeto);
                    if (bout1.size() == bout2.size()) {
                         byte ba1[] = bout1.toByteArray();
                         byte ba2[] = bout2.toByteArray();
                         int igual = 0;
                         for(int w=0; w < bout1.size(); w++) {
                                if (ba1[w] == ba2[w])
                                igual++;
                       }
                       if (igual != bout1.size())
                       listaDeEstados.add((SerialCloneable)
                         objeto.clone());
                 }
                 else
                       listaDeEstados.add((SerialCloneable)
                         objeto.clone());
                 bout1.close();
                 bout2.close();
                 out1.close();
                 out2.close();
           }
           catch (Exception e) {
             ...
           }
      }
      else
           listaDeEstados.add((SerialCloneable) objeto.clone());
      ...
}
```

É por meio da chamada recursiva do método *criaArestasVertice* que são verificadas todas as colaborações possíveis entre os objetos envolvidos em uma inspeção. Para a chamada recursiva, são identificados objetos que possivelmente já tenham sido avaliados e, portanto, evita-se que a execução entre em uma repetição infinita. Isto se dá pelo armazenamento dos objetos avaliados em uma tabela temporária.

```
public boolean criaArestasVertices(Object objeto, int xi, int yi,
                                                    int xr, int yr) {
 ...
 Class c = objeto.getClass();
 if (!c.isPrimitive() &&
     !c.equals(String.class) &&
     !(c.getSuperclass()).equals(Number.class) &&
      objeto != null) {
      try {
       int i = 0;
       Field atributo = null;
       while (c != null) {
         Field[] f = c.getDeclaredFields();
         AccessibleObject.setAccessible(f, true);
          while (i < f.length) {
              atributo = f[i];
               if (atributo != null) {
                 Object conteudo = atributo.get((Object) objeto);
                  if (conteudo != null) {
                   Class catributo = conteudo.getClass();
                   if(!catributo.isPrimitive() &&
                      !catributo.equals(String.class) &&
                     !(catributo.getSuperclass()).equals(Number.class)){
                    if (conteudo instanceof Collection) {
                          Collection colecao = (Collection) conteudo;
                          if (!colecao.isEmpty()) {
                            ...
                            Iterator iter = colecao.iterator();
                             Object objetoNext = (Object) iter.next();
                             indice = procuraObjetoExistente(objetoNext);
                             if (criaArestasVertices...
                             while (iter.hasNext()) {
                               objetoNext = (Object) iter.next();
                               if (criaArestasVertices ...
                            }
                          }
                   }
                   else {
                    if ((conteudo.getClass()).isArray()) {
                     ...
                     for(int w = 0; w < Array.getLength(conteudo); w++){
                      Object objetoArray = (Object) Array.get(conteudo,w);
                        indice = procuraObjetoExistente(objetoArray);
                        if (criaArestasVertices...
                      }
                    }
                    else {
                          indice = procuraObjetoExistente(conteudo);
                          if (criaArestasVertices...
                     }
               ...
               }
               i++;
          }
          c = c.getSuperclass();
          i = 0;
       }
      }
      catch (IllegalAccessException e) { ...
      }
 }
 return(retorno);
}
```

Conforme já mencionado anteriormente, para cada relacionamento previsto na estrutura *relações,* primeiro é verificado se as classes possuem uma estrutura de superclasses em comum. Faz-se, portanto, uma comparação de cada superclasse da primeira, verificando se esta se encontra na estrutura da segunda. A estrutura a seguir representa o fragmento de código onde são feitas estas comparações. Nesta estrutura, são verificadas também as cardinalidades dos relacionamentos.

```
while(k1 > -1) { ...
 int k2 = classes2.size() - 1;
 while(k2 > -1) {
  if (!(((Class) classes1.get(k1)).getName()).equals(c2.getName())){
   if ((((Class) classes1.get(k1)).getName()).equals(((Class)
                                  classes2.get(k2)).getName())) {
      int j = 0;
      boolean achou = false;
      while(j <= (superClasseRelacao.size() - 1)) {
        if ((Class) classes2.get(k2) == (Class) ((SuperClasse)
                superClasseRelacao.get(j)).getSuperClasse()) {
          achou = true;
          j = superClasseRelacao.size();
        }
        j++;
     }
     SuperClasse sc;
     if (achou == false) { ...
     }
     else{
          if (c1 == c2) {
           float cardinalidade = objetoRelacao.getCardinalidade();
           String card = " ";
           if (cardinalidade > 1.0)
              card = " n ";
           else
              card = " 1 ";
           int achoufolhas = 1;
           int cont = 0;
           for(j=0; j < superClasseRelacao.size(); j++) {
            if ((Class) classes2.get(k2) == (Class) ((SuperClasse)
                       superClasseRelacao.get(j)).getSuperClasse())
             if ((Class)((SuperClasse)superClasseRelacao.get(j)).
