216983-1

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Um Sistema de Apoio ao Teste de Aplicações Smalltalk

por

IVETE MARTINS PINTO

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de
Mestre em Ciência da Computação

SABi
UFRGS
05232106

Profa. Dra. Ana Maria de Alencar Price
Orientadora

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

Porto Alegre, maio de 1998.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Pinto, Ivete Martins

Um Sistema de apoio ao teste de aplicações Smalltalk / por Ivete Martins Pinto. - Porto Alegre: CPGCC da UFRGS, 1997.

83f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientadora: Price, Ana Maria de Alencar.

1. Orientação a Objetos. 2. Reflexão Computacional. 3. Teste de Software. 4. Teste de Software Orientado a Objetos. 5. Teste de Aplicações Smalltalk.
I. Price, Ana Maria de Alencar. II. Título.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

N.º CHAMADA	N.º REG:
681.32.063(043) P659S	34530
	13/08/98
ORIGEM: D / DATA: 29/07/98	PREÇO: R$ 20,00
FUNDO: II	FORN.: II

Engenharia de Software - SBC
Engenharia de Software
Testes de Software
Orientação a Objetos
SMALLTALK
CNPq 1.03.03.00-6

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profa. Wrana Panizzi

Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann

Diretor do Instituto de Informática: Prof. Roberto Tom Price

Coordenadora do CPGCC: Profa. Carla Maria Dal Sasso Freitas

Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira

Agradecimentos

Sempre imaginei ser uma tarefa extremamente fácil escrever um agradecimento. Quando principiei a esboçar esta página constatei que, para agradecer nominalmente a todas as pessoas que colaboraram, direta ou indiretamente, para que eu tivesse condições de desenvolver este trabalho, necessitaria de muito cuidado para não cometer injustiças, pois são muitas as pessoas a quem devo isto. Portanto, quero inicialmente fazer um agradecimento a todos aqueles que, embora não tenham sido citados, merecem minha gratidão.

Devo agradecer à CAPES, pelo financiamento desta pesquisa e à FURG, pelo apoio institucional. Particularmente aos colegas do CPD pelo incentivo e apoio.

Aos colegas de mestrado, especialmente à Analia, sempre incansável, que se mostrou uma boa amiga, contribuindo com dicas, incentivo pessoal e intelectual, além do suporte técnico, e à Juliana, pelo auxílio e amizade.

Não poderia deixar de citar a Oni, inseparável companheira durante estes anos de muitas discussões e trabalho conjunto. Sem sua amizade tudo teria sido mais difícil.

Sou muito grata à Profa. Ana Price por ter sido muito mais que uma orientadora, sempre atenta e solícita. Sua atenção especial a cada um de seus orientandos a torna uma pessoa única.

A todos os funcionários do II que colaboraram para tornar esta etapa de minha vida mais fácil e amena, meu muito obrigado.

Minha família, a quem muito devo. Minhas irmãs, especialmente Míriam, e minha mãe Teresa, sempre maravilhosas e disponíveis.

Aos meus sogros pelo "suporte habitacional", e à minha cunhada Viviane pela invasão de seu "*habitat*", um agradecimento especial.

Ao meu esposo, Marino, que foi pai e mãe em nosso lar durante este período, tentando suprir minha ausência. Obrigada por tudo...

Às minhas filhas, Gabriela e Mônica, que suportaram bravamente a falta da mamãe em casa, sempre mostrando-se muito mais fortes e responsáveis que o esperado...

Obrigada a todos.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Bibliotecas da UFRGS

34530

681.32.063(043)
P659S

INF
1998/216983-1
1998/08/13

Dedico este trabalho,
às minhas filhas Gabriela e Mônica.
cujo amor me dá forças para lutar,
ao meu esposo, Marino,
que, à sua moda, sempre me incentivou,
e à minha mãe, Teresa,
que me apoiou em toda minha vida.

Sumário

Lista de Abreviaturas 7

Lista de Figuras 8

Lista de Tabelas 9

Lista de Códigos Smalltalk 10

Resumo 11

Abstract 12

1 Introdução 13
1.1 O Problema 14
1.2 A Proposta 15
1.3 Estrutura do texto 16

2 Orientação a Objetos 18
2.1 Conceitos Básicos 19
2.1.1 Objeto 19
2.1.2 Encapsulamento 19
2.1.3 Classe de Objetos 20
2.1.4 Herança 20
2.1.5 Polimorfismo 20
2.1.6 Ligação dinâmica 21
2.2 Frameworks 21

3 Teste de Software OO 22
3.1 Teste de Software 22
3.1.1 Classificação dos Testes 23
3.1.2 Etapas no Processo de Teste 23
3.1.3 Ferramentas para Teste de Software 24
3.2 Características do Teste Orientado a Objetos 26
3.3 Estratégias de Teste Orientado a Objetos 29
3.3.1 Teste de Unidade de Fiedler 30
3.3.2 Teste Baseado em Estados de Binder 32
3.3.3 Teste Baseado em Estados de Turner 35
3.3.4 Teste Incremental Hierárquico de Classes de Harrold 37
3.4 Ferramentas para Teste de Software OO 40

4 Reflexão Computacional 43

4.1 Reflexão Computacional no Paradigma OO 44

4.1.1 Reflexão Estrutural - Modelo de Meta-Classes 44

4.1.2 Reflexão Comportamental - Modelo de Meta-Objetos 45

4.2 Framework *Luthier MOPs* 47

4.2.1 Estrutura e Funcionalidades 48

5 *ATeste - Uma Ferramenta de Apoio ao Teste* 51

5.1 Teste Baseado em Estados 51

5.2 Características Reflexivas 52

5.3 Características de *ATeste* 53

5.4 Estrutura Geral 54

5.4.1 Classes do Smalltalk/VisualWorks 55

5.4.2 Classes do Framework *Luthier MOPs* 56

5.4.3 Classes do protótipo 56

***5.5* Descrição das classes de *Ateste* 57**

5.5.1 Interface com o Usuário - Classe *InterfaceDeTeste* 57

5.5.2 Manipulação de Informações - Classe *InfoTeste* 61

5.5.3 Gerenciamento de Meta-Objetos - Classe *MetaGerente* 63

5.5.4 Interceptação de Mensagens - Classe *MetaTeste* 64

5.5.5 Busca do Estado do Objeto- Classe *EstadoDoObjeto* 65

5.6 Funcionamento de *ATeste* para um Estudo de Caso 67

6 Conclusões 77

6.1 Contribuições 77

6.2 Extensões Futuras 78

6.3 Considerações Finais 79

Bibliografia 80

Lista de Abreviaturas

cap.	Capítulo
EUA	Estados Unidos da América
fig.	Figura
MEF	Máquina de Estados Finitos
MO	Meta-Objetos
NSA	National Security Agency
OO	Orientado a Objetos
SEATC	Software Engineering Applied Technology Center
tab.	Tabela

Lista de Figuras

FIGURA 1.1 - Estrutura do texto da dissertação 17
FIGURA 2.1 - Caminho Típico de Invocações de Mensagens 19
FIGURA 3.1 - Níveis de Teste de Software 24
FIGURA 3.2 - Diagrama de Fluxo de Dados para Verificação de Resultados 31
FIGURA 3.3 - Modelo para a Classe Conta 33
FIGURA 3.4 - Diagrama de Transição de Estados da Classe Conta 34
FIGURA 3.5 - Árvore de Transição da Classe Conta 35
FIGURA 3.6 - Diagrama de Estado (MEF) 35
FIGURA 3.7 - Técnica de Modificação Incremental 38
FIGURA 3.8 - Decomposição da Hierarquia de Herança 39
FIGURA 4.1 - Arquitetura Reflexiva 43
FIGURA 4.2 - Reflexão Estrutural 44
FIGURA 4.3 - Reflexão Comportamental 45
FIGURA 4.4 - Arquitetura Reflexiva de Meta-Objetos 46
FIGURA 4.5 - Relacionamento entre Objetos, Meta-Objetos e Gerenciadores de Meta-Objetos 47
FIGURA 4.6 - Diagrama de Objetos do Framework *Luthier MOPs* 50
FIGURA 5.1 - Notações de Classes utilizadas em ***ATeste*** 54
FIGURA 5.2 - Diagrama de Objetos de ***ATeste*** 55
FIGURA 5.3 - Classe InterfaceDeTeste 57
FIGURA 5.4 - Classe InfoTeste 61
FIGURA 5.5 - Classe MetaGerente 63
FIGURA 5.6 - Classe MetaTeste 64
FIGURA 5.7 - Classe EstadoDoObjeto 65
FIGURA 5.8 - Menu Principal do VisualWorks 68
FIGURA 5.9 - Menu da Opção Tools 68
FIGURA 5.10 - Janela de Comunicação Principal de ATeste 69
FIGURA 5.11 - Mostra Métodos das Classes Selecionadas 70
FIGURA 5.12 - Comunicação de Ativação da Reflexão 70
FIGURA 5.13 - Ativação da Aplicação de Controle de Cheques 72
FIGURA 5.14 - Aplicação de Controle de Cheques 72
FIGURA 5.15 - Gravação de um cheque 73
FIGURA 5.16 - Caminho de Execução dos Métodos Selecionados para Teste 75
FIGURA 5.17 - Caminho de Execução da Aplicação 76

Lista de Tabelas

TABELA 3.1 - Vantagens e Desvantagens do Teste OO 27
TABELA 3.2 - Questões comuns sobre Teste OO 27

Lista de Códigos Smalltalk

LISTAGEM 5.1 - Método *inicializa* 58
LISTAGEM 5.2 - Método *incluiCateg* 58
LISTAGEM 5.3 - Método *verificaCateg* 59
LISTAGEM 5.4 - Método *selecClasses* 60
LISTAGEM 5.5 - Método *selecMétodos* 60
LISTAGEM 5.6 - Método *ativaMO* 62
LISTAGEM 5.7 - Método *desativaMO* 62
LISTAGEM 5.8 - Método *findMetaObjectFor:* 63
LISTAGEM 5.9 - Método *activateMetaObjects:forMessage:* 64
LISTAGEM 5.10 - Método *handleMessage:* 64
LISTAGEM 5.11 - Método *buscaEstado:* 66
LISTAGEM 5.12 - Método *montaCaminhoMonitorado:e:refMsg:* 67
LISTAGEM 5.13 - Método *montaCaminhoExecução:e:* 67

Resumo

A atividade de teste é de extrema importância em qualquer ambiente de programação para gerar produtos de qualidade e de alta confiabilidade. Entretanto, além das dificuldades inerentes a esta tarefa, soma-se a falta de metodologias e ferramentas que a apoiem. Isto faz com que o teste se torne uma das atividades do ciclo de vida do software com maior custo e tempo associados.

O paradigma de orientação a objetos surgiu na busca constante dos desenvolvedores de software por maior qualidade e novas soluções para os problemas de baixa produtividade. Todavia, as características poderosas deste paradigma geraram um conjunto adicional de problemas nas atividades de teste e manutenção, onde a alta concentração de custo e tempo envolvidos sempre foram uma constatação real.

Em uma aplicação orientada a objetos, cada objeto é uma entidade que possui um comportamento bem definido, com um papel bem claro no domínio. Os valores dos atributos de um objeto definem o estado do objeto, e são acessados através de um conjunto de mensagens bem definidas, através das quais o objeto é capaz de executar alterações nestes valores de atributos. Os valores de atributos reagem à passagem destas mensagens (ativação de métodos) alterando ou não o seu estado.

Neste trabalho é apresentado um protótipo de ferramenta de apoio ao teste de programas orientados a objetos, chamado ***ATeste***, fundamentado no conceito de teste baseado em estado, com uma abordagem reflexiva, para auxiliar o teste de aplicações Smalltalk.

O teste baseado em estado verifica as interações entre os objetos pela monitoração das mudanças que ocorrem em valores dos atributos. Este tipo de teste verifica como um objeto reage à ativação de um método, respeitando o encapsulamento.

A abordagem reflexiva possibilita que mecanismos de análise possam ser associados à aplicação através da utilização de conceitos de reflexão computacional. A reflexão computacional proporciona a análise dinâmica da aplicação, sem interferir nos aspectos comportamentais e estruturais dos objetos do domínio, permitindo a monitoração de objetos específicos do domínio, os quais podem ser dinamicamente selecionados pelo usuário.

Como suporte para o desenvolvimento de ***ATeste*** foi utilizado o framework *Luthier MOPs*. Este framework apresenta, como principal contribuição, a utilização de técnicas de reflexão computacional baseadas no conceito de gerenciadores de meta-objetos para a implementação de meta-arquiteturas visando a captura de informações e análise de aplicações.

Baseado nesta arquitetura reflexiva, ***ATeste*** trabalha com um meta-nível, que monitora os objetos selecionados pelo usuário na aplicação do nível-base. Desta forma, uma das principais características do sistema desenvolvido é a análise da aplicação sob teste de forma dinâmica sem necessidade de instrumentação de seu código fonte. Isto é possível através da utilização de reflexão computacional, que permite implementar diferentes mecanismos de análise que podem ser dinamicamente associados com a aplicação em execução.

O projeto de ***ATeste*** compreende um conjunto de classes próprias que definem a interface com o usuário e as funcionalidades desejadas, bem como classes do framework *Luthier MOPs*, e especializações destas.

Palavras-chave: Orientação a Objetos, Reflexão Computacional, Teste de Software, Teste de Software Orientado a Objetos, Teste de Aplicações Smalltalk.

TITLE: "SUPPORT'S SYSTEM OF THE SMALLTALK APPLICATIONS TEST"

Abstract

The activity of testing is very important for generating high quality and reliable products in any programming environment. Besides the difficulties inherent to this task, there is the lack of methodologies and tools to support the testing process. These facts make testing one of the software life cycle activities of greatest associated cost and time.

The object oriented paradigm had appeared as a result of the constant searching by software developers for both greater quality and new solutions to the problems of low productivity. However, the powerful characteristics of this paradigm have generated an additional set of problems in the test and maintenance activities, where high costs and time expenditure have been always of great concern.

In an object oriented application, each object is an entity that has a well-defined behavior, with a well clear role in the domain. The attribute values of an object define the object state. Object's attributes may only be accessed through a set of proper messages, which can make changes to object's states.

This work presents a prototype of a tool, named ***ATeste***, which aims to support object oriented testing of Smalltalk applications by implementing the concept of state-based testing through a reflexive approach.

In state-based testing objects interactions are verified through monitoring changes that occur to the values of object attributes. This type of test verifies how an object reacts by a method activation, taking into account object encapsulation.

The reflexive approach allows the association of analysis mechanisms to the software system through the application of computational reflection concepts. Computational reflection provides dynamic analysis of the software, without interfering in both behavior and structural aspects of the object's domain. It allows monitoring of specific objects in the domain, which can be dynamically selected by the user.

The framework *Luthier MOPs* has been extended to support the development of ***ATeste***. This framework provides, as its main contribution, facilities of computational reflection techniques based on meta-objects management concepts for the implementation of meta-architectures, aiming the capture of information and analysis of object-oriented applications.

Considering this reflexive architecture, ***ATeste*** works with a meta-level, which manages the objects selected by the user in the application basic-level. In this way, one of the main characteristics of the developed system is the analysis of the application under test in a dynamic way, without necessity of instrumenting its source code. This is possible due to computational reflection that allows the implementation of particular analysis mechanisms that can be dynamically associated with the application in execution.

The project of ***ATeste*** includes a set of particular classes that define the interface with the user, another set for implementing the elected testing strategy, and some classes specialized from the framework *Luthier MOPs*.

Keywords: Object Orientation, Computational Reflection, Software Test, Object Oriented Software Test, Smalltalk Applications Test.

1 Introdução

O desenvolvimento de software é um processo difícil e consumidor de tempo. Por este motivo os desenvolvedores de software estão constantemente procurando novas soluções para os problemas de baixa produtividade e alto custo, na forma de ferramentas, técnicas e métodos.

O Teste de Software tornou-se, pouco a pouco, um tema de grande importância devido à necessidade da aplicação de métodos práticos que assegurem a qualidade dos produtos finais, a fim de torná-los confiáveis e de fácil manutenção. No entanto, segundo [HET87], a maturidade ainda está longe de ser atingida, pois não existe sequer unanimidade quanto à definição do que seja teste de software. Enquanto que a literatura, normalmente, define teste como "o processo de experimentar ou avaliar um sistema por meios manuais ou automáticos, de modo a verificar se o sistema atende às necessidades especificadas ou a identificar as diferenças entre os resultados esperados e reais", Glen Myers [MYE79] diz que "*Teste é o processo de executar um programa com a intenção de encontrar erros*". Myers acrescenta que "*a maioria das pessoas tem uma visão errada do que seja teste, e que este é o motivo principal para a má qualidade dos testes*". Diz ainda que não se deve testar um programa para adquirir confiança no fato de que ele funciona, mas, pelo contrário, deve-se partir da premissa de que o programa contém erros, e testá-lo para encontrar o maior número possível destes.

O processo de teste geralmente ocupa cerca de metade do cronograma de desenvolvimento do sistema, dependendo do cuidado com que tenham sido executadas as atividades iniciais de análise, projeto e programação. Mesmo no caso de terem sido executadas tarefas exemplares de análise, projeto e programação, é necessário que seja despendido algum tempo e esforço na atividade de teste para verificar se o produto contém erros.

As técnicas e métodos baseados em Orientação a Objetos surgiram trazendo um enfoque substancialmente diferente dos métodos tradicionais. Uma grande vantagem da abordagem Orientada a Objetos é que esta adota formas mais próximas dos mecanismos humanos para gerenciar a complexidade inerente ao software, como a abstração, classificando elementos (objetos) em grupos (classes), através de uma estrutura hierárquica [COA93]. Neste paradigma, o mundo real é visto como consistindo de objetos autônomos, concorrentes, que interagem entre si. Cada objeto tem seu próprio estado e comportamento, semelhantes a seu correspondente no mundo real [KUN91]. Esta abordagem encontra justificativa na facilidade de manutenção, em virtude de uma menor mutabilidade do domínio em relação às funções, na reutilização, visto que esta constância de domínio gera descrições reutilizáveis para outras aplicações do mesmo domínio, na facilidade de validação, pois os requisitos são mais claros às pessoas que atuam no domínio graças à reutilização, e na transição natural de uma fase para outra por fornecer uma visão do conjunto em todas fases do desenvolvimento do software [COA92] [CAS94].

A indústria de software está rapidamente aumentando sua aceitação pelo paradigma baseado em Orientação a Objetos. Todavia, as características poderosas deste novo paradigma introduziram um novo conjunto de problemas no teste e manutenção de software [KUN95], tais como: a dificuldade no entendimento do código detalhado, do teste de comportamento do estado do objeto e a interdependência complexa, principalmente em função do encapsulamento, da hierarquia de classes, do polimorfismo e da ligação dinâmica de mensagens; aliados à falta de ferramentas de teste adequadas para os programas orientados a objetos (OO).

Segundo Fayad e Tsai [FAY95], testes extensivos, verificação e validação durante o desenvolvimento do software são essenciais para a construção de softwares

confiáveis. Verificação e validação de especificações e de programas OO são, contudo, tópicos de pesquisas recentes.

Embora tenha melhorado, significativamente, nos últimos anos e anseie por avanços a curto prazo, a automação dos processos de teste orientado a objetos necessitará ainda conviver com as limitações de uma disciplina de engenharia emergente [BIN95a].

Tendo em vista o exposto acima, torna-se clara a necessidade da pesquisa de técnicas e/ou métodos que auxiliem o desenvolvedor de software na fase de teste de sistemas OO, devido, principalmente, à importância dos testes para a qualidade do software e à propagação da utilização do paradigma da orientação a objetos no desenvolvimento de sistemas computacionais [VIE97].

1.1 O Problema

Um dos problemas, constantemente citado quando se discute teste de software, é o alto custo. Segundo [PRE95], a atividade de teste é um elemento crítico da garantia de qualidade de software, e pode consumir até 40 % do esforço despendido no desenvolvimento de software

Este alto custo se deve ao fato que nas últimas décadas o teste de software vem assumindo uma importância crescente em decorrência de aspectos técnicos e econômicos. Passou-se a dar ênfase a testes mais criteriosos por parte dos usuários e gerentes de desenvolvimento em vista da deficiência dos sistemas de computador existentes e do alto custo e impacto causados pela recuperação dos problemas gerados [HET87].

Teste é uma área da Engenharia de Software na qual a distância entre a pesquisa de conhecimentos e a prática atual é muito grande. Uma das razões muitas vezes citadas é a quantidade de esforço extra necessário para o desenvolvimento da infra-estrutura necessária para um processo de teste compreensivo [McG94], tornando-o tedioso, pois requer a determinação dos dados para teste e dos resultados esperados, a execução do programa com os dados selecionados e a análise dos resultados computados.

Embora a abordagem Orientada a Objetos apresente várias vantagens em relação ao paradigma procedimental, já citadas na seção anterior, a realização de testes é um dos maiores problemas encontrados no desenvolvimento de códigos OO. É discutível se o teste orientado a objetos é essencialmente diferente do teste convencional. Entretanto, não é questionável que a realização de testes de códigos OO é mais complexa, principalmente, em função da hierarquia de classes, do polimorfismo, e da ligação dinâmica de mensagens, que introduzem complicações no entendimento do código detalhado.

Devido à presença de polimorfismo, um atributo pode ter mais de um tipo de valor e uma operação pode ser implementada por mais de um método, de mesmo nome, mas com alguma diferença na sua funcionalidade. Desta forma a escolha da implementação a ser invocada é determinada em tempo de execução pela verificação de qual instância da classe está recebendo a mensagem naquele momento (ligação dinâmica). Outro fator que altera a necessidade e o foco dos testes é a herança, que apesar do grande benefício de reuso de software, faz com que os métodos herdados de uma superclasse devam ser retestados, porque os testes usados no contexto da superclasse podem não incluir todos os casos de teste que podem ocorrer no contexto da subclasse [KUN91] [VIE97].

Como os sistemas OO não tem uma linha específica de controle, técnicas tradicionais de teste, que tentam examinar todos os caminhos através do sistema, não são diretamente aplicáveis. O estado da arte e a prática do teste de programas OO tem

evoluído significativamente nos últimos anos, embora ainda existam diversos problemas primários a serem resolvidos: definição de padrões amplamente aceitos para serem embutidos no teste, estratégias práticas de processos para aumentar a qualidade dos testes, e desenvolvimento da automação do teste [BIN95a].

1.2 A Proposta

Face ao contexto apresentado acima, não é surpresa que muitas tentativas estejam sendo realizadas para automatizar o processo de teste OO, principalmente porque, além das dificuldades já citadas, nota-se também uma carência de metodologias e ferramentas que auxiliem esta fase final do desenvolvimento de software.

Conclui-se então, que ferramentas de apoio à análise de aplicações OO são necessárias para auxiliar o trabalho de teste destas aplicações, maximizando os benefícios da abordagem OO.

O presente trabalho apresenta diversas propostas de estratégias de teste Orientadas a Objetos encontradas na literatura, e um protótipo de ferramenta, ***ATeste***, que tem como objetivo auxiliar o desenvolvedor no teste de códigos OO. ***ATeste*** foi implementado em Smalltalk-80, e fornece auxílio para o teste de aplicações Smalltalk, através da análise dinâmica da aplicação. Uma das principais características do sistema desenvolvido é a análise da aplicação de forma dinâmica, sem necessidade de instrumentação de seu código fonte. Isto é possível através da utilização de reflexão computacional, que possibilita a intervenção na computação da aplicação.

Reflexão computacional é a capacidade de um sistema computadorizado realizar operações sobre sua própria execução, sendo capaz de analisar e modificar as entidades que definem seu próprio comportamento. A análise reflexiva permite implementar diferentes mecanismos de análise que podem ser dinamicamente associados com a aplicação em execução, sem interferir nos aspectos comportamentais ou estruturais dos objetos de seu domínio, possibilitando a monitoração de objetos específicos do domínio, os quais podem ser selecionados pelo usuário em tempo de execução.

Para facilitar a utilização deste conceito, ***ATeste*** foi construído usando como base o framework *Luthier MOPs* [CAM97], projetado para suportar a construção de ferramentas de análise dinâmica de programas através de técnicas de reflexão computacional baseadas em meta-objetos.

ATeste trabalha com um meta-nível, que monitora os objetos solicitados pelo usuário da aplicação do nível-base. Assim, quando uma mensagem, enviada a um objeto selecionado para monitoração, for interceptada durante o processo de execução, o gerenciador de meta-objetos faz com que a ocorrência deste evento cause uma interrupção no processo de execução, transfere o controle para um meta-objeto do meta-nível, que realiza as computações especificadas e retorna o processo de execução.

Para utilização do framework *Luthier MOPs* pela ferramenta proposta, foi necessária a composição de um conjunto de meta-objetos. Estes meta-objetos monitoram a execução das aplicações, capturando as informações requeridas para posterior visualização e análise.

Os conceitos de teste de programas OO baseado em estados [BIN95b] [TUR93] [TUR93a] [TUR93b] foram de grande valia para o desenvolvimento do protótipo proposto.

O teste baseado em estados é uma técnica baseada em modelos de estado. Nestes modelos os objetos são caracterizados por estados. O estado de um objeto é determinado por valores armazenados em cada uma das variáveis de estado (ou subestados), que constituem sua representação de dados.

A principal ênfase no teste de programas OO baseado em estados é que, como os diferentes valores armazenados no objeto podem influenciar o fluxo de controle, influenciam também o resultado da execução dos métodos, pois estes valores são eventualmente determinados e/ ou usados pelos métodos.

Em ***ATeste*** as classes selecionadas pelo usuário são refletidas em meta-objetos. Sempre que uma mensagem é enviada a uma classe refletida, esta mensagem é interceptada e o estado do objeto antes e após a execução do método ativado é apresentado ao usuário, em tempo de execução, juntamente com outras informações relevantes, tais como o nome do método ativado, da classe que recebeu a mensagem, da classe que enviou a mensagem, e da classe que implementa o método. Deve-se especificar os possíveis estados que o objeto pode ter antes e após a execução do método, para cada método selecionado para teste, a fim de que sejam comparados com os estados gerados durante a execução.

Além disto, ***ATeste*** também apresenta uma visão do caminho de execução da aplicação, até o momento de uma dada interceptação, com todas as informações já descritas acima. Isto faz com que o usuário tenha um conhecimento detalhado dos eventos executados pela aplicação.

1.3 Estrutura do texto

Este documento está organizado em seis capítulos, os quais foram estruturados em classes, de acordo com o paradigma OO, inspirado na estrutura mostrada em [LIS95]. O esquema referente à organização e relacionamento desses componentes é mostrado na fig.1.1.

O domínio da pesquisa é o **teste de aplicações OO**, o modelo de implementação é a **orientação a objetos**, e a arquitetura utilizada é a **reflexão computacional**. As classes referentes a estes três componentes, consideradas como as classes básicas da estrutura deste texto, são discutidas nos capítulos 2, 3 e 4.

O capítulo 2 introduz uma breve revisão sobre conceitos básicos de orientação a objetos, considerados úteis no decorrer do trabalho.

O capítulo 3 apresenta uma análise do problema de teste OO e trabalhos correlatos, herdando conceitos do capítulo 2 (Orientação a Objetos). Neste capítulo é apresentada, também, uma breve introdução sobre alguns fundamentos de teste de software.

A reflexão computacional é discutida em maiores detalhes no capítulo 4, apresentando alguns conceitos que visam auxiliar o entendimento da proposta. É apresentada uma visão geral dos principais conceitos e modelos de reflexão computacional, estendidos para o paradigma OO.

Ainda no capítulo 4, é apresentado o framework *Luthier MOPs*, que implementa um comportamento genérico de grande valia como ferramenta de apoio à análise de aplicações através da utilização da reflexão computacional. Este framework foi utilizado como base para a implementação das características reflexivas no protótipo desenvolvido.

O capítulo 5 herda de todos os componentes descritos acima, especializando, por herança múltipla, os comportamentos desejados para a proposta de ***ATeste***: apoio ao teste de aplicações Smalltalk. São descritos os aspectos de modelagem e implementação das diferentes classes projetadas, bem como a implementação e utilização das classes do framework *Luthier MOPs* na construção do protótipo. São apresentadas, também, suas interfaces e funcionalidades.

O capítulo 6 apresenta as conclusões, tecendo considerações sobre o trabalho desenvolvido, suas contribuições e possíveis extensões futuras.

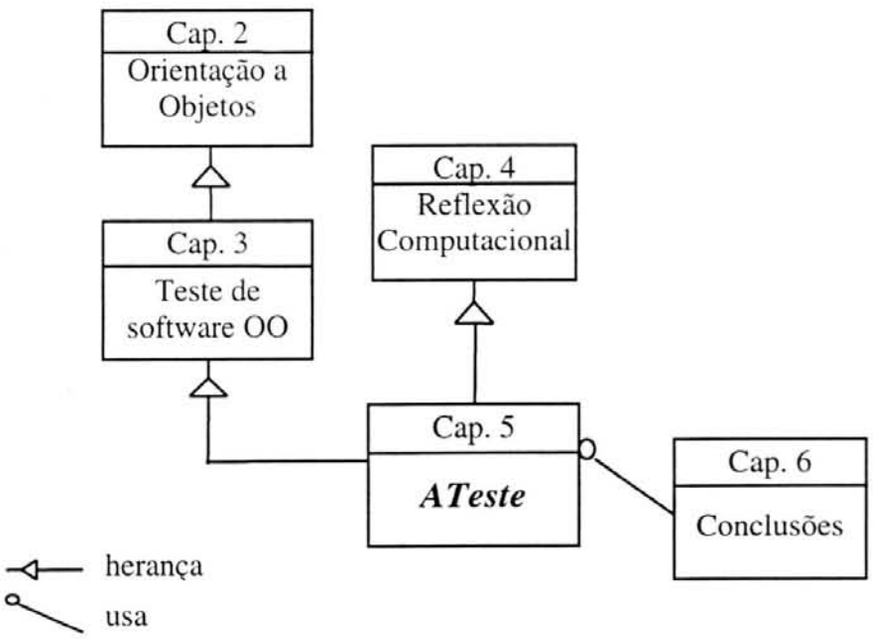

FIGURA 1.1 - Estrutura do texto da dissertação

2 Orientação a Objetos

O desenvolvimento do paradigma OO foi motivado pela busca de soluções para problemas associados com as propostas de desenvolvimento de software orientadas a funções e a dados [KUN95]: pensar em objetos é basicamente diferente de pensar em funções [COA92].

A proposta da abordagem de orientação a objetos, em contraste aos métodos convencionais, que não estão próximos à forma de pensar das pessoas, possui uma visão menos funcional, procurando gerar descrições mais próximas da realidade do domínio do problema. Um sistema OO é composto por um conjunto de objetos que colaboram entre si com a finalidade de alcançar os requisitos do sistema

A orientação a objetos facilita a representação explícita dos pontos comuns do domínio da aplicação através do uso dos mecanismos de estruturação (herança, agregação, servidor/cliente). São elaboradas especificações que facilitam as alterações, que se fazem constantemente necessárias devido às mudanças de requisitos, e existe também o potencial de reutilização dos resultados através da incorporação de resultados anteriores, levando ao aperfeiçoamento do desenvolvimento de sistemas [CAS94].

Conforme [COA92], a abordagem OO encontra justificativa nas seguintes características:

- Facilidade de manutenção, em virtude de uma menor mutabilidade do domínio em relação às funções;
- Reutilização, visto que esta constância do domínio gera descrições reutilizáveis para outras aplicações no mesmo domínio;
- Facilidade de validação, pois os requisitos são mais claros às pessoas que atuam no domínio graças à reutilização;
- Transição natural de uma fase para outra pela visão do conjunto desde a análise até a implementação.

Mas, apesar de todas as vantagens oferecidas por este paradigma, os programas OO são, de forma geral mais difíceis de compreender que os sistemas procedimentais tradicionais. Uma das maiores causas desta dificuldade é que, nos programas OO, deve-se levar em conta a visão dinâmica e a construtiva[1], e os relacionamentos que se estabelecem entre estas, visto que a correspondência entre o modelo de execução e a estrutura do código não é necessariamente direta.

Outro fator que dificulta a compreensão de programas OO é a distribuição do código em diversos pequenos métodos que implementam pequenas parcelas de uma dada funcionalidade. Isto faz com que, para compreender uma simples linha do código, seja necessário seguir uma cadeia de métodos através de uma hierarquia de classes, conforme é demonstrado na fig.2.1 [WIL92].

---

[1] Na visão dinâmica os programas OO são constituídos por objetos que se comunicam através de troca de mensagens, sendo criados e destruídos dinamicamente durante a execução do programa. A visão construtiva utiliza conceitos tais como classes, polimorfismo e herança, que fazem dos programas OO um conjunto de componentes reutilizáveis.

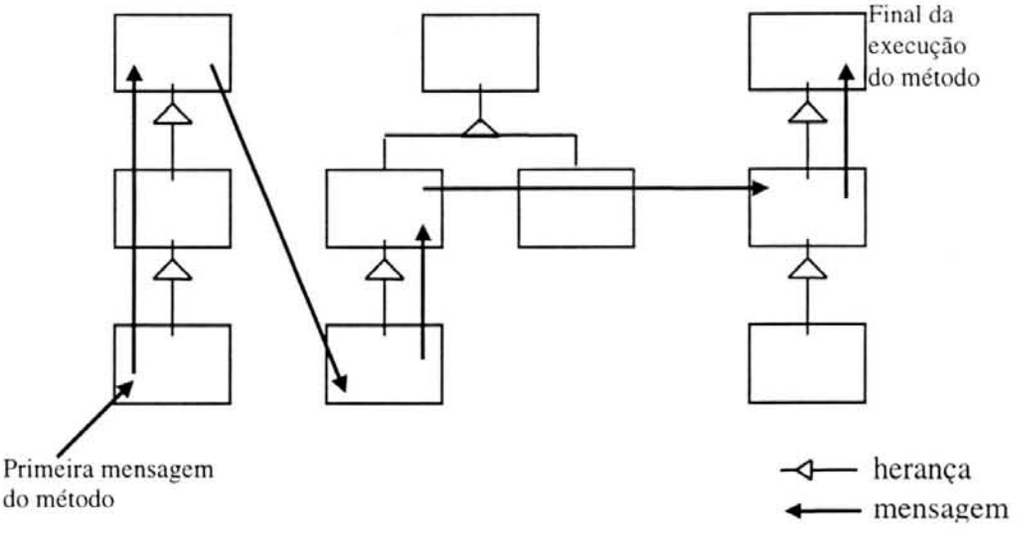

FIGURA 2.1 - Caminho Típico de Invocações de Mensagens

2.1 Conceitos Básicos

Os programas OO são construídos baseados em conceitos tais como classe, polimorfismo e herança. Estes conceitos levam à satisfação dos objetivos dos projetos OO, quais sejam extensibilidade e reutilização.

Desenvolver software OO implica em definir ou reutilizar um conjunto de classes a partir das quais se instanciam os objetos que devem interagir para cumprir a funcionalidade atribuída à aplicação. A herança de classes e a composição de objetos permitem utilizar a funcionalidade definida em outros componentes.

Na próxima seção serão descritos alguns conceitos, baseados em objetos, necessários para o desenvolvimento de sistemas no paradigma OO.

2.1.1 Objeto

Objeto é a entidade básica, real ou abstrata, do domínio da aplicação de um sistema OO, que possui uma identidade que pode ser usada para identificá-lo, ou distingui-lo de outros objetos similares, mesmo que possuam os mesmos atributos.

Um objeto exibe um comportamento bem definido, encapsulando dados (atributos) e as operações (métodos) que estão relacionadas com estes. Isto significa que os dados e as operações de um objeto são modeladas e armazenadas junto com ele.

O acesso aos dados é feito por um conjunto de mensagens bem definidas, que ativam as operações relacionadas com esses dados, impedindo que outros objetos tenham acesso direto a esta informação.

O estado de um objeto é definido pelo conjunto de atributos e os valores que eles possuem [KUN91].

2.1.2 Encapsulamento

Encapsulamento significa modelar e armazenar, junto com o objeto, os atributos e as operações que este objeto é capaz de executar.

O encapsulamento, também chamado de ocultamento de informações, consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, dos detalhes internos na implementação daquele objeto, que ficam ocultos dos demais objetos [RUM97].

As vantagens de um tratamento integrado dos atributos e operações de um objeto são [KUN91]:

- A semântica de um objeto é mais completa porque também descreve quais atributos são alterados por quais operações, e quais são os efeitos destas alterações;
- Suporta ocultamento de informações, facilitando o teste e manutenção de software;
- Facilita o reuso de software porque: a semântica de um objeto é mais clara e fácil de entender; os objetos são livremente acoplados, podendo ser facilmente adicionados ou removidos; e a construção de objetos reusáveis provoca uma menor propagação de alterações, devido ao ocultamento de informações.

2.1.3 Classe de Objetos

Uma classe de objetos descreve um conjunto de objetos com propriedades semelhantes (atributos), com o mesmo comportamento (operações), os mesmos relacionamentos com outros objetos, e a mesma semântica [RUM97].

O agrupamento de objetos em classes leva à abstração de um problema, dando à modelagem poder e capacidade para generalizar a partir de alguns casos específicos para um hospedeiro de casos semelhantes.

Uma classe pode ser definida como uma extensão (generalização) ou restrição (especialização) de outra, gerando uma hierarquia de classes.

Um objeto é uma instância de uma classe. Uma classe é definida sem objetos, que são criados e destruídos dinamicamente durante a execução do sistema.

As classes podem ser definidas como abstratas e concretas. As classes abstratas permitem especificar conhecimento e funcionalidade genérica de um conjunto determinado de entidades, e concretizam-se em subclasses concretas. As classes concretas definem o comportamento de um conjunto de objetos, e sempre compreendem a implementação completa deste comportamento, mesmo que parte dele tenha sido herdado.

2.1.4 Herança

Herança define o relacionamento entre classes, no qual uma classe compartilha a estrutura ou comportamento definido em uma ou mais classes. A nova classe (denominada subclasse) herda todas as propriedades da anterior (chamada superclasse) podendo aumentar ou redefinir a estrutura herdada.

Através deste conceito é possível representar a relação de especialização e generalização entre duas classes. Uma superclasse é uma generalização da subclasse, e a subclasse é uma especialização da superclasse. A superclasse contém características comuns a todas as classes, enquanto as subclasses contém características específicas de cada classe.

Uma das principais vantagens da herança é a reusabilidade do código, minimizando o volume de código a ser desenvolvido para adição de novas funções.

2.1.5 Polimorfismo

Polimorfismo é a habilidade de assumir mais de uma forma: um atributo pode ter mais de um tipo de valor, e uma operação pode ser implementada por mais de um método. Através do polimorfismo duas ou mais classes podem responder à mesma mensagem de formas diferentes.

Quando uma mensagem é enviada, o método a ser ativado depende da classe alvo, pois o mesmo nome pode identificar métodos distintos em uma mesma hierarquia

de classes. Desta forma uma mesma operação pode resultar em diferentes comportamentos em classes distintas.

Este conceito é útil para distinguir mensagens de um método. Um objeto envia uma mensagem, se o objeto alvo implementa um método de mesmo nome, ele poderá responde-la. Diferentes respostas poderão ocorrer, dependendo de como os métodos dos receptores foram implementados.

2.1.6 Ligação dinâmica

É um mecanismo efetivo para implementar polimorfismo pois, sem a ligação dinâmica, o polimorfismo se confunde à sobrecarga de nomes.

Nos programas OO, quando uma mensagem é enviada, o método a ser executado dependerá da classe do objeto que está recebendo esta mensagem. Como um mesmo nome pode identificar métodos distintos definidos através da hierarquia de classes, uma operação pode ter mais de uma implementação. A escolha de qual implementação ativar quando uma mensagem é enviada, é determinada em tempo de execução, no momento em que se conhece a classe do objeto que recebe a mensagem.

2.2 Frameworks

A definição de framework é aqui apresentada tendo em vista a utilização de um framework como base para o desenvolvimento de ***ATeste***.

A reutilização de componentes de software é um dos argumentos a favor da abordagem de orientação a objeto, embora a reutilização não seja uma característica inerente desta abordagem, mas que deve ser obtida a partir do uso de técnicas que produzam software reutilizável. Os frameworks OO são estruturas de classes interrelacionadas, que permitem não apenas reutilização de classes, mas minimizam o esforço para o desenvolvimento de aplicações por conterem o protocolo de controle da aplicação, ou seja, a definição da arquitetura.