                                              getClasse1() !=
                 (Class) ((SuperClasse) superClasseRelacao.get(j)).
                                              getClasse2()) {
              SuperClasse superc=(SuperClasse)
                                      superClasseRelacao.get(j);
              superc.modificaRelacao(0);
              cont = cont + 1;
              achoufolhas = 2;
              if (superc.getCardinalidade().equals(" n "));
                 card = " n ";
            }
          }
          if (!card.equals(" n ")) {
             if (cont > 1)
                card = " n ";
          }
          sc = new SuperClasse((Class) classes2.get(k2), c1, c2,
                      achoufolhas,card);
          if (achoufolhas > 1)
             listaM.addElement("Identificada estrutura recursiva " +
                                      c1.getName());
          superClasseRelacao.add(sc);
        } ...
```

Caso as classes sejam diferentes mas possuam a mesma estrutura de superclasses, faz-se a verificação no intuito de analisar se existe só uma relação com filhos ou se existe uma relação recursiva.

```
...
else {
  float cardinalidade = objetoRelacao.getCardinalidade();
  String card = " ";
  if (cardinalidade > 1.0)
     card = " n ";
  else
    card = " 1 ";
  int jaexiste = 1;
  int cont = 0;
  String novacard = " 1 ";
  int indiceRel = -1;
  int primeiroElemento = -1;
  for(j=0; j < superClasseRelacao.size(); j++) {
    if ((Class) classes2.get(k2) == (Class) ((SuperClasse)
         superClasseRelacao.get(j)).getSuperClasse()) {
     if (primeiroElemento == -1)
         primeiroElemento = j;
     if ((Class) ((SuperClasse) superClasseRelacao.get(j)).
         getClasse1() == (Class) ((SuperClasse)
         superClasseRelacao.get(j)).getClasse2()) {
         jaexiste = 0;
         indiceRel = j;
     }
     else {
           cont = cont + 1;
     }
    }
  }
  // para quando existe estrutura recursiva
  if (indiceRel > -1) {
     SuperClasse superc = (SuperClasse)
               superClasseRelacao.get(indiceRel);
      ...
     superc.modificaRelacao(2);
     if (!card.equals(" n ")) {
        if (cont > 1)
        novacard = " n ";
     }
     listaM.addElement("Identificada estrutura recursiva " +
               c1.getName());
     superc.alteraCardinalidade(novacard);
  }
  // para quando nao existe estrutura recursiva
  else {
        if (primeiroElemento > -1) {
           SuperClasse superc = (SuperClasse)
               superClasseRelacao.get(primeiroElemento);
         superc.modificaRelacao(2);
         if (!card.equals(" n ")) {
            if (cont > 1)
               novacard = " n ";
         }
         superc.alteraCardinalidade(novacard);
         jaexiste = 0;
         ...
         listaM.addElement("Nao Identificada estrutura
                     recursiva " + c1.getName());
      }
  }
  sc = new SuperClasse((Class) classes2.get(k2), c1, c2,
         jaexiste,card);
  superClasseRelacao.add(sc);...
```

É importante salientar que o conjunto *relacoes* é alterado na identificação de cardinalidades à medida que se identifica qual relacionamento será construído com a superclasse, conforme já comentado anteriormente.

O código abaixo demonstra um outro teste feito sobre os métodos da classe. O padrão *Composite* demonstra a existência de métodos com a mesma assinatura na superclasse e nas classes que compõem a estrutura do relacionamento. São avaliados todos os métodos da superclasse com os métodos da primeira classe; após, todos os métodos da superclasse com os métodos da segunda classe e, finalmente, comparam-se os métodos das duas classes.

```
boolean metodosMesmaAssinatura = false;
for(i=0; i < superClasseRelacao.size(); i++) {
   metodosMesmaAssinatura = false;
   Class supc1 =(Class)((SuperClasse)
         superClasseRelacao.get(i)).getSuperClasse();
   Class cl11 = (Class)((SuperClasse)
         superClasseRelacao.get(i)).getClasse1();
   Class cl12 = (Class)((SuperClasse)
         superClasseRelacao.get(i)).getClasse2();
   // Verifica metodos com mesma assinatura
   Vector metodosc1 = new Vector(10);
   Vector metodosc2 = new Vector(10);
   Method msupc1[] = supc1.getDeclaredMethods();
   for(int y=0; y < msupc1.length; y++) {
     String smsupc1 = converteString(msupc1[y].toString());
     Method mcl11[] = cl11.getDeclaredMethods();
     for(int y1=0; y1 < mcl11.length; y1++) {
          String smcl11 = converteString(mcl11[y1].toString());
          if (smsupc1.equals(smcl11))
             metodosc1.add(smcl11);
     }
   }
   for(int y=0; y < msupc1.length; y++) {
     Method mcl12[] = cl12.getDeclaredMethods();
     String smsupc1 = converteString(msupc1[y].toString());
     for(int y2=0; y2 < mcl12.length; y2++) {
         String smcl12 = converteString(mcl12[y2].toString());
         if (smsupc1.equals(smcl12))
            metodosc2.add(smcl12);
         }
   }
   for(int y=0; y < metodosc1.size(); y++) {
      for(int y1=0; y1 < metodosc2.size(); y1++) {
          if (metodosc1.get(y).equals(metodosc2.get(y1)))
              metodosMesmaAssinatura = true;
      }
   }
   if (metodosMesmaAssinatura == false)
      listaM.addElement("Nao ha metodos com assinatura iguais em " +
                        cl11.getName() + " e " + cl12.getName());
                else
       listaM.addElement("Ha metodos com assinatura iguais em " +
                        cl11.getName() + " e " + cl12.getName());
```

Finalmente, considerando todas as condições, atributo de relacionamento, relação com objetos da mesma classe, relação de objetos de classe diferente com uma mesma superclasse em comum aos de mesma classe, assinatura de métodos em comum, passa-se à verificação da cardinalidade. Conforme já comentado, se a relação está marcada, verifica-se a sua cardinalidade.

```
if (imprime > 1) {
       String card = (String) ((SuperClasse)
          superClasseRelacao.get(i)).getCardinalidade();
       if (card.equals(" n ")) {
          if (jaExisteComposite == false) {
            listaPadroes.addElement("Composite");
            jaExisteComposite = true;
           }
       }
       ...
```

Este anexo tem por finalidade demonstrar, resumidamente, alguns métodos utilizados pela ferramenta de inspeção. Faz-se importante observar a dificuldade de demonstrar os processos de investigação dos padrões por meio de código Java devido a complexidade para descrição. Neste anexo, portanto, preferiu-se comentar resumidamente fragmentos dos métodos utilizados na identificação do padrão *Composite*, o qual foi utilizado como exemplo. Preferiu-se, portanto, omitir os códigos responsáveis pela investigação dos outros padrões aqui abordados. Até a escrita desta tese, a ferramenta já conta com os procedimentos necessários para a investigação dos padrões *Composite*, *Decorator* e *Strategy*. No momento, fazem-se as implementações necessárias para o reconhecimento do padrão *Observer*.

Referências

AGARWAL R.; LAGO, P. A Paradigmatic Approach To High-level Object-oriented Sofftware development. **ACM SIGSOFT Software Engineering Notes**, New York, v.20, n. 2, p.36-41, Apr. 1995.

AHO, A. et al. **Compilers:** principles, techiniques, and tools. Reading: Addison-Wesley, 1988. 796p.

ALEXANDER, C. et al. **A Pattern Language**. Oxford: Oxford University Press, 1977.

ALPERT, S. et al. **The Design Patterns - Smalltalk Companion**. Reading: Addison-Wesley, 1998. 444p.

APPLETON, B. **Patterns and Software:** essential concepts and terminology. Disponível em: <http://www.enteract.com/~bradapp/>. Acesso em: 15 jun.2001.

ARANGO, G. et al. A process for Consolidating and Reusing Design Knowledge. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 13., 1993, Baltimore. **Proceeding...** California: IEEE Press, 1993.

BOOCH, G. **Object-Oriented Analysis and Design with Aplications**. Redwood City: Benjamin Cummings, 1994. 589p.

BANSIYA, J. Automating Design-Pattern Identification. **Dr.Dobb's Journal**, New York, v.23, n. 6, p.20-26, June 1998.

BUDINSKY, F. et al. Automatic Code Generation from Design Patterns. **IBM – Systems Journal**, New York, v.35, n. 2, p.151-158, Feb. 1996.

CAMPO, M. **Compreensão Visual de Frameworks através da Introspecção de Exemplos**. 1997. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

CHIKOFSKY, E.; CROSS, J. Reverse Engineering and Design Recovery: a Taxonomy. **IEEE Software**, New York, v. 7, n.1, p. 13-17, 1990.

COAD, P. et al. **Objet Models - Strategies, Patterns & Applications.** [S.l.]: Prentice Hall, 1997. 515p.