Um framework é constituído por classes que definem um projeto abstrato, ou uma infra-estrutura, fornecendo uma solução genérica para os problemas de um grupo de aplicações dentro de um domínio particular, servindo como molde para a construção de aplicações. Framework pode ser definido como o projeto de um conjunto de objetos que colaboram entre si para satisfazer um conjunto de responsabilidades.

Aplicações específicas são desenvolvidas especializando-se as classes do framework para fornecer a implementação de alguns métodos, enquanto grande parte da funcionalidade da aplicação é herdada.

Os componentes de um framework podem ser classificados em abstratos, os quais contém as abstrações principais do domínio, e concretos, que implementam as diferentes materializações dos componentes abstratos. Uma classe abstrata é projetada para ser utilizada como um molde para especificar subclasses, enquanto uma classe concreta é projetada para especificar objetos.

Os métodos podem ser classificados em: abstratos, os quais definem a interface de uma operação mas não fornecem sua implementação, métodos base, que fornecem a implementação completa de um comportamento comum a qualquer aplicação do domínio, métodos *hook*, os quais possuem a implementação *default* que pode ser redefinida, e métodos *template*, que definem algoritmos genéricos que invocam ao menos um método abstrato.

Os frameworks orientados a objetos oferecem um grande potencial para aumentar a produtividade e a qualidade no desenvolvimento de software, no entanto possuem normalmente, como fator limitante, a dificuldade de compreensão para outros usuários que não o projetista do framework.

3 Teste de Software OO

O teste de software é uma questão importante em qualquer ambiente de programação, sendo uma das atividades realizadas para garantir a qualidade do produto. O crescente destaque do software como elemento de qualquer sistema de informação e os custos associados às falhas de software são forças propulsoras para uma atividade de teste cuidadosa e bem planejada [PRE95].

O paradigma de orientação a objetos é uma nova proposta para o desenvolvimento de software que trouxe consigo alguns problemas adicionais para o teste e manutenção de programas. Segundo [KUN95] estes problemas podem ser resumidos nos seguintes aspectos:

- Entendimento de aplicações OO (devido às características de encapsulamento, polimorfismo e ligação dinâmica);
- Interdependência complexa (causada pelos relacionamentos complexos que existem em um programa OO, tais como herança, agregação, associação, criação dinâmica de objetos, polimorfismo, e outros);
- Teste de comportamento dependente de estado (os objetos possuem comportamentos dependentes de seu estado, isto é, o efeito de uma operação em um objeto depende de seu estado e pode alterar este estado);
- Carência de ferramentas de suporte ao teste (o desenvolvimento de ferramentas para suporte ao teste OO ainda é imaturo, e o uso de técnicas convencionais não atendem aos problemas específicos de teste OO).

O teste estrutural procedimental seleciona caminhos para teste (geralmente em um grafo de fluxo de controle que representa o programa) a partir de uma entrada única, alcançando uma única saída. Entretanto, uma classe contém vários pontos de entrada e vários pontos de saída, que são representados por seus métodos. Nos sistemas OO, devido à presença do polimorfismo e da ligação dinâmica, técnicas tradicionais de teste, que tentam examinar todos os caminhos através do sistema, não são diretamente aplicáveis.

Nas próximas seções são apresentadas algumas considerações sobre teste de software antes de serem enfocadas as questões específicas de teste de software OO.

3.1 Teste de Software

Teste é o processo de executar um programa com a intenção de detectar erros, enquanto depuração é o processo de analisar um programa para localizar e corrigir os erros detectados, ocorrendo usualmente após um teste bem sucedido (onde erros são encontrados) [MYE79]. As fronteiras entre teste e depuração não estão claramente definidas, pois algumas estratégias são hábeis para detectar e localizar erros ao mesmo tempo. Por esse motivo, teste e depuração muitas vezes se confundem.

Segundo [PRE95], o crescente destaque do software como elemento dos sistemas de informação e os custos associados às falhas de software são forças propulsoras para uma atividade de teste cuidadosa e bem planejada. Um processo de teste deve fornecer um nível de confiança tal, a fim de que o software encontre, através de condições específicas, seus objetivos; e também deve detectar eventuais falhas no mesmo.

O cerne do teste está na localização de erros no software, pois, ao encontrá-los, tem-se certeza que o teste foi bem sucedido.

3.1.1 Classificação dos Testes

É conveniente classificar os tipos de teste em categorias bem definidas. Uma classificação importante baseia-se na origem dos dados de teste. Dentro dessa classificação [HET87], os tipos principais de teste são os seguintes:

- Teste Funcional
  O teste funcional é a mais conhecida abordagem de teste. O objetivo é verificar se o sistema executa corretamente as funções especificadas nos requisitos do usuário. Os casos de teste são derivados das especificações funcionais que o programa deve realizar. As especificações definem como o programa deve comportar-se sob o ponto de vista externo, determinando as entradas e saídas e enumerando as funções realizadas. Os casos de teste são desenvolvidos e submetidos ao sistema, e as saídas são examinadas para verificar [PRE95]: funções incorretas ou ausentes, erros de interface, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho, e erros de inicialização e término.
  Este tipo de teste independe da lógica da codificação. É também chamado de *caixa preta (black-box)*, pois como a função que o produto executa é conhecida, os testes são conduzidos sem se preocupar com sua estrutura lógica interna.
- Teste Estrutural
  Este tipo de teste deriva da lógica do programa e da estrutura de codificação. Testa os segmentos lógicos de codificação a partir de um conjunto de casos de teste que garantam que: todos os caminhos independentes dentro de um módulo sejam executados pelo menos uma vez, e que todas as decisões lógicas, todos os laços do programa e as estruturas de dados internas sejam executadas. Depende da lógica de codificação, e não possibilita o teste de funções ausentes. É também chamado *caixa aberta (white-box)*, pois propõe-se a examinar os detalhes procedimentais do programa.
- Análise de Mutantes:
  A análise de mutantes é uma técnica baseada em erros para avaliação da adequação de dados de teste. Utiliza um conjunto de dados de teste a fim de verificar sua qualidade, com relação à eficácia na detecção de erros, baseada no resultado da mutação do programa. Para avaliar a adequação de um conjunto de dados **T** em relação a um programa **P**, utiliza um conjunto de programas, com algumas diferenças de **P**, chamados *mutantes*. O objetivo é obter um conjunto de casos de teste que revele diferenças de comportamento entre **P** e seus mutantes.
  A análise de mutantes ocorre em quatro etapas: geração de mutantes, execução de um programa **P** a partir de um conjunto de casos de teste **T**, execução dos mutantes a partir de **T**, e a análise dos mutantes [DEM87].

3.1.2 Etapas no Processo de Teste

Normalmente são realizados pelo menos os níveis de testes descritos abaixo [HET87], e que são apresentados na fig.3.1:

- Teste de unidade, ou teste de programa: Os programas, ou módulos de programas, são testados isoladamente, a fim de assegurar que funcionam apropriadamente como uma unidade. Tem como objetivos principais verificar se a lógica funciona corretamente e se toda a lógica necessária está presente;
- Teste de Integração: É realizado o teste de grupos de programas, ou módulos de programas, integrando as unidades. Deve verificar as interfaces, a reunião dos componentes do sistema e a interação dos módulos;

- Teste de Sistema: A proposta do teste de sistema é comparar o sistema, ou programa, com seus objetivos originais, a fim de verificar o desempenho funcional do sistema e a combinação com os recursos mais abrangentes do sistema (por exemplo, hardware e outros sistemas de informação);
- Teste de Aceitação: É aquele que verifica se as necessidades da organização foram atendidas, e a disponibilidade para implantação ou uso do sistema;
- Teste de Regressão: Verifica o programa após a realização de mudanças, para garantir que não foram introduzidos erros.

FIGURA 3.1 - Níveis de Teste de Software

Existem diversas estratégias de teste de integração, sendo que as duas mais conhecidas são os testes *bottom-up* e os *top-down*. A abordagem *bottom-up* (também conhecida como teste de unidade, teste de módulo ou teste de programa) começa por testar os módulos de forma individual, depois reunindo-os em unidades cada vez maiores para serem testadas em conjunto. Por fim, todos os componentes do sistema são combinados para serem testados.

A abordagem de testes *top-down* principia com um arcabouço do sistema, isto é, presume que os módulos de execução de alto nível do sistema foram desenvolvidos, mas que os módulos de baixo nível existem apenas como simulações. O propósito é apenas começar a testar as interfaces entre os principais subsistemas, aumentando depois a abrangência dos testes até chegar aos detalhes do sistema.

3.1.3 Ferramentas para Teste de Software

As ferramentas de teste podem ser classificadas em dois grupos, de acordo com a análise que realizam, estática ou dinâmica, embora muitos sistemas de teste não utilizem apenas uma destas, mas sim uma combinação das técnicas de análise.

A análise estática provê uma análise limitada do programa, focalizando os requisitos e o aspecto estrutural. As ferramentas de análise estática de teste analisam as características da estrutura do programa sem verificar a executabilidade do programa em teste. Neste grupo se enquadram as proposta dirigidas para o código.

Na análise dinâmica, o teste é realizado pela execução direta do programa que está sendo testado. A análise dinâmica inclui várias operações (tais como *trace*, execução simbólica, simulação, e outras) que se distribuem em quatro categorias, que são avaliadores simbólicos, geradores de dados de teste, instrumentadores de programas e analisadores de mutação de programas.

Existem diversos esforços sendo realizados na pesquisa e desenvolvimento de testes de softwares. Os testes para softwares procedimentais, embora necessitando de adaptações às complexidades peculiares dos softwares OO, ainda continuam sendo válidos.

Por este motivo, faz-se importante citar alguns exemplos de esforços brasileiros relevantes que vêm sendo desenvolvidos na área de teste, tais como:

- PROTESTE (Instituto de Informática/ UFRGS)
  Proteste é um protótipo desenvolvido para suportar o teste estrutural de programas, incluindo critérios baseados no fluxo de controle, tais como todos-nós e todos-ramos, e baseados na análise de fluxo de dados, como Potenciais Usos. Foi desenvolvido para o teste de programas escritos em Pascal, podendo ter seu ambiente ajustado para outras linguagens através da utilização de uma ferramenta que gera analisadores de código fonte específicos para cada linguagem [PRI94] [SIL95].
- POKE-TOOL (EMBRAPA/ CNPTIA, DCA/ FEEC/ UNICAMP e ICMSC/ USP)
  Poke-Tool é uma ferramenta multilinguagem que auxilia a aplicação dos critérios Potenciais Usos no teste estrutural de programas, baseado na análise de fluxo de dados. A versão para a linguagem C foi a primeira a ser desenvolvida. Posteriormente foram geradas versões da ferramenta para linguagens distintas como COBOL, FORTRAN e CLIPPER. Esta ferramenta encontra-se em evolução para permitir sua viabilidade prática em ambientes industriais [CHA97].
- PROTEUM e PROTEUM/IM (IFSC e ICMSC/ USP)
  A ferramenta PROTEUM [DEL93] é um ambiente de teste que apoia a aplicação do critério de Análise de Mutantes, uma técnica de teste baseada em erros, para avaliação e seleção de conjuntos de teste. PROTEUM/IM é uma versão desta ferramenta que implementa um critério de teste de integração, denominado Mutação de Interface. Este critério busca exercitar as interações entre as unidades de um programa, apoiando o teste de integração de software de programas em C [DEL97] [DEL97a].

Deve-se citar, também, outros trabalhos já disponíveis comercialmente, tais como LOGISCOPE [VER92], e TCAT (Test Coverage Analysis Tool) [DRA96], que são ferramentas que auxiliam a verificação da qualidade de software.

LOGISCOPE é uma ferramenta que auxilia o desenvolvimento de softwares de qualidade através do uso de métricas e do teste estrutural. Suas funções se estendem por todo o ciclo de vida do software, e são agrupadas em duas fases: análise estática e análise dinâmica, suportando o teste de programas desenvolvidos em diversas linguagens.

A análise estática verifica o código fonte usando técnicas de representação, tais como, grafos de chamada, que provêm um visão da arquitetura da aplicação, e grafos de controle, fornecendo a representação da estrutura lógica interna dos componentes, bem como métricas de complexidade, que computam medidas relativas à complexidade do programa, às estruturas de controle e às estruturas de chamadas.

O teste de unidade e integração é realizado dinamicamente. LOGISCOPE utiliza uma técnica de instrumentação do código fonte, que realiza a cobertura dos caminhos lógicos durante a execução dos casos de teste. Os testes de unidade são verificados com relação às estruturas de controle de cada componente (blocos de instrução, caminhos de decisão e seqüências de código linear), enquanto os testes de integração avaliam as estruturas de chamada, mostrando os caminhos percorridos durante a execução através de um grafo de chamada dinâmico.

TCAT foi utilizada pela NSA (*National Security Agency*) dos EUA, através do SEATC (*Software Engineering Applied Technology Center*), em um estudo de caso que realizou a comparação de resultados obtidos a partir de um software desenvolvido sem processos de controle, atividades de teste, inspeções de código, etc... e de um novo software, gerado a partir do primeiro, utilizando processos de reengenharia, onde foram

integrados indicadores de qualidade estáticos correlacionados com o teste dinâmico. O SEATC tem analisado medidas efetivas para assegurar a geração de softwares de alta qualidade, as quais já foram aplicadas em cerca de 2,5 milhões de linhas de código desenvolvidas em C, C++, FORTRAN e ADA.

TCAT realiza a análise de cobertura do código, através de um grafo dirigido (grafo de fluxo de controle). Cada nodo representa um comando do código, ligado a outros comandos por setas, que representam os segmentos. A partir do grafo são geradas estatísticas contendo um resumo de informações do código, tais como número de comandos, número de segmentos, tamanho médio de caminho do código, menor e maior caminho, número de laços (interações), tamanho do caminho base (sem interações).

3.2 Características do Teste Orientado a Objetos

O paradigma OO, sendo um paradigma novo e em desenvolvimento, possui diversas questões a serem resolvidas, principalmente no que tange ao teste de software. Alguns fatores comumente discutidos são as implicações da herança, encapsulamento, polimorfismo e ligação dinâmica, entre outros.

Erros de código são comuns em qualquer paradigma de programação, portanto não existem razões que levem a supor que programas OO são mais imunes a erros que aqueles desenvolvidos em linguagens tradicionais. Ironicamente, ao mesmo tempo em que algumas características próprias das linguagens orientadas a objetos tentam reduzir alguns tipos de erros, podem favorecer a adição de novos riscos de falhas [BIN95].

Entre as facilidades introduzidas por este paradigma, pode-se citar o tamanho dos métodos, que tendem a ser pequenos, com algoritmos menos complexos, facilitando a localização de falhas. O encapsulamento previne muitos dos problemas causados por um escopo de dados sem controle.

Uma diferença importante do teste de programas procedimentais em relação aos modelos orientados a objetos, encontra-se no fato que as aplicações OO não são executadas seqüencialmente. Como nenhuma ordem de invocação dos métodos é especificada explicitamente, a análise do fluxo de controle ou do fluxo de dados para o teste de classe é dificultada, fazendo com que técnicas de teste estrutural não sejam diretamente aplicáveis.

A tab.3.1 resume algumas vantagens e desvantagens do teste de programas OO em relação ao teste procedimental [McG96].

Algumas questões comuns sobre teste OO, coletadas a partir de uma análise de diversas propostas atuais para o teste OO e discutidas em [BIN95], são apresentadas na tab.3.2.

TABELA 3.1 - Vantagens e Desvantagens do Teste OO

VANTAGENS	DESVANTAGENS
• As interfaces são bem definidas.	• O encapsulamento das informações torna a avaliação da correteza mais difícil (ocultamento de informações).
• O número reduzido de parâmetros resulta na necessidade de menos casos de teste para sua cobertura.	• Os múltiplos pontos de entrada para um objeto dificultam o gerenciamento do teste.
• A herança provê um guia natural para o reuso de casos de teste.	• O polimorfismo e a ligação dinâmica expandem as possíveis interações entre os objetos.

TABELA 3.2 - Questões comuns sobre Teste OO

QUESTÃO	CONCLUSÃO
Unidade básica para teste	Existe um consentimento quase universal que a classe é a unidade natural para projetos de casos de teste. Os métodos são fragmentos sem sentido de suas classes. Outras unidades para teste são agregações de classes, categorias de classes e pequenos subsistemas.
Implicações da herança	Os métodos herdados devem ser retestados no contexto da subclasse. Herança múltipla aumenta o número de contextos para teste. Se existe uma estrutura de herança usada corretamente para implementar especialização, alguns casos de teste poderão ser usados na subclasse, no entanto, se a herança foi usada como conveniência de implementação, devem ser feitos testes extras.
Implicações do encapsulamento	Esta característica geralmente benéfica das linguagens OO pode dificultar o trabalho de teste, que necessita relacionar o estado concreto e abstrato de um objeto. Um objeto não consegue ver as mudanças de estado produzidas em outro objeto, dificultando a verificação do resultado do teste.
Implicações do polimorfismo	Cada ligação possível de um componente polimórfico requer um teste individual. É difícil localizar todas estas ligações, aumentando as chances de erro e dificultando a correteza.

Teste caixa aberta	Esta proposta baseada na análise ou controle do fluxo de dados pode ser adaptada para teste de métodos, mas não é suficiente para teste de classes. A aplicabilidade das propostas de grafos de fluxo é questionada por diversas fontes. Os problemas principais com este tipo de teste são os numerosos casos de teste e dificuldade de gerar conjuntos de teste adequados a partir do código fonte.
Teste caixa preta	Existe uma concordância geral que os métodos convencionais caixa preta, baseados na especificação, serão úteis para os sistemas OO. Se as especificações são desenvolvidas de forma correta expressando cada detalhe da implementação, apenas um exame do código fonte é necessário. Por exemplo, uma análise da estrutura da classe é necessária para retestar características herdadas. A distância entre a especificação e a implementação orientada a objetos é pequena comparada com os sistemas convencionais.
Teste baseado em estados	Deriva casos de teste pela modelagem da classe como uma máquina de estados. Métodos são transições de estados. O modelo de estados define uma seqüência de transições possíveis, e os casos de teste são criados para exercitar cada transição. Existem algumas dificuldades para realizar o teste baseado em estados: o teste de classes complexas, evitar a violação do encapsulamento e o teste de um sistema de aplicações.
Adequação e cobertura	Algumas propostas chamam de teste "minucioso", o que não é uma definição operacional, exceto para estratégias que vêm do teste convencional. As estratégias existentes definem um escopo geral de teste, mas não uma cobertura específica baseada em uma hipótese de falha provável. Algumas defendem que a cobertura inclui comandos, múltiplas condições e decisões.
Estratégias de integração	A integração das classes deve ser rigorosamente realizada na proposta de desenvolvimento global. Existem duas estratégias básicas: uma consiste em fazer com que todas as classes respondam a um sinal de entrada. Cada classe é testada como uma unidade e, então, o conjunto de classes é exercitado. A outra testa inicialmente as classes que usam poucas, ou não usam classes servidoras, depois as classes que usam o primeiro grupo são testadas, seguidas das classes que usam o segundo grupo, e assim por diante. Em geral, o uso de métodos stubs e *drivers* consegue ser evitado, mas pode ser necessário.
Estratégias do processo de teste	O desenvolvimento OO tende a pequenos ciclos incrementais. Com o teste, o desenvolvimento pode ser caracterizado como um pequeno projeto, um pequeno código e um pequeno teste. As questões sobre este processo incluem: quando fazer o projeto de teste, a estratégia de integração, e estender o teste às características herdadas.

3.3 Estratégias de Teste Orientado a Objetos

Existem basicamente três abordagens de teste no paradigma OO: funcional, estrutural, e de interação [McG94]. O teste estrutural, também conhecido como caixa aberta, difere do teste funcional, também chamado de teste caixa preta, por utilizar a visão interna de investigador em vez da visão externa de juiz que considera como um objeto de boa qualidade aquele que implementa o projeto [HAL93]. Ambos testes são importantes para a boa qualidade do software. O teste caixa preta é importante na verificação do atendimento das funcionalidades requeridas, baseando-se na especificação de requisitos do sistema, e o teste caixa aberta é necessário para assegurar que os desenvolvedores saibam exatamente como o software operará.

Testes caixa aberta são derivados do código do objeto e suas estratégias concentram-se na tentativa de exercitar todo o código do objeto. Segundo [HAL93], as estratégias mais sofisticadas, tais como cobertura de decisão e cobertura de condição, são consideradas as mais completas, pois além de exercitar todo o código tentam exercitar todas as combinações importantes do código. Todavia, estas propostas ainda não estão consolidadas no paradigma OO.

No teste de interações, pode-se considerar as interações entre os métodos de uma classe, e as interações entre métodos de diferentes classes. No teste de interação são executados pares de métodos de uma mesma classe (intra-classe), ou de classes diferentes (inter-classes). Na tecnologia orientada a objetos ocorrem algumas complicações no teste de interações pelo uso da ligação dinâmica.

A seguir é apresentada uma classificação dos níveis de teste para o paradigma OO [SMI92]:

- Teste de Classes
  Este nível de teste é comparável ao teste de unidade da proposta procedimental, embora uma classe tenha um escopo diferente de um simples procedimento. Classes são executadas através da execução de objetos. Cada objeto, por ser totalmente encapsulado, é essencialmente um programa completo com um escopo definido de operações e entradas. A especificação do objeto descreve o resultado desejado de cada operação. Portanto, o processo de verificar o objeto é a comparação dos resultados de uma operação com o resultado desejado, contido na especificação do objeto [HAL93].
  O teste de classe é o primeiro nível do teste de integração, por este motivo, "teste de unidade" no paradigma OO tem um nível de abstração maior que "teste de unidade" no paradigma procedimental. Esta estratégia de teste verifica os métodos intra-objetos, as saídas corretas na interação de procedimentos e os estados dos objetos.
  Após implementado, o objeto pode ser testado. Se os testes são executados em paralelo com as classes, classes melhores são desenvolvidas.
- Teste de *Cluster* (categorias)
  Um *cluster* de classes é um grupo de classes que colaboram entre si. O teste de *cluster* representa o segundo nível de integração. Seu foco é a interação entre as instâncias das classes no *cluster*, sendo que cada classe deve ter sido testada individualmente. Uma especificação deve ser desenvolvida para cada *cluster*, que é similar à especificação da classe.
  Esta estratégia de teste verifica os métodos inter-objetos e saídas corretas na interação de grupos de objetos cooperantes.
- Teste de Sistema

O sistema é testado funcionalmente usando casos de teste derivados dos casos de usos e outros requisitos do sistema. Verifica as respostas do sistema à entradas externas. Fornece ao usuário final uma visão do sistema, e os casos de teste executam ações típicas deste usuário.

Os testes tradicionais (unidade, integração, sistema, aceitação e regressão), apresentados na seção 3.1.2, continuam sendo válidos, mas devem ser adaptados para softwares OO e, neste novo paradigma, não seguem uma seqüência rígida para sua realização.

O teste de unidade é usualmente executado pelos próprios desenvolvedores. Envolve o desenvolvimento de uma base de teste para a classe, que estabelecerá um ambiente simulado em torno da classe a ser testada. É então enviada uma mensagem para a classe e os resultados são analisados de acordo com um conjunto de critérios esperados.

Estratégias de teste OO ainda são imaturas, todavia muitas pesquisas tem sido realizadas. A seguir são apresentadas algumas propostas encontradas na literatura e que foram de grande valia para o amadurecimento e desenvolvimento da proposta apresentada em ***ATeste***.

Segundo Turner e Robson [TUR93], a menor unidade de teste para sistemas Orientados a Objetos é a classe. Estes autores recomendam que seja realizado inicialmente o teste das classes que não possuem dependentes, e então, recursivamente, o teste das classes/objetos que usam seus serviços, juntando desta forma os testes de unidade/integração. Embora, na realidade, sejam os objetos que são testados, não as classes, o mais importante é o efeito da capacidade de uma classe/objeto armazenar informações de estados entre os acessos. Esta estratégia de teste é denominada *Teste Baseado em Estados*.

Fiedler [FIE89] descreve uma proposta baseada na experiência adquirida em um projeto de software da Hewlett Packard, onde foram aplicadas técnicas de teste tradicionais para softwares escritos em C++. As classes são validadas na ordem determinada por um grafo de chamada, sendo que as classes base do grafo são validadas primeiro, da mesma forma que na proposta de Turner.

Harrold e McGregor [HAR92] consideram explicitamente a validação de novas classes, e a revalidação de classes modificadas, almejando minimizar o teste baseando-se na hierarquia de herança pela existência de classes bem-testadas nos níveis mais altos da hierarquia, embora outros autores previnam que a confiança nos testes realizados nas superclasses não eliminam a necessidade de retestar os mesmos serviços nas subclasses derivadas.

3.3.1 Teste de Unidade de Fiedler

Fiedler [FIE89] descreve os processos e experiência obtidos na realização de teste de unidade em módulos desenvolvidos em uma linguagem OO. O projeto aplicou técnicas de teste tradicionais ao teste de software codificado em C++.

O autor justifica sua abordagem: "*já que o paradigma da programação OO enfatiza o comportamento externo de abstrações de dados, dando menor importância ao comportamento interno, esperar-se-ia que apenas técnicas de teste funcional devessem ser empregadas. Entretanto, uma estrutura de teste mais robusta, que empregue o teste de caminho completo, é atualmente necessária*".

Quando uma classe é definida e codificada, ela pode ser considerada uma unidade completa e disponível para ser utilizada por outros módulos do sistema, enquanto que, no ambiente procedimental, a unidade somente existe quando alguma função ou procedimento for implementado.

ambiente procedimental, a unidade existe quando alguma função ou procedimento for implementado.

O teste de unidade proposto por Fiedler necessita de uma especificação bem definida da classe sob teste. As classes são validadas de acordo com a ordem determinada pelo diagrama de chamada do projeto. As classes na base do diagrama são validadas primeiro, sem a necessidade de classes *driver* que emulam classes parametrizadas não validadas.

Quanto ao teste de classes definidas através da herança, os métodos da classe que são herdados da superclasse requerem mínimo esforço de teste, somente as funcionalidades básicas necessitam ser testadas.

A estratégia para desenvolvimento dos casos de teste requer, inicialmente, a determinação dos caminhos de um método que requerem cobertura de teste e, então, a criação dos casos de teste baseados na especificação da classe e no código fonte dos métodos.

A métrica de complexidade ciclomática de McCabe tem um papel chave na implantação desta estratégia para garantir a cobertura do caminho completo de cada método. Através do seu uso é assegurado que todos os caminhos independentes dos métodos da classe são testados.

São gerados casos de teste derivados do intervalo de valores de entrada esperados para os métodos, da análise de limites dos valores e valores errôneos.

Os resultados das execuções de teste são verificadas através da comparação de um arquivo com os resultados esperados, como é mostrado no diagrama de fluxo de dados da fig.3.2. Esta técnica exige que dados de saída esperados sejam gerados manualmente, o que constitui-se em seu maior problema. Entretanto, é uma técnica com fácil aplicação, permitindo que os próprios desenvolvedores possam conduzir o teste, utilizando seu conhecimento do código da unidade.

Esta metodologia foi aplicada no teste de diversas classes genéricas, e os problemas encontrados foram relacionados com a complexidade dos métodos e com o número de comandos do código. O código dos métodos herdados não foi considerado.

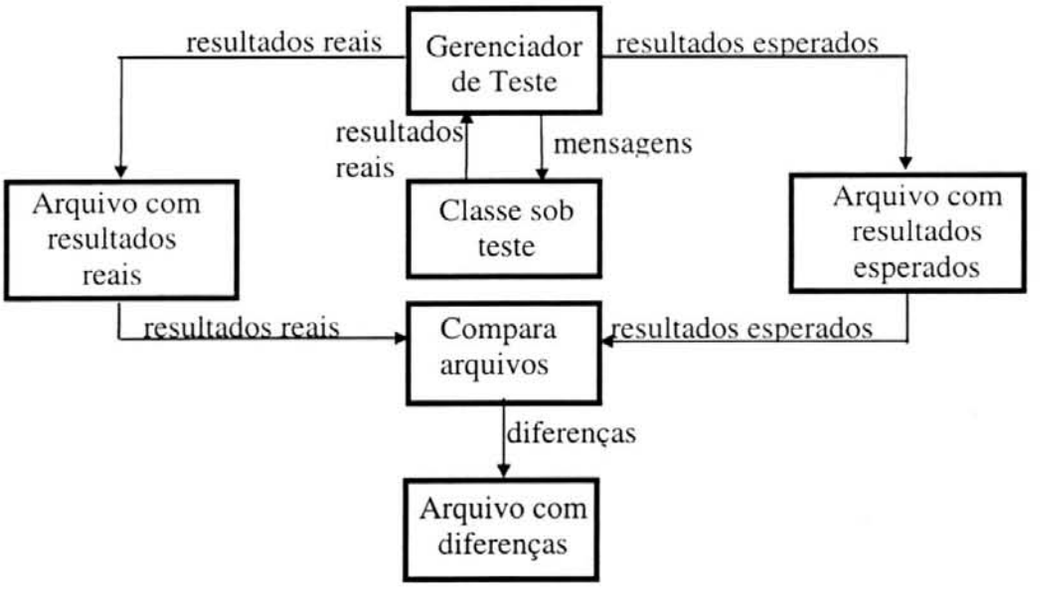

FIGURA 3.2 - Diagrama de Fluxo de Dados para Verificação de Resultados

3.3.2 Teste Baseado em Estados de Binder

O comportamento de uma classe é definido por um conjunto de valores encapsulados que a classe possui em determinado momento, sendo este controlado pelos valores encapsulados, por seqüências de mensagens, ou ambos. As classes respondem de forma diversa às mensagens recebidas dependendo de cada conjunto particular de valores encapsulados.

O teste de estados pretende testar o sistema OO sem tentar todas as combinações possíveis, fornecendo confiança de que o comportamento correto será produzido para todas combinações.

[BIN96] apresenta FREE (Flattened Regular Expressions), um framework para o desenvolvimento de casos de teste de unidade, integração e de sistema para softwares OO. Permite integração de casos de teste individuais produzidos por outros modelos de teste, fornecendo um framework coerente e consistente para teste

A proposta de FREE é uma técnica baseada em estados para desenvolvimento de casos de teste para sistemas OO. Não sendo uma metodologia específica, dirige-se a vários escopos de teste, tais como: teste de classes baseado na implementação e baseado na representação, teste de *clusters*, de subsistemas e de sistemas.

FREE necessita dos requerimentos e especificações do sistema para produzir os casos de teste.

A nível de unidade, uma classe ou *cluster* é nivelada e modelada como uma máquina de estados finitos (MEF). Este nivelamento fornece uma visão necessária para testes adequados dos métodos herdados. Seqüências de estados são geradas para cobrir todas transições de estado para a classe sob teste, gerando o modelo de estados.

Como um estado é o resultado de uma ativação de um método, pode-se substituir estados em máquinas de estados por grafos de métodos, a fim de obter o conjunto completo de caminhos por uma classe. FREE constrói o grafo de fluxo de controle, que permite testar os caminhos internos de uma classe, em três passos:

1. A máquina de estados é desenvolvida para cada classe sendo testada;
2. Um grafo de fluxo de controle é desenvolvido para cada método da classe; e
3. Cada arco de transição é substituído pelo grafo de fluxo de controle correspondente. O arco de transição entra no nodo de entrada do método e um arco é colocado do nodo de saída do método, conectando-o ao nodo do estado resultante da transição.

No teste de classes baseado na representação ou teste de *clusters* pequenos, as classes são divididas em modais[2] e não modais[3].

De posse de uma MEF, para as classes modais, o primeiro passo é preparar uma árvore de transição e, então, transcrever a seqüência de teste a partir desta, sendo que cada ramo gera um caso de teste. Os casos de teste devem ser acrescidos de casos de teste que verifiquem caminhos condicionais e caminhos que ocasionem a ativação de transições ilegais.

Em classes não modais é utilizada uma estratégia para verificar a seqüência de ativação, para obter a cobertura de caminhos através das definições e usos das variáveis.

No teste de integração para *clusters* grandes ou subsistemas, é utilizada uma máquina *mode*, a qual modela estados de sistema sintetizados por modelos de estado das

---

[2] Classes que restringem a seqüência de mensagens aceitáveis. Ex.: classes que representam entidades do domínio do problema.

[3] Classes que não fazem qualquer restrição à seqüência de mensagens que podem ser aceitas por elas. Ex.: classes que implementam operações em tipos de dados simples.

classes. As mesmas técnicas de derivação de expressões de caminhos utilizadas para a classe são utilizadas para o sistema. Seqüências de caminhos externos são identificados para verificar os limites do sistema sob teste. Os caminhos a serem testados são seqüências de mensagens executáveis, do estado inicial da máquina ao estado final.

Para o teste de sistemas é usada a definição do caso de uso estendido (*extended use-cases*), que produz a cobertura completa das especificações do usuário, maximizando a confiança operacional no sistema.

É apresentado a seguir, um exemplo desta abordagem [BIN95b] para o teste da classe Conta, cujo modelo é apresentado na fig.3.3, e que possui as seguintes especificações:

- Transações de suporte a uma conta: abertura, débito, crédito, fornecimento do saldo, bloqueio/liberação, desativação, liquidação e fechamento;
- Conta aberta aceita todas as transações, exceto liquidação;
- Se a conta está com saldo negativo, a conta está negativa;
- Conta negativa pode aceitar somente transações de verificação de saldo, crédito, e débito de taxas (não aceita débito do cliente);
- Conta bloqueada somente pode ser liberada ou ter seu saldo verificado;
- A conta que não tiver qualquer transação realizada por cinco anos, torna-se inativa. Somente uma conta inativa pode ser liquidada. Se a conta liquidada estiver com saldo igual a zero, ela é fechada;
- Uma conta deve ter seu saldo igual a zero para ser fechada. Conta fechada não aceita nenhuma transação.

Classe: **Conta**

Super classe: **Object**

Operaçoes:

saldo	-	verifica o saldo atual da conta,
crédito	-	adiciona um valor ao saldo,
débito	-	subtrai um valor do saldo,
abre		cria uma nova conta,
bloqueio	-	suspende todas as transações da conta,
libera	-	libera novamente as transações da conta,
liquida	-	zera saldo da conta,
fecha	-	termina todas as atividades da conta.

Atributos:

saldo	-	saldo atual da conta,
número	-	número da conta,
ultimaAtiv	-	data da última transação.

FIGURA 3.3 - Modelo para a Classe Conta

A classe Conta tem várias restrições a seqüências e valores relacionados em seu comportamento, mostrados no diagrama de transição de estados na fig.3.4. Cada transição recebe o nome do método correspondente, expressões entre colchetes

representam as pré e pós-condições para que as transições sejam executadas, e *s* é o saldo da conta.

Para derivar seqüências de teste de transições de estado, é necessário que exista uma máquina de estados finita (MEF) completa (todos os eventos tem estados e ações), mínima (sem estados redundantes ou desnecessários), na qual todos os estados devem ser alcançáveis, com uma indicação de seu estado inicial..

A partir daí, é gerada uma árvore de transição, cujos ramos representam todas as seqüências de transições possíveis. O estado inicial da máquina é transformado na raiz da árvore (*aberta* é o estado inicial da classe Conta). Para cada transição que sai do estado raiz, um arco é desenhado para um nodo que representa o estado resultante. Isto se repete para cada nodo de estado resultante, até que este estado já apareça em um nodo antecessor ou seja um estado final.

As seqüências de teste de transições da classe Conta foram modeladas na árvore de transição apresentada na fig.3.5.

O próximo passo do projeto de teste é a transcrição das seqüências de teste de transições a partir da árvore. Cada arco total ou parcial resulta em um caso de teste. O plano de teste é completado identificando-se os valores de parâmetros dos métodos, estados esperados e exceções de cada transição. O teste é executado inicializando o objeto sob teste com o estado inicial, aplicando a seqüência e verificando o estado resultante com o estado esperado.

Os seguintes tipos de erros podem ser encontrados com este tipo de teste: transições esquecidas, transições incorretas, ações de saída incorretas e estados incorretos.

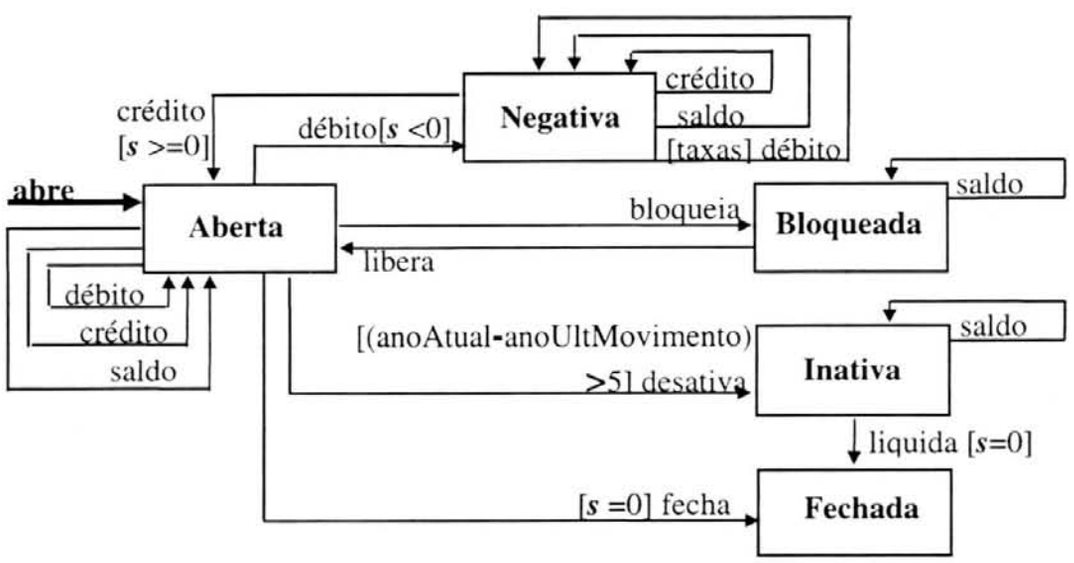

FIGURA 3.4 - Diagrama de Transição de Estados da Classe Conta

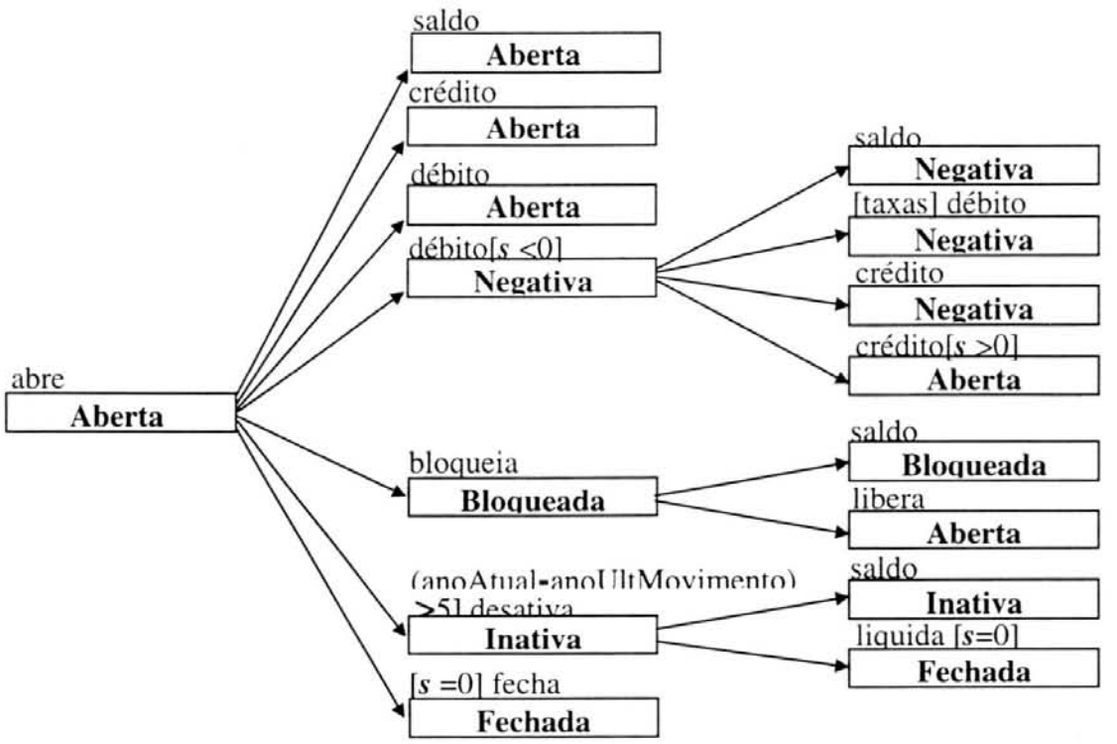

FIGURA 3.5 - Árvore de Transição da Classe Conta

3.3.3 Teste Baseado em Estados de Turner

A idéia de estado apresentada por Turner e Robson [TUR93] [TUR93a] é bastante similar à idéia apresentada por Binder. A ênfase do teste baseado em estados está nos valores armazenados na representação dos objetos, consistindo no teste das interações em uma classe pela monitoração das alterações dos valores dos atributos em um objeto, que conseqüentemente alteram seu estado.