DEUTSCH, P. Frameworks and Reuse in Smalltalk 80 System. In: BIGGERSTAFF T.; PERLIS, A. **Software Reusability:** applications and experience. New York: ACM Press, 1989. v.1, p. 57-71.

DOURISH, P. Developing a Reflective Model of Collaborative Systems. **ACM Transactions on Computer-Human Interaction (TOCHI)**, New York, v.2, n. 1, p.40-63, Mar. 1995.

EDEN, A.; GIL, J. Automating the Application of Design Patterns. **Journal of Object Oriented Programming**, New York, v.20, n. 1, p.45-50, May 1997.

EICHELBERGER, F.; WOLFF von GUDENBERG, J. On the Visualization of Java Programs. In: SOFTWARE VISUALIZATION INTERNATIONAL SEMINAR, 2001, Dagstuhl Castle, Germany. **Proceedings…** [S.l.:s.n.], 2002. p.295-306.

FOLEY, J. et al. **Computer Graphics**: principles e practice. Reading: Addison-Wesley, 1995. 1174p.

FREITAS, A.; PRICE, A. Formalização de Heurísticas para o Apoio a Modelagem de Sistemas Orientados a Objetos. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE

SOFTWARE, SBES, 12., 1998, Maringá, PR. **Anais...** Maringá: UEM/SBC, 1998. p.313-327.

FREITAS, A.; PRICE, A. Disponibilizando o uso de Design Patterns através de um Framework em Prolog++. In: SIMPÓSIO CATARINENSE DE COMPUTAÇÃO, 1., 2000, Itajaí, SC. **Anais...** Itajaí: UNIVALI, 2000. p.13-24.

FREITAS, A.; PRICE, A. **Um Estudo sobre Metodologias e Padrões de Projeto para o Desenvolvimento de Software Orientado a Objetos.** 2000. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

FREITAS, A.; PRICE, A. Design Patterns Automatic Identification Tool In: ARGENTINE SYMPOSIUM ON SOFTWARE ENGINEERING, JAIIO, 29., 2000, Tandil, Argentina. **Proceedings...** Tandil: ISISTAN/SADIO, 2000. p. 117-131.

FREITAS, A.; PRICE, A. Heuristics for Automatic Detection of Design Patterns in Object-Oriented Software. In: WORKSHOP DISSERTAÇÕES E TESES, SBES, 5., 2000, João Pessoa, PB. **Proceedings...** João Pessoa: UFPE/SBC, 2000. p. 373-378.

FREITAS, A.; PRICE, A. Uma Ferramenta de Inspeção para Aplicações Java Utilizando Reflexão Computacional. In: CONGRESSO BRASILEIRO DE COMPUTAÇÂO, 2., 2002, Itajaí, SC. **Anais...** Itajaí: UNIVALI, 2002. 1 CD-ROM.

FREITAS, A. **Uma Ferramenta para Avaliação de Heurísticas na Detecção e Inserção Automáticas de Padrões de Projeto em Software Orientado a Objetos**. 2002. Projeto de Pesquisa - FURG – FAPERGS. Edital PROADE2, Porto Alegre.

GAMMA, E. et al. **Design Patterns:** reusable elements of object oriented design. Reading: Addison-Wesley, 1994. 416p.

GRAND, M**. Patterns in Java:** a catalog of reusable design patterns with UML. [S.l.]: John Wiley & Sons, 1998.

GROSSO, W. Dynamic Design Patterns in Objective-C. **Dr.Dobb's Journal**, New York, v.22, n. 8, p.38-44, Aug. 1997.

GRUNE, D.; BALl, H.; JACOBS, C. **Projeto Moderno de Compiladores**: implementação e aplicações. Rio de Janeiro: Campus, 2001. 671p.

GUÉHÉNEUC, Y.; JUSSIEN, N. Using Explanations for Design Patterns Identificaiton. In: WORKSHOP ON MODELLNG AND SOLVING PROBLEMS WITH CONSTRAINTS, 2001, Santa Bárbara, Califórnia, Estados Unidos. **Proceedings…** [S.l.:s.n.], 2001. p.296-303.

HORSTMANN, C; CORNELL, G. **Core Java 2**: fundamentals. [S.l.]: Prentice Hall, 2000. 832p.

HORSTMANN, C; CORNELL, G. **Core Java 2**: advanced features. [S.l.]: Prentice Hall, 2001. 1232p.

JIAZHONG, Z.; ZHIJIAN, W. NDHORM: An OO Approach to Requirements Modeling. **ACM SIGSOFT Software Engineering Notes**, New York, v.21, n. 5, p.65-69, Sept. 1996.