O estado é definido como um conjunto particular de atributos. Diagramas de estados, ou MEF's representam os estados do objeto, que são modificados através de estímulos (ativação de métodos) [TUR93]. Cada estado é determinado pelo próximo método a ser chamado a partir do estado atual. Entretanto, há alguns métodos que não afetam o estado de um objeto, mas apenas consultam seu valor.

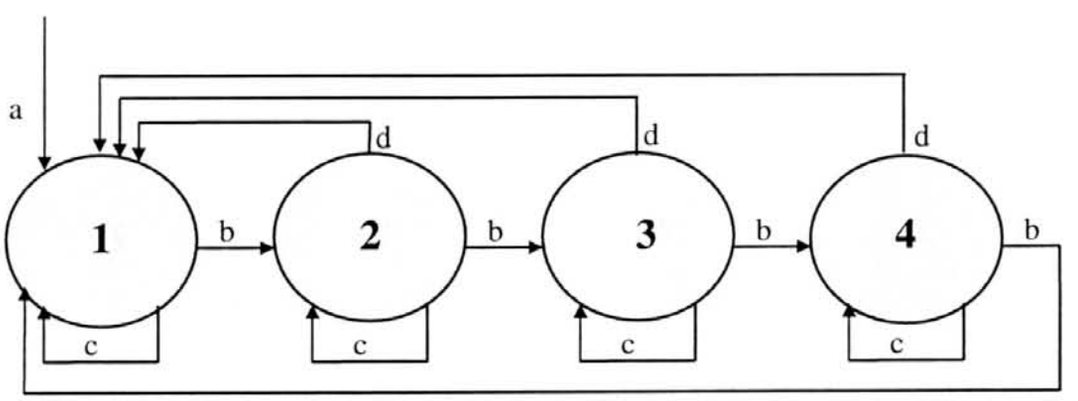

FIGURA 3.6 - Diagrama de Estado (MEF)

A fig.3.6 apresenta uma MEF que representa um programa o qual realiza uma contagem de 1 a 4, com quatro estados correspondentes. O teste para este diagrama seria simples, consistindo de envios de mensagens a todos os métodos, a partir de todos os estados (que são finitos, e em número consideravelmente reduzido). Neste diagrama estão indicadas quatro mensagens, que ativam os seguintes métodos:
(a) *Constructor ()* - Inicializa a MEF, inicializando o estado do objeto.
(b) *Next ()* - Altera o valor, causando a mudança de estado na seqüência.
(c) *GetValue ()* - Não afeta o estado atual do objeto, apenas informa seu valor.
(d) *Reset ()* - Inicializa o estado do objeto.

A ativação destes métodos pode causar quatro tipos de resposta para cada estado particular:
1. O método pode alterar o estado para um estado novo apropriado;
2. O estado pode continuar intacto.
   Esta resposta pode assumir dois resultados distintos, pois pode determinar um ***erro***, caso o estado devesse ser modificado;
3. O estado pode ser alterado para um estado indefinido no diagrama (***erro***);
4. O método pode mudar o estado para um estado inadequado (***erro***).

O exemplo acima necessita apenas de atributos simples para armazenar o estado do objeto. Quando este tipo de teste é aplicado a classes de sistemas de software OO, surgem vários outros fatores que influenciam o processo. Inicialmente, o número de atributos que a classe possui influencia a complexidade do teste, pois quanto mais atributos, mais casos de teste devem ser gerados. Também, existem diferentes tipos de classes, fornecendo uma maior interação entre os métodos e representações de dados, fazendo com que o processo de teste baseado em estados tenha que ser mais efetivo. Em terceiro lugar, quanto maior a complexidade do código dos métodos, mais difícil será testá-los. Para contornar estas dificuldades se faz necessária a inserção de asserções no final de cada subtarefa dos métodos para verificar o estado do objeto.

Para realização do teste primeiramente são determinados os valores dos subestados da classe. Um subestado representa o valor de um dado atributo em um determinado instante, que pode ser classificado como específico[4] ou geral[5]. O seguinte procedimento deve ser executado, para cada atributo:
1. Pela análise do código, ou do projeto, verificar quais valores são significantes;
2. Alocar um valor de subestado para cada valor significante;
3. Alocar um valor de subestado para cada grupo de valores relacionados.

O próximo passo no processo de teste é determinar o conjunto de estados válidos que podem ser aceitos como entrada e o conjunto de estados que podem ser produzidos, para todos os métodos da classe. Este procedimento deve ser feito a partir do projeto, já que podem haver erros no código.

A maior parte do processo do teste baseado em estados consiste na determinação do estado corrente do objeto. Para tal, é gerada uma nova versão da classe sob teste. Pelo menos um método por subestado é necessário, permitindo que o testador inspecione os valores dos subestados escolhidos. Entretanto, se há uma modificação de estado entre dois valores considerados parte do mesmo grupo geral, a modificação não é identificada. Este problema é contornado adicionando-se um conjunto extra de atributos,

---

[4] Testado diretamente no código, ou descrito no projeto como especial.

[5] Grupo de valores de subestados que são considerados da mesma maneira, não sendo necessário distingui-los um do outro para o teste.

com o propósito de refletir o valor dos atributos originais. Para cada atributo inspecionado, um método extra é necessário para testar as diferenças entre os valores dos atributos originais e os gerados na execução do teste.

São inseridos comandos no início e no final de cada método, com o objetivo de registrar em um arquivo os valores dos parâmetros passados e os valores retornados pelos métodos. Pode-se fazer isso também com os métodos de subestados, auxiliando a depuração dos erros que podem ocorrer, permitindo a execução do caso de teste a ser executado.

Além de detectar alterações no estado do objeto, o teste baseado em estado também identifica erros na construção de estruturas complexas e dinâmicas (tais como listas encadeadas), verificando as modificações que podem ocorrer na estrutura e o momento em que podem ocorrer. Esta análise produz uma lista de situações significantes no modelo utilizado, que são os cenários de dados.

Dois conjuntos diferentes de cenários são produzidos para os estados de um objeto, e projetados testes para cada um deles, considerando os subestados definidos:

- Cenários utilizados pelos métodos da classe que agem sobre os seus próprios elementos, alterando desta forma o valor do estado;
- Cenários utilizados pelos métodos que agem sobre as ligações entre os nós, sendo que estes alteram a estrutura do atributo, incluindo ou eliminando elementos.

O teste de integração de classes [TUR93a] enfatiza interações entre métodos de diferentes classes, envolvendo a criação de estados, não apenas para a classe que está sendo testada, mas também para os objetos que são passados como parâmetros.

Turner justifica a validade do teste baseado em estado para o teste de integração mostrando como exemplo duas classes, A e B, onde um objeto da classe A é passado como um parâmetro para a classe B. Os seguintes passos devem ser executados:

1. A classe A deve ser testada isoladamente;
2. A classe B deve ser testada como uma unidade usando *stubs* de teste;
3. O teste de integração da classes A com a classe B é realizado.

Um caso de teste para a classe A deve criar um estado inicial para o objeto desta classe, invocar o método a ser testado e validar o estado resultante do objeto sob teste.

Um caso de teste para o teste de integração das classes A e B deve criar um estado inicial para o objeto da classe B, criar um estado inicial para o objeto da classe A, invocar o método da classe B sob teste, validar o estado resultante para o objeto da classe A e validar o estado resultante para o objeto da classe B.

3.3.4 Teste Incremental Hierárquico de Classes de Harrold

A herança é um mecanismo para especificação de classes e compartilhamento de código, que suporta o desenvolvimento de novas classes baseadas na implementação de classes já existentes. Ela permite que uma subclasse herde os atributos[6] de suas superclasses, podendo estendê-las, restringi-las ou redefini-las.

A técnica de teste incremental de classes proposta em [HAR92] explora a natureza hierárquica da relação de herança para testar grupos de classes relacionadas, reutilizando as informações de teste das superclasses para guiar o teste das subclasses. É utilizada uma abordagem top-down, na qual as superclasses são testadas inicialmente,

---

[6] Nesta proposta, o termo "atributos" de uma classe é utilizado como consistindo de membros de dados ou variáveis de instância, que implementam o estado do objeto, e funções ou métodos, que implementam as operações no estado do objeto.

como unidades, sendo guardada a história de teste que será analisada no teste das subclasses.

Uma "história de teste" associa cada caso de teste com os métodos testados por este caso. Quando uma subclasse herda os atributos de uma superclasse, esta história de teste é também herdada. A história de teste herdada é incrementalmente atualizada para refletir diferenças em relação à superclasse. Uma história de teste da subclasse direciona a execução dos casos de teste, pois ela indica que casos de teste devem ser executados para testar a subclasse. Com esta técnica, automaticamente pode-se identificar novos atributos na subclasse que devem ser testados, juntamente com atributos herdados que devem ser retestados.

Os atributos herdados são retestados no contexto da subclasse devido à identificação e teste de suas interações com atributos novos da subclasse. Pode-se também identificar quais casos de teste da superclasse podem ser reutilizados para validar a subclasse, e que atributos da subclasse requerem novos casos de teste.

Para definição da subclasse é criado um *modificador* que define atributos diferentes, ou que foram alterados da superclasse. O modificador e a superclasse, junto com as regras da herança para a linguagem, são utilizados para definir a subclasse.

A fig.3.7 apresenta a herança como uma técnica de modificação incremental, que transforma uma superclasse **P**, através de uma classe modificadora **M** (que contém os novos atributos a serem acrescentados a R, em relação a P), em uma classe **R**. Utilizando o operador de composição ⊕, pode-se dizer que **R = P ⊕ M**.

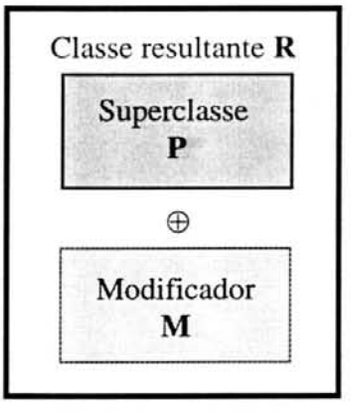

FIGURA 3.7 - Técnica de Modificação Incremental

A fig.3.8 mostra a decomposição da estrutura de herança em estruturas independentes. A hierarquia de herança mostrada à esquerda indica que a classe A possui duas subclasses, B e C, e a classe B possui uma subclasse C. A figura no centro ilustra o formato incremental da hierarquia de herança de classes, onde cada nova subclasse é formada pela combinação da superclasse com um modificador. À direita pode-se ver como a hierarquia de classes pode ser decomposta em estruturas distintas.

Na proposta de Harrold, a estrutura de herança é decomposta em um conjunto de pares ordenados de classe, dentro da hierarquia de herança, permitindo considerar somente uma classe e sua superclasse imediata na definição da nova classe. Por exemplo, a classe C pode ser determinada sem considerar a classe A, mas a relação de A para B deve ser resolvida antes da relação de B para C.

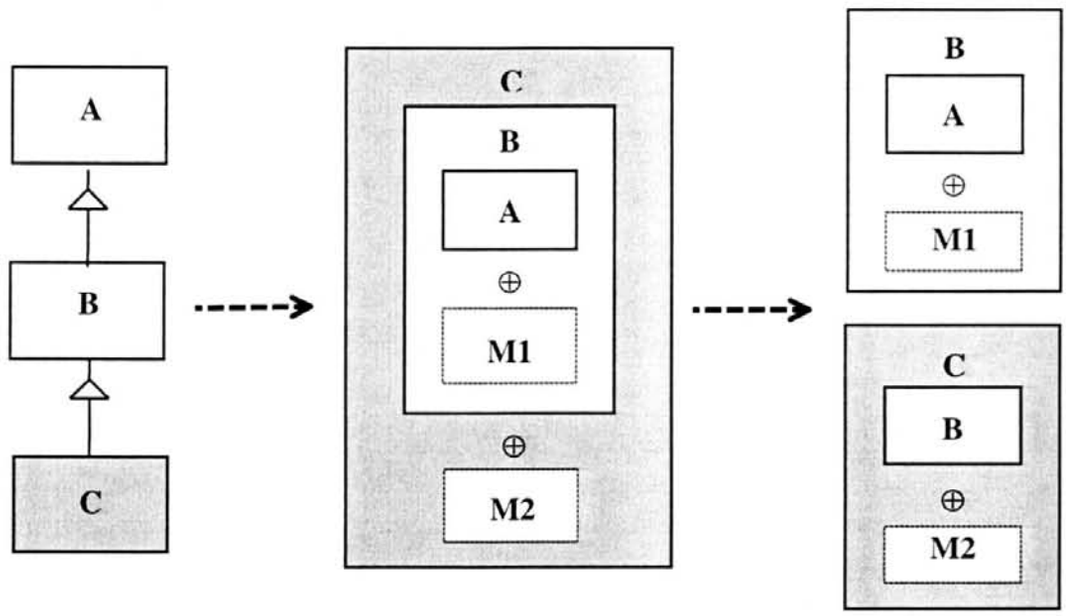

FIGURA 3.8 - Decomposição da Hierarquia de Herança

O projetista da classe controla a especificação do modificador, especificando a classe modificadora, que deve conter vários tipos de atributos que alteram ou são acrescentados aos atributos da superclasse, enquanto a herança controla a combinação do modificador e da superclasse para obter a subclasse.

A relação de herança também determina a visibilidade, a disponibilidade e o formato dos atributos de P em R. Uma linguagem pode suportar mais de um tipo de relação de herança, permitindo a especificação da visibilidade como, por exemplo, em C++ com os tipos: public, private e protected.

As superclasses são testadas isoladamente, utilizando-se técnicas de teste de unidade tradicionais, para testar métodos na classe. Depois, cada método é testado utilizando um conjunto de teste que contém casos de teste funcionais e estruturais. Estes testes requerem o uso de *drivers*, que simulam os métodos chamadores, e *stubs*, que representam os métodos chamados.

A história de teste da superclasse contém associações entre cada método na classe e os casos de teste (funcionais e estruturais), no formato de triplas {mi, (TSi, test?), (TPi, test?)}, onde mi é o método, TSi é o caso de teste baseado na especificação, TPi é o caso de teste baseado no código, e 'test?' indica se o caso de teste deve ser (re)executado.

É realizado então o teste das interações entre os métodos da classe. São realizados os testes intra-classe e também o teste de interações entre os métodos que acessam de outras classes (teste inter-classes).

As interações intra-classes são testadas através de um grafo de classes, no qual cada nodo representa um método na classe ou um membro de dados, e cada arco representa uma mensagem. Os métodos e membros de dados indicados pelo grafo são combinados e casos de teste são desenvolvidos para testar suas interfaces. A história de teste da classe contém nodos raiz dos subgrafos do grafo da classe representando as interações entre os métodos e seus casos de teste. Assim, a segunda parte da história de teste também consiste de triplas {mi, (TISi, test?), (TIPi, test?)}, onde mi é o método raiz do subgrafo do grafo da classe, TISi é o conjunto de casos de teste funcionais, TIPi é o conjunto de caso de teste estruturais, e 'test?'" indica se os casos de teste deve ser (re)executados. Neste momento 'test?' pode assumir três valores: Y, se deve ser

totalmente (re)executado, P, se deve ser parcialmente (re)executado e N se não necessita ser (re)executado.

O teste inter-classes é baseado nas interações de classes que ocorrem quando métodos de uma classe interagem com métodos de outra classe. O teste inter-classes é realizado de forma similar ao teste intra-classes, sendo que apenas os atributos estão em classes diferentes. As interações inter-classe ocorrem quando:

- Um método de uma classe é passado a uma instância de outra classe como um parâmetro e então envia uma mensagem a esta instância;
- Uma instância de uma classe é parte da representação de outra classe e então esta instância como uma mensagem.

O algoritmo para teste das subclasses usa uma técnica incremental que transforma a história de teste da superclasse P na história de teste da subclasse R, realizando o teste de acordo com o tipo do método.

Este algoritmo principia inicializando a história de teste da subclasse R com a história de teste da superclasse P, que já foi testada. A seguir cada atributo A no modificador M é analisado. Dependendo do tipo de atributo analisado, são realizadas as alterações apropriadas na história de R, adicionando-se novos casos de teste ou modificando-se os casos de teste herdados de P.

Em resumo, as informações relativas ao projeto e à execução dos casos de teste são armazenadas na "história de teste". Desta forma, quando uma subclasse é definida, a história de teste de sua superclasse, a definição do modificador e a relação de herança são utilizados para determinar quais atributos devem ser (re)testados e quais casos de teste da superclasse podem ser reutilizados. A técnica é hierárquica porque é guiada por uma ordem parcial da relação de herança; e é incremental porque utiliza os resultados do teste de um nível da hierarquia para reduzir os esforços necessários nos níveis subseqüentes.

3.4 Ferramentas para Teste de Software OO

Embora as pesquisas na área de teste de software estejam avançando consideravelmente, existindo várias estratégias e/ou técnicas sendo propostas na literatura existente, poucas ferramentas (ou protótipos de ferramentas) são encontradas. A seguir são apresentadas algumas ferramentas que visam automatizar a tarefa de teste de softwares OO.

- ASTOOT (A Set of Tools for Object-Oriented Test):
  ASTOOT [DOO94] é um conjunto de duas ferramentas para teste de classes definidas codificadas na linguagem Eiffel, cuja proposta é baseada em especificações algébricas de tipos abstratos de dados (especificação formal dos requisitos do sistema). ASTOOT automatiza a geração de *drivers* de teste, que automaticamente realizam a execução dos casos de teste e a verificação do resultado do teste. Para classes que possuem uma especificação algébrica, pode automatizar parcialmente a geração de casos de teste.
  A classe é considerada a unidade sob teste, utilizando especificação formal dos requisitos do sistema para testá-la. Os casos de teste consistem de seqüências de mensagens e de um indicador, que informa se esta seqüência de mensagens deverá resultar em uma mudança de estado do objeto.
  A execução do teste é realizada enviando-se estas mensagens para um objeto da classe sob teste, e a verificação consiste da execução de uma rotina que testa se ocorreram mudanças no estado do objeto e se este está no estado correto.

ASTOOT compõem-se de três módulos: o gerador de *drivers*, o compilador e o simplificador. O gerador de *drivers*, a partir de uma entrada que contém as especificações das interfaces da classe sob teste e de algumas classes relacionadas, gera o *driver* de teste. Este *driver* de teste, quando executado, lê os casos de teste fornecidos pelo usuário, verifica sua sintaxe, executa-os e confere os resultados. O compilador e o simplificador juntos formam uma ferramenta para geração semi-automática de casos de teste a partir de uma especificação algébrica. Esta ferramenta necessita que o usuário forneça uma especificação escrita em LOBAS e uma seqüência de operações possíveis.

- FOOT (Framework for Object-Oriented Test):
  FOOT [SMI92] é um framework que fornece suporte para o teste de programa OO. Este framework possui como componentes a classe sob teste, um programa de orientação principal, um arquivo de informações da classe sob teste, as estratégias de teste e um objeto executor de teste.
  O programa de orientação principal controla a execução do teste e fornece uma interface com o desenvolvedor. Este programa envia mensagens ao objeto executor de teste, que realiza a verificação, e apresenta os resultados ao desenvolvedor.
  Após a seleção da classe a ser testada, as informações a seu respeito são extraídas através de um analisador. Estas informações incluem o nome dos métodos, seus parâmetros e seus tipos, e são armazenadas juntamente com informações dos testes executados e dos resultados obtidos.
  As estratégias de teste do framework direcionam o teste da classe, com o objetivo de reduzir os testes realizados. Elas selecionam os métodos que são usados, em que ordem e com quais parâmetros, para testar a classe. Verificam os tipos de métodos disponíveis e fornecem rotinas de suporte para determinar o processo de teste, bem como o final de cada ciclo de teste.
  As informações da classe sob teste aliadas às estratégias de teste geram o objeto executor de teste que testará a classe. Este objeto é gerado como uma instância da classe sob teste com o objetivo de efetuar sua verificação. O objeto executor de teste tem acesso e recebe chamadas do programa de orientação principal.
  O programa de orientação principal cria instâncias do programa executor de teste que por sua vez cria instâncias da classe sob teste. O teste é realizado sob o controle do programa de orientação principal, sendo que o executor de teste realiza a verificação da classe utilizando as estratégias de teste, e os resultados são apresentados ao desenvolvedor.
  Um ponto que não é discutido nesta proposta, é uma das mais difíceis tarefas a serem executadas: a criação de objetos complexos para serem passados como parâmetros. Um outro ponto é a construção das estratégias de teste do framework, não havendo nenhuma descrição de como funcionam ou interagem com o mesmo.

- Ferramentas para o Teste Baseado em Estados:
  De acordo com a técnica de teste baseado em estados, já apresentada na seção 3.4.3, os casos de teste consistem de: código para gerar o estado inicial para o teste, verificação de que os estados requeridos foram produzidos, código para realização do teste, e código para validar o estado final do objeto.
  Com o objetivo de automatizar a geração de casos de teste, realizar a ligação entre estes casos e executá-los foi desenvolvido um conjunto de ferramentas [TUR93b], cujos protótipos ainda não estão completamente validados. Este conjunto é formado por três ferramentas, descritas a seguir:

1. MKTC (*Make Test Cases*):
   Esta ferramenta foi projetada para a geração dos casos de teste, onde a criação do estado inicial, e o teste dos estados inicial e final são automatizados. O testador fornece informações através de *scripts* de teste para que sejam desenvolvidos os casos de teste requeridos e vários relatórios. Um *script* deve possuir um formato determinado e conter informações sobre os subestados que serão usados na classe sob teste, os cenários de dados, uma lista de geradores para a criação dos estados iniciais, e casos de teste extras.
   Os geradores são descritos em termos de seus estados iniciais (pré-condições), e estados finais (pós-condições), e o código requerido para alterar o estado dos objetos para estas condições.
   MKTC fornece um relatório no qual são listados os detalhes de todos os casos de teste gerados, e uma lista de todos os geradores que não foram encontrados. Desta forma a ferramenta pode ser usada em dois passos: inicialmente verificando a sintaxe e a semântica do *script* de teste e, a seguir, listando todos geradores que não foram encontrados. Os geradores requeridos são adicionados ao script, e a ferramenta é executada novamente gerando os casos de teste. Cada caso de teste é armazenado em um arquivo, identificado por um número que varia de 1 a 999.

2. MKMFTC (*Make MakeFile for Test Cases*):
   Esta é uma ferramenta simples, que requer a existência de um arquivo, descrevendo os casos de teste que devem ser compilados, no diretório corrente. Em um primeiro estágio os casos de teste, gerados a partir do MKTC ou não, são compilados e é realizada a sua ligação através do utilitário *make*. *Make* usa o arquivo MakeFile como uma descrição de como executar este processo. Um MakeFile é um arquivo que contém uma descrição da dependência entre os módulos requeridos para formar casos de teste executáveis.
   MKMFTC possui, também, capacidade de gerar um MakeFile para casos de teste que não sejam baseados em estado. Os casos de teste baseados em estado e os outros casos de teste são tratados separadamente, gerando cada um seu próprio MakeFile. A geração do(s) arquivo(s) MakeFile é realizado em um segundo estágio da execução da ferramenta.

3. TESTRUN:
   Auxilia o testador na execução dos casos de teste, já compilados, do diretório corrente. Armazena em arquivos todas saídas geradas por cada caso de teste, e os casos de teste que falharam.
   Da mesma forma que MKMFTC, TESTRUN pode manipular casos de teste baseados em estado e casos de teste que não sejam baseados em estado.

4 Reflexão Computacional

O conceito de reflexão computacional apresentado por Maes em[MAE87] é o seguinte: "*Reflexão computacional é a atividade executada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações*".

Os aspectos principais da capacidade de reflexão são a *introspeção*, que é a capacidade que um programa possui de inspecionar entidades que definem sua computação, podendo raciocinar acerca de seu próprio estado, e a *intervenção*, que é a capacidade de alterar este estado.

A reflexão tem se tornado uma questão importante no projeto de linguagens e de arquiteturas, devido às suas características. As principais características dos sistemas reflexivos podem ser resumidos na capacidade que apresenta para:

- Extensões incrementais de sistemas e re-implementações dos mesmos;
- Monitoração de entidades de linguagens (inspeção, análise e modificações dinâmicas do código dentro da própria linguagem);
- Depuração de códigos (através da implementação de ferramentas de programação na própria linguagem) [FER89].

A computação reflexiva não contribui diretamente para resolver os problemas do domínio externo do sistema, mas contribui para a organização interna do sistema e/ou para o seu relacionamento com o mundo real.

A reflexão computacional define 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. As computações do objeto, localizado no nível base, tem por objetivo resolver problemas e retornar informações sobre o domínio externo (domínio da aplicação), enquanto o nível reflexivo, localizado no meta-nível, resolve os problemas e retorna informações sobre as computações do objeto, podendo adicionar funcionalidades extras a este objeto.

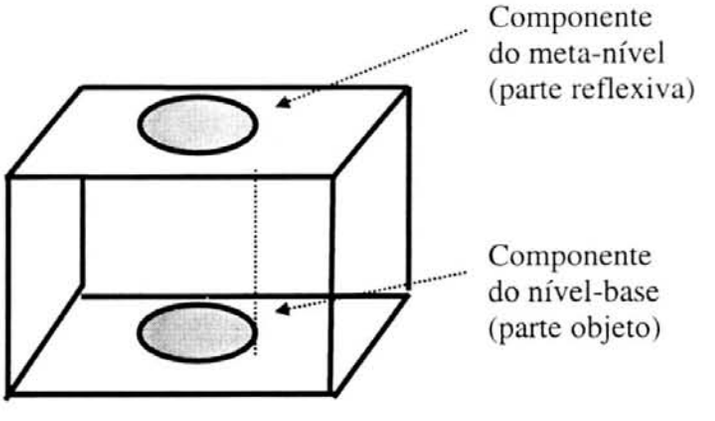

FIGURA 4.1 - Arquitetura Reflexiva

A fig.4.1 permite observar uma arquitetura reflexiva, composta por um meta-nível, que contém a reflexão do sistema objeto e um nível-base, onde ficam os objetos do domínio da aplicação. Os dados do nível base são usados no meta-nível para a realização de computações reflexivas, que podem interferir nas computações de nível-

base. O nível reflexivo pode atuar sobre a estrutura ou sobre o comportamento do sistema objeto, de acordo com os objetivos da reflexão computacional.

4.1 Reflexão Computacional no Paradigma OO

A reflexão computacional não é um conceito ligado unicamente às linguagens OO, embora este paradigma lhe tenha investido uma importância crescente devido à sua aplicação prática cada vez maior. Linguagens funcionais, como Lisp, e linguagens de programação em lógica, como Prolog, também possuem capacidades para implementar computações reflexivas.

Reflexão computacional é toda atividade que um sistema computacional realiza sobre si mesmo, de forma separada das computações em curso, com o objetivo de resolver seus próprios problemas e obter informações sobre suas computações, [LIS95], capacitando o sistema a analisar e modificar seu próprio comportamento ou estrutura.

A computação reflexiva pode atuar tanto sobre a estrutura do sistema objeto, ocorrendo a nível de classes, quanto sobre o comportamento, quando ocorre a nível de objetos. Quando a computação reflexiva ocorre a nível de classes, o meta-nível é composto por meta-classes, que contém informações sobre os aspectos estruturais do nível-base. Se a computação reflexiva ocorre a nível de objetos, o meta-nível é composto por meta-objetos, os quais contém informações sobre os aspectos do comportamento dos objetos (instâncias das classes) do nível-base.

A principal diferença entre a reflexão estrutural e comportamental é que a primeira realiza a associação de classes a classes de objetos (meta-classes), enquanto a segunda realiza a associação de objetos a objetos do meta-nível (meta-objetos).

4.1.1 Reflexão Estrutural - Modelo de Meta-Classes

Na reflexão estrutural, o meta-nível é composto por meta-classes, que descrevem a estrutura de suas classes. Desta forma, as classes são instâncias de outras classes, chamadas meta-classes (fig.4.2).

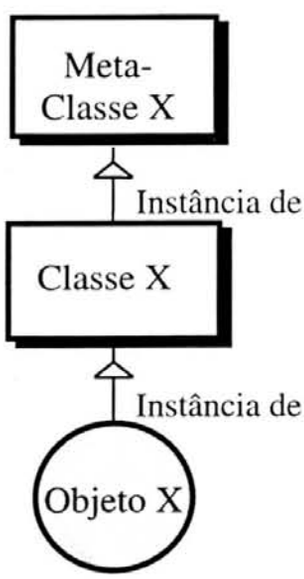

FIGURA 4.2 - Reflexão Estrutural

As meta-classes contém informações sobre os aspectos estruturais do nível-base, tais como o nome da classe, a descrição de variáveis e de métodos que irão compor todas as suas instâncias, e informações sobre herança. A estrutura da classe pode ser alterada, alterando assim todas suas instâncias.

Em resumo, reflexão estrutural de uma classe é toda atividade realizada em uma meta-classe com o objetivo de obter informações e realizar transformações sobre a estrutura estática da classe, tais como, métodos, variáveis de classe e herança.

Informações e transformações típicas de reflexão estrutural são [LIS95] :

- Obter informações sobre uma classe e sua classe ascendente, suas descendentes, suas instâncias, seus métodos e interfaces;
- Alterar uma classe, modificando suas variáveis e seus métodos;
- Atuar sobre classes, criando novas classes, eliminando classes existentes e renomeando outras.

Este modelo de reflexão foge ao escopo deste trabalho, onde foi utilizado o modelo de meta-objetos, desta forma é apresentada apenas uma síntese conceitual inserida em uma visão geral da reflexão computacional.

4.1.2 Reflexão Comportamental - Modelo de Meta-Objetos

Neste modelo, o qual foi utilizado pelo protótipo proposto através do framework *Luthier MOPs*, o meta-nível é composto por meta-objetos (objetos do meta-nível), associados a objetos do nível-base. Os meta-objetos contém informações sobre os aspectos de comportamento e estrutura dos objetos (instâncias de classes) do nível-base.

Objetos e meta-objetos são interconectados de tal forma que uma modificação em um deles, provoca efeitos no outro, de forma dinâmica (conexão causal).

A fig.4.3 mostra como se estruturam as classes na reflexão comportamental.

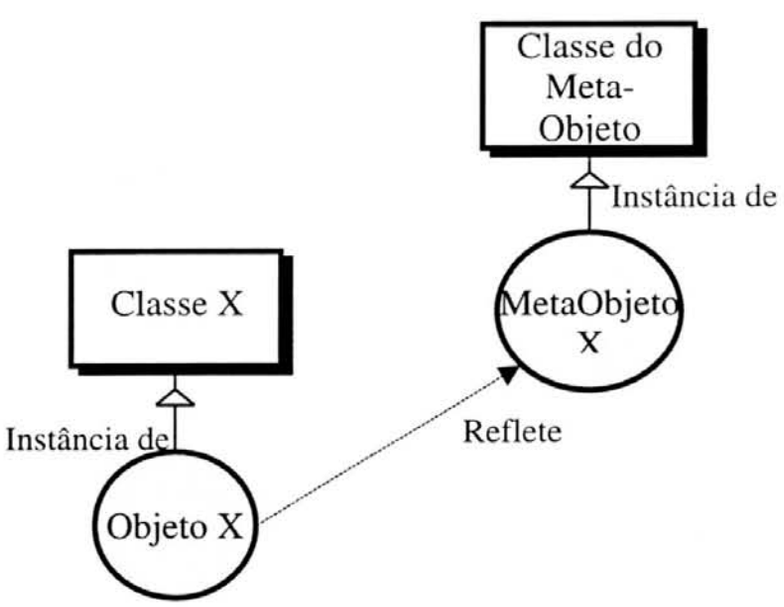

FIGURA 4.3 - Reflexão Comportamental

O conceito de meta-objetos foi introduzido como um mecanismo para suportar comportamento reflexivo em linguagens orientadas a objetos [MAE87]. Os meta-objetos são objetos que definem, implementam, dão suporte ou participam de alguma maneira da execução da aplicação. O comportamento computacional de cada objeto que faz parte de uma aplicação é determinado por um meta-objeto que controla e define a operação deste objeto. Novos comportamentos podem ser adicionados, ou a forma como os métodos são executados pode ser modificada.

A reflexão computacional permite o controle do comportamento do objeto ao receber uma mensagem, possibilitando a verificação de informações sobre o processo de execução, com o objetivo de monitorá-la, podendo modificar o curso desta, se necessário.

A conexão é realizada ligando-se um objeto a um meta-objeto. Sempre que o objeto receber uma mensagem, o meta-objeto a intercepta. Na interceptação de mensagens, todas as mensagens enviadas ao objeto (ou seja, a algum método do objeto reflexivo) são interceptadas pelo meta-objeto, que passa a realizar o tratamento da mensagem, dependendo de informações dinâmicas, obtidas durante a computação. O meta-objeto realiza a transformação de atributos do programa em dados disponíveis ao próprio programa (reificação[7]), e realiza as computações no meta-nível.

Os objetos reificados constituem as meta-informações sobre as quais são realizadas as computações reflexivas.

No contexto deste trabalho, são utilizadas informações estruturais para a verificação e análise do comportamento dos objetos da aplicação.

A fig.4.4 apresenta uma arquitetura reflexiva de meta-objetos:

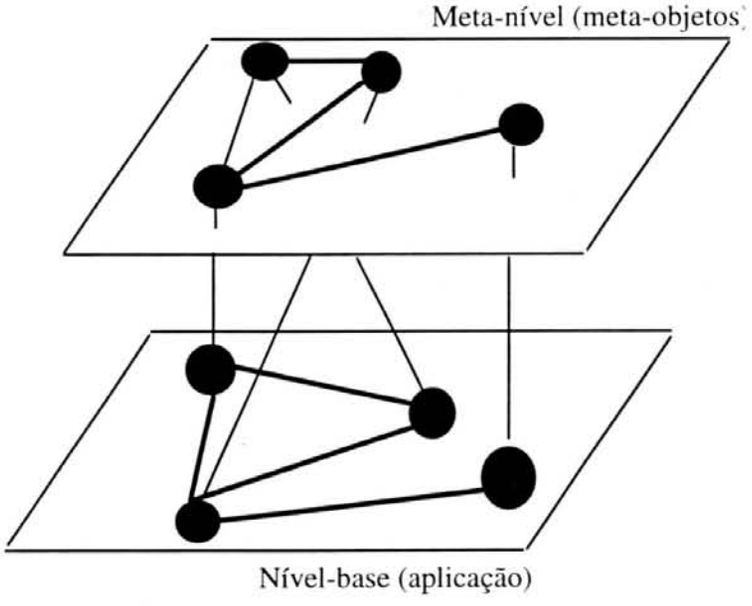

FIGURA 4.4 - Arquitetura Reflexiva de Meta-Objetos

Exemplos de aplicações que utilizam este modelo de reflexão computacional em suas implementações podem ser encontrados em [AMA97], que propõe um ambiente para programação de sistemas multiagentes, em [LIS95], que descreve um

---

[7] Traduzido do termo *reification* da literatura sobre computação reflexiva, o qual refere-se à operação de transformação da atividade computacional do nível base em dados para o nível superior, determinando quais computações podem ser realizadas.

framework de apoio ao desenvolvimento de aplicações tolerantes a falhas, e em [SIC97], onde é mostrada uma ferramenta de avaliação de qualidade de aplicações Smalltalk.

Também o framework *Luthier MOPs* [CAM97], sobre o qual ***ATeste*** foi implementado, utiliza este modelo de reflexão.

4.2 Framework *Luthier MOPs*

O framework *Luthier MOPs* (Meta-Object Protocol) [CAM96] [CAM97], desenvolvido em Smalltalk-80, foi utilizado como base para a construção de ***ATeste*** por suas características de apoio à construção de ferramentas de análise dinâmica de programas. Na seção 5.4 são descritas as classes reutilizadas no protótipo e a sua implementação.

Este framework foi projetado para suportar a construção deste tipo de ferramentas através de técnicas de reflexão computacional baseadas em meta-objetos. Neste modelo de reflexão, os meta-objetos definem, implementam, dão suporte e participam da execução da aplicação do nível base, permitindo implementar diferentes mecanismos de análise que podem ser dinamicamente associados com a aplicação analisada.

Luthier MOPs fornece uma infra-estrutura altamente flexível e adaptável para a associação de meta-objetos com classes. Um meta-objeto pode ser associado a um método, a um conjunto de métodos (ainda que de classes diferentes); a uma instância ou a um conjunto de instâncias de diferentes classes; a uma classe ou a um conjunto de classes. Um método, instância ou classe pode ter vários meta-objetos associados [CAM97].

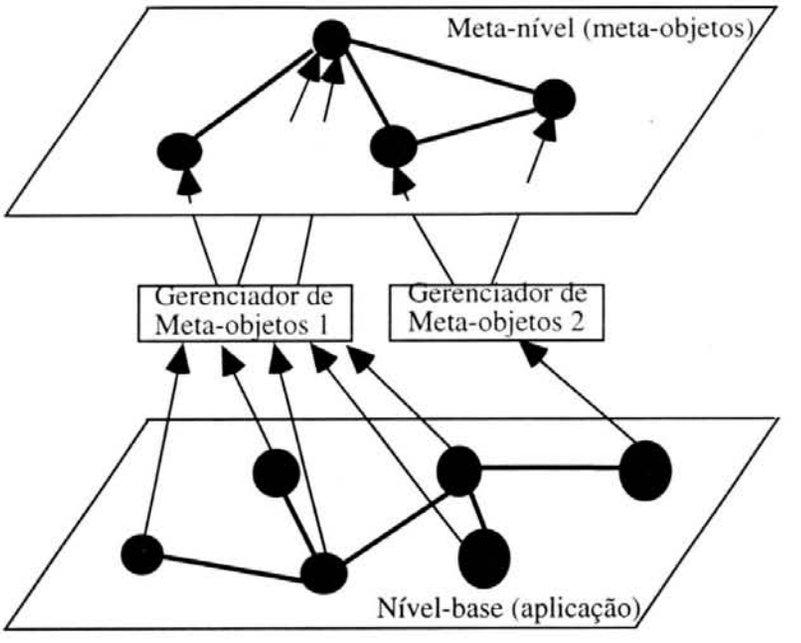

FIGURA 4.5 - Relacionamento entre Objetos, Meta-Objetos e Gerenciadores de Meta-Objetos

Entre as principais contribuições de *Luthier MOPs* pode-se mencionar o conceito de gerenciadores de meta-objetos (fig.4.5), que provê facilidades para a associação de meta-objetos com classes, instâncias ou métodos específicos, e suporte para múltiplos meta-objetos com prioridade de ativação.

Um gerenciador de meta-objetos age como um mediador entre o nível base e os meta-objetos. Quando uma mensagem do nível base é refletida, ela é dirigida para um dado gerenciador que encapsula o mecanismo que seleciona o meta-objeto que deverá tratar a mensagem. O gerenciador de meta-objetos contém a informação necessária para determinar qual meta-objeto, se houver algum, deve ser ativado quando uma mensagem é refletida.

A utilização deste framework pela ferramenta proposta acontece através da implementação de um conjunto de meta-objetos específicos, que são, de forma dinâmica, ligados às classes selecionadas pelo testador, para análise e monitoração, em tempo de execução. Estes meta-objetos monitoram a execução das aplicações, verificando os resultados da execução das mensagens, para posterior visualização e análise dos mesmos.

4.2.1 Estrutura e Funcionalidades

O diagrama de objetos do framework *Luthier MOPs*, é apresentado na figura 4.6. Suas principais classes são:

- *MetaObjectManager*: Implementa o mecanismo genérico de gerenciadores de meta-objetos, e deve ser especializada para implementar mecanismos específicos de associação e ativação de meta-objetos;
- *MetaObjectClass*: Classe abstrata que define o comportamento genérico dos meta-objetos, os quais tem como funcionalidade básica tratar as mensagens recebidas pelo(s) objeto(s) que reflete(m) seu comportamento neles;
- ReflectedMessage: Uma instância de *ReflectedMessage* encapsula toda a informação relativa à mensagem refletida tal como seletor, argumento, receptor e originador;
- MessageInterceptor: Define a interface geral para interceptar mensagens, que é o aspecto central da implementação do suporte de meta-objetos.