JOHNSON, R. Documenting Frameworks Using Patterns. In: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 7., 1992, Vancouver, Canadá. **Proceedings...** New York: ACM Press, 1992.

JOHNSON, R. **Design Patterns**. Departament of Computer Science at University of Illinois in Urbana Champaign. Disponível em: <http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic?DesignPatterns> >. Acesso em: 04 dez. 1998.

KRÄMER, C.; PRECHELT, L. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 1996. **Proceedings...** New York: IEEE CS Press, 1996. p. 208-215.

LAUDER, A.; KENT, S. Precise Visual Specification of Design Patterns. In: ECOOP - EUROPEAN CONFERENCE OBJECT-ORIENTED PROGRAMMING, 12., 1998, Brussels, Belgium. **Proceedings...** New York: ACM Press, 1998. p. 114-134.

MAES, P. Concepts and Experiments in Computational Reflextion. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE, 1987. **Proceedings...** New York: ACM Press, 1987.

MCCLUSKEY, G. **Using Java Reflection.** Disponível em: <http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/>. Acesso em: 05 maio 1999.

MÜLLER, H. et al. Reverse Engineering: a Roadmap. In: CONFERENCE ON THE FUTURE OF SOFTWARE ENGINEERING, 2000. **Proceedings...** New York: ACM Press, 2000.

NIEMEYER, P.; PECK, J. **Exploring Java**. [S.l.]: O'Reilly & Associates, 1997. 614p.

PARR, T. et al. **ANTLR Reference Manual**: ANTLR version 2.7.1. Disponível em: <http://www.antlr.orgl>. Acesso em: 04 abr. 2001.

PARR, T. **What's An ANTLR ?** . Disponível em: <http://www.antlr.orgl>. Acesso em: 04 abr. 2001.

PETERS, J. **Engenharia de Software:** teoria e prática. Rio de Janeiro: Campus, 2001. 602p.

PRECHELT, L. et al. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. **IEEE Transactions on Software Engineering**, New York, v.27, n.12, p.1134-1144, Dec. 2001.

PREE, W. **Design Patterns for Object-Oriented Development**. Reading: Addison-Wesley, 1995. 268p.

PRESSMAN, R. **Engenharia de Software**. São Paulo: Makron Books, 1995. 1056p.

PRESSMAN, R. **Software Engineering**: a practitioner's approach. [S.l.]: McGraw-Hill, 2001. 850p.

RAO, R. From the Broad Notion of Reflection to the Engineering Practice of Object-Oriented Metalevel Architectures. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS LANGUAGES AND APPLICATIONS, 1991, Vancouver. **Proceedings...** New York: ACM Press, 1991.

RIEL, A.J. **Object-Oriented Design Heuristics**. Reading: Addison-Wesley, 1996. 379p.

RUMBAUGH, J. et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Campus, 1994. 676p.

SCHNEIDEWIND, N. The State of Software Maitanance. **IEEE Transactions on Software Engineering**, New York, v.13, n.3, p.303-310, May 1987.

SEEMANN, R.; WOLFF von GUDENBERG, J. Pattern-Based Design Recovery of Java Software. In: SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 6., 1998, Orlando, FL, USA. **Proceedings...** New York: ACM Press, 1998. p. 10-16.

SINGH, A. **A Practical Use of Java's Reflection API**. Disponível em: <http://www.asingh.net/technical/reflection.html >. Acesso em: 10 set. 2001.

SOMMERVILLE, I. **Software Engineering**. New York: Addison-Wesley, 1996. 592p.

SULLIVAN, G. Aspect-Oriented Programming Using Reflection and MetaObject protocols. **CACM – Communications of the ACM**, New York, v.44, n. 10, p.95-97, Oct. 2001.

TOPLEY, K. **Core Java Foundations Classes**. [S.l.]: Prentice Hall, 1998. 1300p.

TREMBLETT, P. Java Reflection. **Dr.Dobb's Journal**, New York, v.23, n. 1, p.36-39, Jan. 1998.

VLISSIDES, J. Composite Design Patterns. **C++ Report**, New York, v.10, n. 6, p.1-6, June 1998.

WATERS, R.; CHIKOFSKY, E. A Reverse engineering: progress along many dimensions. **ACM SIGSOFT Software Engineering Notes**, New York, v.37, n. 5, p.22-25, May 1994.

WILD, F. Instantiating Code Patterns - Patterns Applied to Software Development. **Dr.Dobb's Journal**, New York, v.18, n. 6, p.45-48, June 1996.

ZANCAN, J. **Um Estudo sobre Requisitos de Ferramentas de Apoio a Instanciação de Frameworks.** 1997. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.