Quando um objeto refletido recebe uma mensagem, esta é desviada pelo mecanismo de interceptação para o gerente associado. Este gerente verifica o(s) meta-objeto(s) associado(s) com o objeto que refletiu a mensagem através da mensagem *findMetaObjectFor:* e decide quando transferir o controle para este(s) meta-objeto(s), o que é realizado através da mensagem *activateMetaObjects:forMessage:*, enviando-lhe(s) a mensagem *handleMessage:*.

Na ativação, uma instância da classe *ReflectedMessage* é enviada ao(s) meta-objeto(s), como parâmetro da mensagem *handleMessage:*, fornecendo-lhe(s) as informações relativa à mensagem refletida. Este(s) meta-objeto(s) pode(m) realizar computações próprias, e então executar o método original enviando ao objeto refletido a mensagem *send*.

As características das principais classes e métodos envolvidos no processo descrito acima são apresentadas a seguir:

- Implementação de MOPs - classe *MetaObjectManager*
  A classe abstrata *MetaObjectManager* implementa o mecanismo genérico dos gerenciadores de meta-objetos. Subclasses específicas devem fornecer a implementação dos métodos abstratos definidos para:

- Seleção de meta-objetos associados com um objeto que refletiu uma mensagem:
  O método abstrato ***findMetaObjectFor:*** busca o(s) meta-objeto(s) associados com o receptor da mensagem e deve ser implementado por subclasses de *MetaObjectManager*.
- Transferência do controle para os meta-objetos associados:
  Para implementar um método alternativo para a transferência do controle para os meta-objetos associados com os objetos do nível base deve ser especializado o método ***activateMetaObjects:forMessage***:.

A associação de objetos com meta-objetos: é realizada pelo método ***registerReflectionOf:on***:, que realiza a conversão dos métodos e objetos em reflexivos e cria a associação entre objetos do nível base e os meta-objetos.
O método ***reflectMessage:args:forObject***:, chamado pelo mecanismo de interceptação de mensagens, implementa o algoritmo que proporciona a busca e ativação dos meta-objetos associados com o objeto que refletiu uma mensagem.

- Definição de meta-objetos - classe *MetaObjectClass*
  A classe abstrata *MetaObjectClass* é responsável pela definição do comportamento genérico dos meta-objetos. Um meta-objeto tem como funcionalidade básica o tratamento das mensagens recebidas pelo(s) objeto(s) refletidos.
  Para o tratamento das mensagens refletidas por seu(s) objeto(s), subclasses específicas de *MetaObjectClass* devem prover a implementação do método abstrato ***handleMessage:***. Este método é o responsável pela definição das ações a serem executadas quando da reflexão de uma mensagem.
  Quando uma mensagem é refletida, este método é chamado por um gerenciador, que envia como argumentos uma instância da classe ReflectedMessage, a qual contém as informações acerca da mensagem refletida.
  Após a execução da ação desejada, um meta-objeto pode enviar a mensagem ao objeto receptor original, a qualquer outro objeto que deva ser ativado, ou até mesmo interromper a execução da aplicação.

- Reificação de mensagens - classe *ReflectedMessage*
  Na reificação de mensagens as informações relativas a uma mensagem (seletor, argumentos, receptor e originador) são transformadas em um objeto que pode ser tratado como um objeto normal no meta-nível.
  A classe *ReflectedMessage* implementa esta funcionalidade, criando uma instância desta classe quando uma mensagem é refletida. Cada instância encapsula toda a informação relativa à mensagem refletida.
  O método ***send,*** o qual não precisa ser redefinido, faz com que um meta-objeto possa executar o método original enviando a mensagem *send* para a instância de *ReflectedMessage* que está refletindo.

- Interceptação de Mensagens - classe *MessageInterceptor*
  A classe abstrata *MessageInterceptor* define a interface geral para interceptar mensagens, sendo especializada em duas subclasses: *ReflectedMethod* e *WrapperInterceptor*. O método para criação de interceptores não é visível para o usuário do framework, sendo utilizado pela interface de reflexão provida pela classe *MetaObjectManager*.
  A subclasse *ReflectedMethod* realiza a interceptação de mensagens baseada em métodos. Este tipo de interceptação foi utilizada em ***ATeste***. Esta subclasse

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

implementa o comportamento para encapsular métodos e transferir o controle para um dado gerenciador.
A reflexão de um método é implementada substituindo no dicionário de métodos da classe, a instância de *CompiledMethod*, que contém o código do método, por uma instância de *ReflectedMethod,* que mantém a referência ao método original.
A subclasse *WrapperInterceptor* implementa a interceptação baseada nos objetos, fornecendo o suporte para a reflexão de mensagens através da redefinição do método ***doesNotUnderstand***. Este método está definido na biblioteca de métodos do Smalltalk.

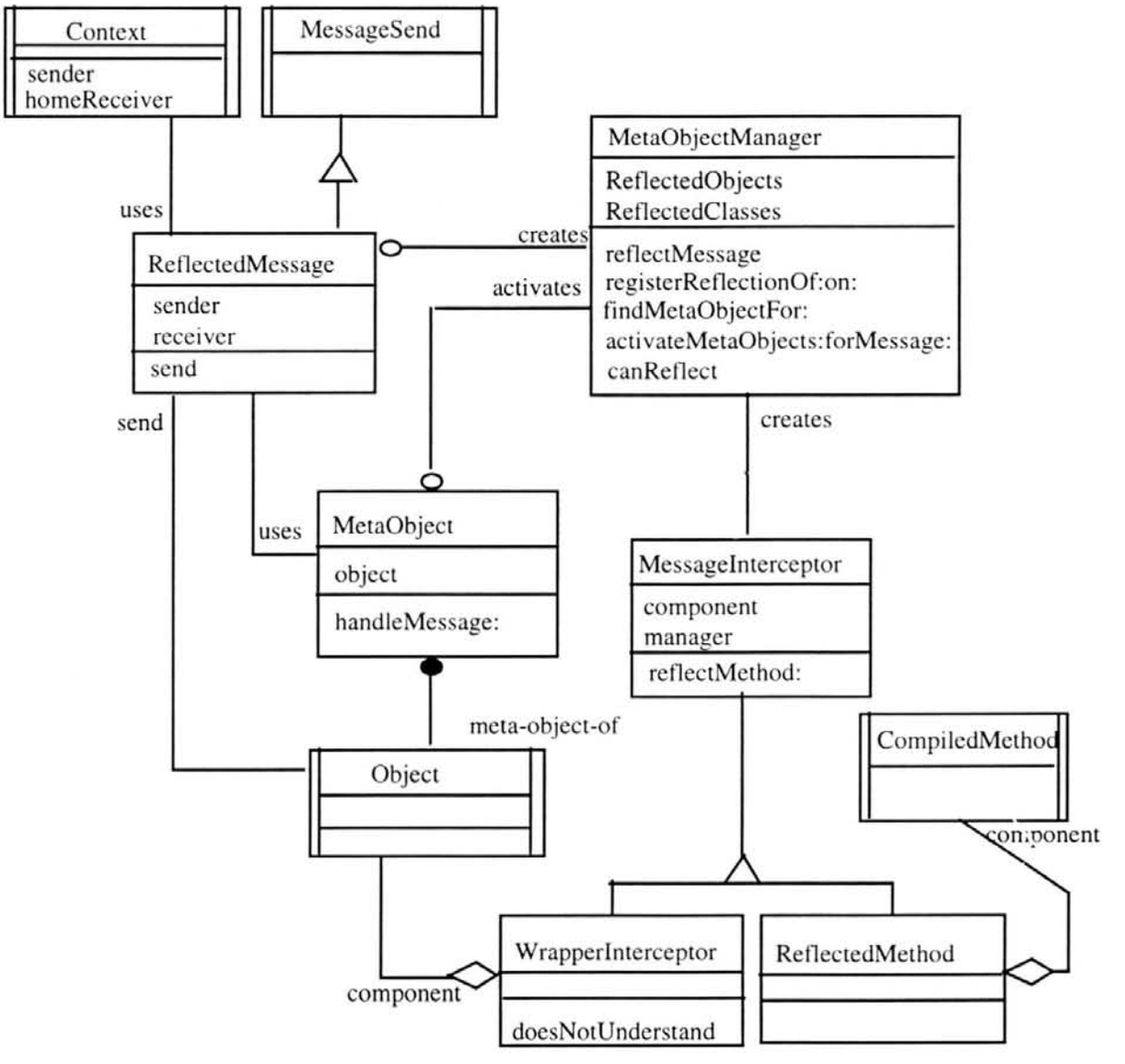

FIGURA 4.6 - Diagrama de Objetos do Framework *Luthier MOPs*

A interação deste framework com ***ATeste*** é apresentada no próximo capítulo.

5 *ATeste - Uma Ferramenta de Apoio ao Teste*

ATeste é um protótipo de ferramenta para auxílio ao teste de programas orientados a objetos, desenvolvido para uso em aplicações Smalltalk, com a finalidade de prover facilidades ao desenvolvedor de software nas atividades relacionadas à análise de códigos OO.

ATeste implementa uma estratégia de teste, denominada *teste dinâmico de caminho*, a qual aplica conceitos de teste baseado em estados [BIN96] [BIN95b] [TUR93] [TUR93a], aliados às vantagens da reflexão computacional. A reflexão computacional possibilita a análise dinâmica da aplicação através de mecanismos que podem ser associados à aplicação em tempo de execução, sem necessidade de alterar o código fonte da mesma.

Para o desenvolvimento de ***ATeste*** foi utilizado o ambiente VisualWorks, um ambiente totalmente orientado a objetos para construção de aplicações na linguagem de programação Smalltalk-80. A escolha deste ambiente para desenvolvimento se deve ao fato de que as ferramentas disponíveis facilitam a aplicação do estilo incremental de desenvolvimento de software, oferecendo as seguintes características:

- É adequado à prototipação;
- Provê facilidades para a construção de interfaces para comunicação com o usuário;
- Provê facilidades para a construção de ferramentas de Engenharia de Software devido à biblioteca de classes existente;
- Por hospedar uma linguagem interpretada, fornece as condições necessárias para a implementação de mecanismos reflexivos de forma eficiente.

Foi considerado também, o fato de ser este o ambiente de desenvolvimento do framework *Luthier MOPs* [CAM97], que fornece a ***ATeste*** o suporte para a implementação da reflexão computacional. A estrutura e funcionamento do framework *Luthier MOPs* foram apresentadas anteriormente, na seção 4.2, e sua utilização é discutida com maiores detalhes no decorrer deste capítulo.

Apesar de introduzir novos conceitos e, conseqüentemente, novos problemas a serem resolvidos, a programação Orientada a Objetos apresenta boas perspectivas para o teste de software. Cada objeto, por ser totalmente encapsulado, é essencialmente um programa completo com um escopo definido de operações e entradas. A especificação das classes descreve o resultado desejado de cada operação. Portanto, o processo de verificar um objeto pode ser simplificado como a comparação dos resultados de uma operação com o resultado especificado.

5.1 Teste Baseado em Estados

Objetos tem estados e comportamentos dependentes de estados. Isto é, o efeito de uma operação sobre um objeto depende do estado do objeto e pode alterar seu estado [KUN95]. O teste de aplicações orientadas a objetos deve verificar o efeito combinado destas operações.

Optou-se pelo teste baseado em estados por ser uma técnica que complementa as propostas de teste estrutural e funcional mais tradicionais. Os métodos de uma classe fornecem o comportamento desejado pela interação com a representação de dados. Embora estas interações sejam encontradas em programas escritos em linguagens tradicionais, são mais visíveis e predominantes em linguagens OO. O teste

baseado em estados testa as interações monitorando os efeitos que os métodos tem no estado de um objeto.

A técnica baseada em estados considera a classe como a menor unidade de teste, ao invés do método. O alto nível de acoplamento entre as representações de dados e os métodos implica que, mesmo que os casos de teste individuais se concentrem em um método simples, as representações da classe são ainda requeridas para o teste. Isto ocorre porque o estado determina a resposta do método na maioria dos casos [BIN96]. A idéia de estado apresentada por Turner [TUR93] [TUR93a] é bastante similar à idéia apresentada por Binder [BIN96] [BIN95b], consistindo no teste das interações em uma classe pela monitoração das alterações dos valores dos membros de dados.

O comportamento de uma classe é definido por um determinado conjunto de valores encapsulados que a classe possui em determinado momento, e controlado por valores encapsulados, seqüências de mensagens, ou ambos.

Cada método é considerado como um mapeamento entre o estado de entrada do objeto e o estado de saída, afetado somente pelo estado atual do objeto e pelos parâmetros passados. O estado atual de um objeto é determinado pelos valores de seus atributos. Um subestado é definido como o valor de um determinado atributo. Um estado parcial é a combinação de valores de um subconjunto de todos os atributos do objeto, em determinado instante. Assim, o estado do objeto é a combinação de todos os seus subestados, em um determinado instante.

5.2 Características Reflexivas

A implementação de mecanismos de análise dinâmica de aplicações, sem alteração do código fonte, tornou-se possível através da utilização de técnicas de reflexão computacional, que possibilitam a intervenção na computação da aplicação e o acréscimo de novas funcionalidades, sem a necessidade de imersão no código.

Smalltalk apresenta um ambiente rico em facilidades reflexivas. Exceto por algumas primitivas que incorporam sua máquina virtual, o interpretador e o ambiente são construídos a partir de classes do próprio ambiente Smalltalk, fazendo com que a reflexão computacional seja uma conseqüência natural do projeto da linguagem. Para facilitar a reflexão sobre um método compilado, Smalltalk contém as classes *CompiledMethod* e *MethodDictionary*, que são responsáveis pela representação dos métodos compilados e pela forma de ativação dos métodos em objetos através de mensagens. A manipulação destas classes torna possível a implementação de reflexão computacional sobre o programa do sistema ou do interpretador [LIS93].

Em Smalltalk, o dicionário de dados de uma classe está incluso em um vetor que contém, também, todos os dicionários de métodos de suas superclasses, objetivando otimizar a busca de métodos. O procedimento de busca e execução dos métodos é codificado diretamente no interpretador da linguagem, dificultando a introdução de novos comportamentos na linguagem. Mas, através da manipulação do vetor de dicionários, fazendo com que possua dicionários de métodos diferentes dos que supostamente deveria possuir, e dos métodos compilados localizados em cada dicionário, é possível a interceptação de mensagens [BAL96] [LIS93].

Esta técnica de interceptação foi implementada pelo framework *Luthier MOPs* [CAM97], que incorpora características reflexivas ao ambiente de desenvolvimento VisualWorks/Smalltalk, sendo utilizado pois como base para a construção de ***ATeste***.

5.3 Características de *ATeste*

ATeste tem como objetivo auxiliar o teste de aplicações Smalltalk, através da monitoração dinâmica de métodos executados para análise do estado dos objetos. ***ATeste*** intercepta o envio de mensagens quando o objeto receptor pertence a uma classe cuja reflexão foi solicitada. Suas principais funcionalidades são:

- Captura e apresentação do estado dos objetos receptores das mensagens interceptadas (valores de suas variáveis de instância) "antes e após a execução dos métodos selecionados para teste", juntamente com outras informações tais como nome do método que implementa a mensagem interceptada, objeto que enviou a mensagem e objeto que recebeu a mensagem;
- Apresentação de uma lista de todas as mensagens enviadas até um determinado momento aos objetos monitorados, contendo adicionalmente, o estado destes objetos no momento da execução, o nome do método que implementa a mensagem interceptada, objeto que enviou a mensagem e o objeto que recebeu a mensagem. Esta lista fornece, desta forma, o caminho completo da aplicação sob teste, que envolve as classes selecionadas;
- Permissão ao testador de informar os possíveis estados do objetos antes e após a execução dos métodos selecionados, para que a verificação dos resultados seja feita automaticamente.

Uma propriedade importante de ***ATeste*** é que as funcionalidades descritas acima foram implementadas sem que houvesse necessidade de instrumentação do código fonte da aplicação testada, tornando a ferramenta altamente flexível e de fácil utilização para o teste de qualquer aplicação Smalltalk. Não sendo necessária a imersão no código o usuário pode executar a aplicação sob teste quantas vezes forem necessárias, podendo ativar e desativar a reflexão de classes de forma transparente, sem que isto tenha qualquer repercussão na aplicação, simplesmente interagindo com a janela principal da ferramenta para informar que classes/métodos devem ser monitorados, alterando estas informações, e interrompendo a reflexão a qualquer momento.

Outro mérito da ferramenta, que deve ser salientado é que suas funcionalidades podem ser facilmente alteradas e novas estratégias de teste adicionadas, possibilitando inclusive incorporar métodos de geração automática de casos de teste, especializando classes componentes de ***ATeste***.

ATeste trabalha com um meta-nível, que monitora os objetos solicitados pelo usuário da aplicação do nível-base. As classes informadas são refletidas em meta-objetos no meta-nivel, sendo que todas as mensagens enviadas a estas classes são interceptadas. Quando uma mensagem é interceptada durante o processo de execução, o gerenciador de meta-objetos faz com que a ocorrência deste evento cause uma interrupção no processo de execução e o controle seja transferido para um meta-objeto do meta-nível, que realiza as computações especificadas e retorna o processo de execução. Nestas computações é realizada a verificação do método ativado e, se este foi selecionado para teste, são capturadas as informações necessárias para análise do estado do objeto, antes e após sua execução.

Inicialmente, o testador deve preparar um plano de teste de forma a identificar valores de parâmetros de métodos e estados esperados. Seguindo os conceitos de teste baseado em estados, devem ser especificados os possíveis estados que o objeto pode ter antes e após o envio da mensagem, o que pode ser feito através da análise de cada método e geração de pré e pós-condições.

A estratégia aqui proposta é realizada de forma iterativa, realizando a monitoração da execução da aplicação para as classes e métodos selecionados pelo usuário, como é descrito a seguir.

Em uma primeira etapa, o usuário de ***ATeste*** interage com a ferramenta, devendo informar a(s) categoria(s)[8] que deseja testar. A partir desta informação o sistema recupera a lista de classes que compõem cada categoria para que o usuário selecione somente aquelas que deseja testar. Os nomes dos métodos encapsulados nestas classes são mostrados para que o testador tenha ainda a possibilidade de selecionar apenas aqueles métodos mais críticos que deseja refletir. A reflexão das classes selecionadas é ativada e a informação sobre os métodos que devem ser monitorados armazenada.

Na segunda etapa, o teste é realizado executando-se a aplicação em análise através da ativação de um determinado método e inicializando-se os objetos com um estado inicial, sendo que próximos estados são determinados pelo próximo método a ser chamado a partir do estado atual.

A ativação deste primeiro método faz com que uma seqüência de mensagens seja ativada. Quando uma classe cuja reflexão foi solicitada, recebe uma mensagem, o estado do objeto é verificado, antes e após a execução do método ativado, para que o testador compare o estado atual (gerado durante a execução da aplicação) com o estado esperado (estados definidos na especificação).

A partir destas informações são apresentados, na seqüência de execução, o estado dos objetos antes e após a execução dos métodos selecionados para teste. Adicionalmente, é fornecida uma visão de todo o caminho de execução da aplicação, com o estado dos objetos antes e após a execução de todos os métodos ativados pelas mensagens recebidas por objetos que pertencem às classes refletidas.

5.4 Estrutura Geral

A arquitetura do protótipo compõem-se três grupos de classes: classes próprias, que implementam as funcionalidades do protótipo; classes especializadas a partir do framework *Luthier MOPs*, necessárias para implementar a reflexão computacional desejada, e classes do próprio ambiente VisualWorks/Smalltalk. Na fig.5.2 são mostrados os três grupos de classes através de um diagrama de objetos que os destaca.

A notação utilizada é baseada em [RUM97], acrescida de ícones próprios que objetivam distinguir melhor as classes apresentadas (fig.5.1).

NomeClasse	Retângulos com bordas simples indicam classes abstratas
NomeClasse	Retângulos com bordas largas indicam classes concretas
NomeClasse	Nomes de classes em itálico indicam classes pré-definidas no framework *Luthier MOPs*
NomeClasse	Nomes de classes em itálico e negrito indicam classes pré-definidas do Smalltalk

FIGURA 5.1 - Notações de Classes utilizadas em ***ATeste***

[8] As categorias identificam, normalmente, conjuntos de classes que formam uma aplicação.

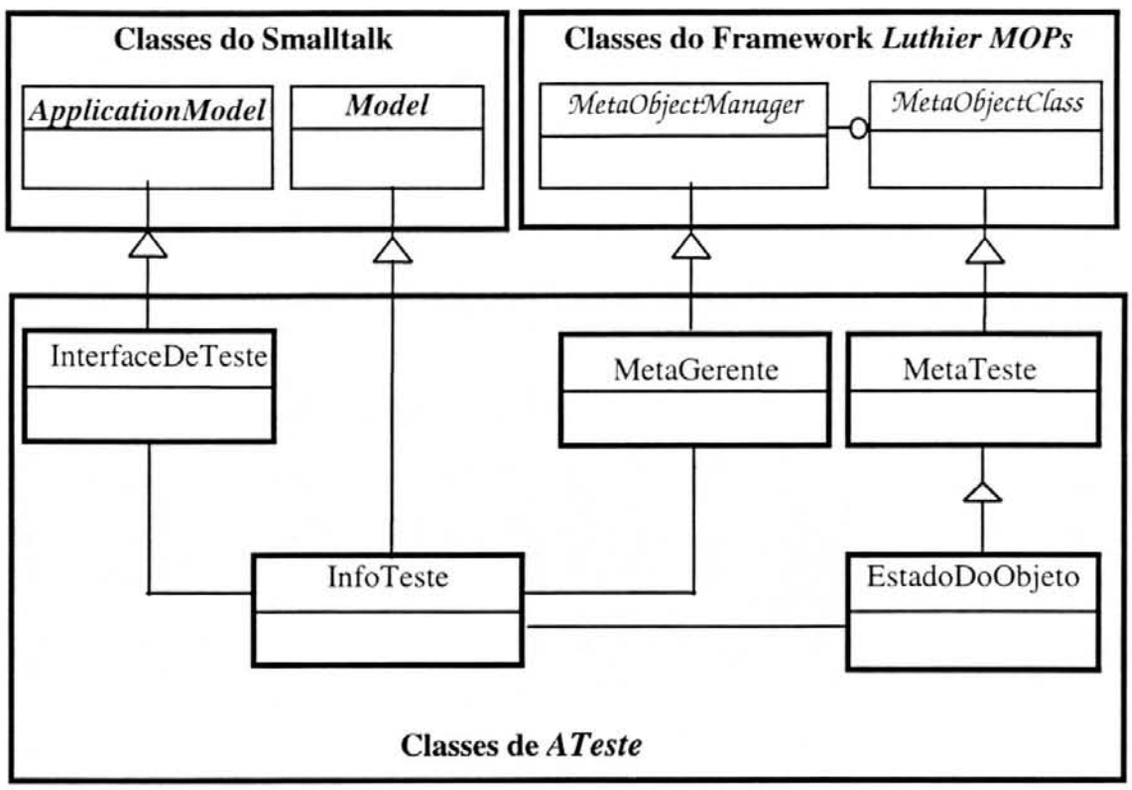

FIGURA 5.2 - Diagrama de Objetos de ***ATeste***

As próximas subseções fornecem uma descrição textual das classes que compõem cada grupo, apresentando um resumo de suas funcionalidades. A seção 5.5 fornece maiores detalhes sobre as classes modeladas no projeto de ***ATeste***, suas estruturas de dados e os comportamentos definidos.

5.4.1 Classes do Smalltalk/VisualWorks

Na programação com Smalltalk, toda a biblioteca de classes da linguagem fica disponível ao usuário.

No protótipo apresentado, o comportamento das seguintes classes é herdado da biblioteca de classes do Smalltalk:

- Model:
  Model é uma classe abstrata, subclasse de Object (classe raiz da biblioteca de classes de Smalltalk), que tem como principal funcionalidade definir e manipular estruturas de dados. Através do mecanismo de herança, Model fornece às suas subclasses as variáveis e métodos que suportam o mecanismo de dependência de informações, que tem por objetivo coordenar as atividades de diferentes objetos em uma aplicação. Desta forma, a classe InfoTeste, subclasse de Model, fica capacitada a manter uma lista de objetos que dependem dela para acessar informações, e a informar a estes objetos alterações ocorridas.
- ApplicationModel:
  Esta classe que, na hierarquia de classes do Smalltalk, é uma subclasse de Model, é uma classe abstrata que provê auxílio para definir desenhos de tela para criar uma interface com o usuário, e coordenar a comunicação entre os campos da interface e o modelo de dados. Os métodos e variáveis necessários para implementar estas funcionalidades são herdados pela subclasse InterfaceDeTeste.

5.4.2 Classes do Framework *Luthier MOPs*

Para utilização da estrutura do framework *Luthier MOPs*, as seguintes classes foram especializadas com o objetivo de atender as necessidades específicas de ***ATeste***:

- MetaObjectClass:
  MetaObjectClass é responsável pela definição do comportamento genérico dos meta-objetos, implementado pela subclasse MetaTeste. A classe MetaTeste provê a implementação do método abstrato *handleMessage:* para tratar as mensagens refletidas, recebendo as informações sobre estas mensagens (método ativado pela mensagem, classe do originador da mensagem, classe do receptor e a classe na qual a implementação da mensagem foi encontrada).
- MetaObjectManager:
  Esta classe implementa os mecanismos específicos de associação e ativação de meta objetos, sendo especializada pela classe MetaGerente que implementa os métodos *findMetaObjectFor:*, e *activateMetaObjects:forMessage:*.

5.4.3 Classes do protótipo

As seguintes classes foram modeladas em ***ATeste***, a fim de implementar a interface com o usuário, o encapsulamento das informações relevantes, bem como os mecanismos para reflexão, interceptação e análise dos métodos selecionados para teste:

- InterfaceDeTeste:
  InterfaceDeTeste é uma subclasse de ApplicationModel. Define a interface de comunicação com o usuário a fim de verificar os métodos que devem ser testados, os requisitos de teste e a apresentação dos resultados.
- InfoTeste:
  Esta classe fornece suporte para administrar as informações de entrada e saída, definindo a interface com as classes que necessitam destas informações para a interceptação de mensagens, associação de meta-objetos e execução do teste. InfoTeste é uma subclasse de Model.
- MetaGerente:
  Esta é uma classe especializada da classe abstrata MetaObjectManager (framework *Luthier MOPs*), que implementa os mecanismos específicos de associação e ativação de meta objetos através dos métodos *findMetaObjectFor:* e *activateMetaObjects:forMessage:*.
- MetaTeste:
  A classe MetaTeste provê a implementação do método *handleMessage:* para tratar as mensagens refletidas, sendo uma especialização da classe abstrata MetaObjectClass do framework *Luthier MOPs*. O método *handleMessage*: define as ações que devem ser tomadas quando uma mensagem é interceptada, ativando métodos da classe EstadoDoObjeto, que realizam as verificações necessárias.
- EstadoDoObjeto:
  Esta classe recebe da classe MetaTeste as informações acerca da mensagem interceptada e, a partir das informações sobre o objeto receptor, busca o estado deste objeto. Implementa o *teste de caminho* do objeto selecionado para análise se o método ativado pela mensagem interceptada foi selecionado para teste, montando uma lista de mensagens refletidas contendo o estado do objeto neste momento. É responsável também pela geração de uma lista com o caminho completo da aplicação sob teste, ou seja, de todas mensagens recebidas pela(s) classe(s)

monitoradas, mesmo que estas não ativem métodos selecionados para teste, com os respectivos estados dos objetos receptores das mensagens.

5.5 Descrição das classes de *Ateste*

Nesta seção são apresentadas as classes implementadas em Ateste. São descritos os principais comportamentos e variáveis de instância modelados em cada classe.

5.5.1 Interface com o Usuário - Classe *InterfaceDeTeste*

A interface de comunicação com o usuário é definida pela classe InterfaceDeTeste. A funcionalidade básica desta classe é gerenciar as informações recebidas do usuário, a fim de verificar os métodos que devem ser testados e os requisitos de teste, e a posterior apresentação dos resultados obtidos.

A coordenação da comunicação entre os campos da interface e o modelo de dados é facilitada pela herança de atributos e do comportamento da classe ApplicationModel. As informações recebidas do usuário são tratadas em interfaces (janelas) desenvolvidas no próprio VisualWorks, através da ferramenta Canvas, que auxilia o desenvolvimento de desenhos de telas para interação com o usuário. Estas janelas são apresentadas na seção 5.6, que mostra o funcionamento do protótipo.

As principais variáveis de instância e métodos da classe InterfaceDeTeste estão ilustrados na fig.5.3.

FIGURA 5.3 - Classe InterfaceDeTeste

A variável de instância *categs* refere-se ao campo de entrada que contém a categoria digitada pelo usuário. A lista *listaCategs* é apresentada na tela após a digitação de cada categoria, para visualização do nome de todas as categorias digitadas.

Conforme as categorias vão sendo informadas, o nome da categoria e todas as classes que compõem esta categoria são apresentadas na tela através da lista

listaClasses, na forma de pares categoria/classe. Esta lista é mostrada na tela para que sejam selecionadas as classes que se deseja monitorar.

A lista *listaMétodos* permite a apresentação da lista de métodos (como duplas classe/método) ao usuário, a fim de que este selecione os métodos para teste.

As listas *listaCaminhoMonit* e *listaCaminhoExecução* permitem mostrar ao usuário a lista dos métodos interceptados pelo sistema, com o estado dos objetos no momento desta interceptação, e todo o caminho de execução da aplicação sob teste, fornecendo uma visão geral do caminho (com os métodos executados e o estado dos objetos envolvidos nesta execução), respectivamente.

Os métodos principais da classe InterfaceDeTeste são descritos a seguir.

5.5.1.1 Inicialização da estrutura de dados da classe

O método *inicializa*, apresentado na listagem 5.1, é responsável pela inicialização da estrutura de dados. Neste momento uma instância da classe InfoTeste é criada, contendo as informações da instância de interface, e o campo de entrada do nome da categoria, *categs*, recebe um valor inicial nulo.

```
inicializa
"Inicializa estrutura de dados"

infoTeste := InfoTeste new.
infoTeste interfaceDeTeste: self.
categs := nil.
```

LISTAGEM 5.1 - Método *inicializa*

5.5.1.2 Inclusão e Verificação de Categorias para Teste

O método *incluiCateg*, mostrado na listagem 5.2, realiza o controle da categoria digitada, ativando o método *verificaCateg* (vide listagem 5.3), que faz as verificações necessárias com relação à categoria digitada, e mostrando nas respectivas janelas, a lista de categorias digitadas e a lista de classes correspondentes (classes que pertencem às categorias digitadas) atualizadas. Após a execução deste método, o usuário pode digitar uma nova categoria ou selecionar as classes que deseja testar através da lista *listaClasses*.

```
incluiCateg
"Ativa a verificação da categoria digitada e
atualiza a lista de categorias/classes"

self verificaCateg.
self listaCategs list: infoTeste nomeCategs.
self listaClasses list: infoTeste nomeClasses.
self categs value: nil
```

LISTAGEM 5.2 - Método *incluiCateg*

O método *verificaCateg* realiza as verificações pertinentes com relação à categoria digitada: se esta já está inclusa na lista de categorias digitadas (inclusão de categoria repetida constitui erro), e se é uma categoria válida (se existe alguma classe na biblioteca do Smalltalk pertencente a esta categoria).

Após as verificações, se não ocorrerem erros, são ativados os métodos *registraCategs* e *registraCateg:eClasse:*, ambos implementados na classe InfoTeste. O primeiro tem como função guardar as categorias válidas digitadas, enquanto a funcionalidade do segundo método é armazenar as classes agrupadas nestas categorias (na forma de pares categoria/classe). Estas informações são armazenadas, respectivamente, nas coleções ordenadas *nomeCategs* e *nomeClasses* da classe InfoTeste.

```
verificaCateg
"Verifica se a categoria digitada é válida"

| achou |
achou := false.
(infoTeste nomeCategs includes: categs
                                    value asSymbol)
   ifTrue:
      [^Dialog warn: 'Categoria ja informada'].
Smalltalk classNames do:
   [:umaClasse |
   (Smalltalk at: umaClasse asSymbol) category =
                        categs value asSymbol
      ifTrue:
         [infoTeste registraCateg: categs
                                    value asSymbol
                        eClasse: umaClasse
                                    value asSymbol.
         achou = false
            ifTrue:
               [infoTeste registraCategs: self
                        categs value asSymbol.
            achou := true]]].
achou = false
   ifTrue:
      [^Dialog warn: 'Categoria invalida']
```

LISTAGEM 5.3 - Método *verificaCateg*

5.5.1.3 Seleção de Classes e Métodos

O usuário pode selecionar as classes que deseja testar dentre as classes apresentadas na janela que mostra a lista *listaClasses*.

O método *selecClasses*, mostrado na listagem 5.4, verifica se foram selecionadas classes para teste nesta lista. Se nenhuma classe foi selecionada, é apresentada uma mensagem advertindo o usuário deste fato. Caso contrário, é realizada a busca de todos métodos encapsulados nas classes selecionadas, na biblioteca do Smalltalk.

Estes métodos e suas respectivas classes (na forma de pares classe/método) são armazenados em uma coleção ordenada na classe InfoTeste (*nomeMétodos*) e mostrados ao usuário através da lista listaMétodos, a fim de que este selecione os métodos mais críticos, ou seja, aqueles que serão interceptados para visualização do estado do objeto, antes e após sua execução.

De forma idêntica ao método descrito anteriormente, o método *selecMétodos* (listagem 5.5) verifica se foram selecionados métodos para teste na lista *listaMétodos*. Se nenhum método foi selecionado, é apresentada uma mensagem ao usuário caso contrário, os métodos selecionados para teste são guardados em uma

coleção ordenada na classe InfoTeste (*selecParaReflexão*), que contém o nome dos métodos selecionados e a classe à qual pertencem.

```
selecClasses
"Busca dos métodos de cada classe selecionada"

| catClasMet     "tupla categoria/classe/método"
  catClas        "par categoria/classe" |
catClas:= Array new: 2.
listaClasses selections isEmpty
  ifTrue: [^Dialog warn: 'Nenhuma classe foi selecionada'].
infoTeste nomeMétodos removeFirst: infoTeste
                                        nomeMétodos size.
self listaClasses selections do:
  [:umElemento |
  infoTeste buscaElem: umElemento paraArray: catClas.
  (Smalltalk at: (catClas at: 2) asSymbol) selectors do:
    [:umMétodo |
    catClasMet:= (catClas at: 1) asString,' ',(catClas at: 2)
                        asString , ' ' , umMétodo asString.
    infoTeste nomeMétodos add: catClasMet]].
self listaMétodos list: infoTeste nomeMétodos
```

LISTAGEM 5.4 - Método *selecClasses*

```
selecMétodos
"Armazena os métodos selecionados e chama o método que
ativa a reflexão."

| catClasMet     "tupla categoria/classe/método"
  clasMet        "par classe/método"
| catClasMet:= Array new: 3.
listaMétodos selections isEmpty
  ifTrue: [^Dialog warn: 'Nenhum método foi selecionado'].
infoTeste selecParaReflexão removeFirst:
                infoTeste selecParaReflexão size.
self listaMétodos selections do:
  [:umElemento |
  infoTeste buscaElem: umElemento
                paraArray: catClasMet.
  clasMet:= (catClasMet at: 2) asString, ' ',
        (catClasMet at: 3) asString.
  infoTeste selecParaReflexão add: clasMet]
```

LISTAGEM 5.5 - Método *selecMétodos*

5.5.1.4 Solicitação da Ativação e Desativação da Reflexão

Os métodos *ativaReflexão* e *desativaReflexão* são responsáveis pelo início dos processos de ativação e desativação de meta-objetos para reflexão das classes/ métodos selecionados.

O primeiro método, após chamar o método *selecMétodos*, ativa o método *ativaMO*, implementado na classe InfoTeste, que irá solicitar a reflexão, em meta-objetos, das classes selecionadas para teste. Após, informa ao usuário que a reflexão foi realizada e que a aplicação a ser testada pode ser executada. Este método também abre a

janela de monitoração de estado dos objetos, onde será apresentado o estado dos objetos antes e após a execução dos métodos selecionados, através da lista *listaCaminhoMonit*.

O segundo método chama o método *desativaMO*, também implementado na classe InfoTeste, que desencadeia o processo de desativação dos meta-objetos.

A desativação da reflexão de meta-objetos pode ser solicitada também em conjunto com o final da aplicação através do método *desativaEFim*, que por sua vez, chama o método *desativaReflexão* e finaliza a execução de ***Ateste***.

5.5.1.5 Apresentação do Caminho de Execução

Além da monitoração do estado dos objetos antes e após a execução dos métodos selecionados para teste, é armazenado o estado dos objetos durante toda a execução da aplicação sob teste, antes e após a execução de cada método. Estas informações são apresentadas ao usuário através da lista *listaCaminhoExec*, se forem solicitadas.

O método encarregado de ativar a janela para que esta lista seja mostrada é *mostraCaminhoExec*, sendo que a chamada deste método é realizada na classe InfoTeste e a montagem desta lista é realizada na classe EstadoDoObjeto.

5.5.2 Manipulação de Informações - Classe *InfoTeste*

A administração da estrutura de dados necessária para o processamento das atividades de teste é implementada na classe InfoTeste, ilustrada na fig.5.4. Esta classe gerencia as informações que são recebidas do usuário, bem como as informações que são enviadas a ele.

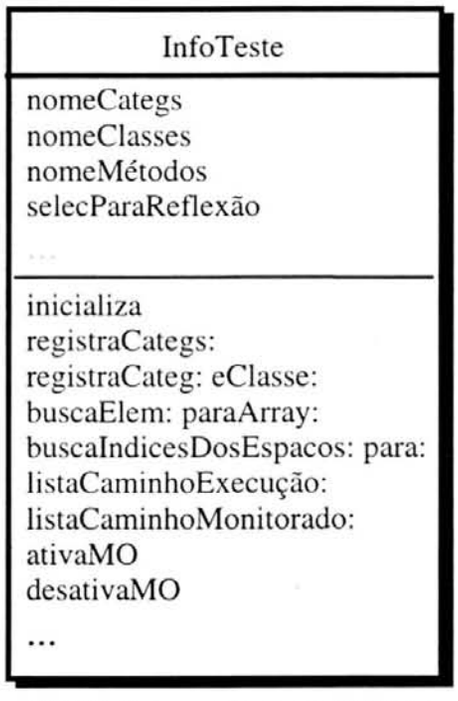

FIGURA 5.4 - Classe InfoTeste

InfoTeste contém as variáveis de instância *nomeCategs*, *nomeClasses* e *nomeMetodos*, que são listas ordenadas que armazenam informações que vão ser mostradas ao usuário, bem como a variável *selecParaReflexão*, que guarda o nome dos métodos que devem ser monitorados. Possui como comportamentos principais, além da administração de informações, a ativação e desativação da reflexão das classes a serem

testadas, solicitadas pela classe InterfaceDeTeste. Nas próximas subseções são apresentados os métodos considerados de maior interesse para o entendimento das funcionalidades do protótipo.

5.5.2.1 Ativação e Desativação da Reflexão

Os métodos *ativaMO* e *desativaMO* são responsáveis pela ativação e desativação da reflexão das classes a serem monitoradas (listagens 5.6 e 5.7).

Para a ativação de meta-objetos, a lista ordenada que contém pares de classe/método selecionados para monitoração (*selecParaReflexão*) é percorrida, e é solicitada à classe MetaGerente, pelo envio da mensagem *reflectInstancesOf:on:*, que ative a reflexão de cada classe encontrada nesta coleção.

A desativação da reflexão das classes pode ser solicitada a qualquer momento à classe MetaGerente. Ela é processada através dos métodos *reflectedClasses* e *keys*, que retornam as classes refletidas e seus identificadores, respectivamente, a fim de que as mensagens *unreflectClass*: e *unreflect:*, que desativam a reflexão da classe e a retiram da lista de classes refletidas, possam ser enviadas com as identificações destas classes como parâmetros.

Os métodos referenciados no dois parágrafos anteriores são herdados da classe MetaObjectManager.

O método *buscaElem:paraArray:*, chamado no método ativaMO, localiza, a partir de uma seqüência de caracteres, os elementos separados por espaços, armazenando-os em um *array*.

```
ativaMO
"Ativa meta-objetos"

| clasMet   "par classe/método"
  classeAnt mo estadoDoObjeto |
estadoDoObjeto := EstadoDoObjeto new.
estadoDoObjeto inicializa.
estadoDoObjeto infoTeste: self.
clasMet:= Array new: 2.
classeAnt := ' '.
self selecParaReflexão do:
  [:umElemento |
  self buscaElem: umElemento paraArray: clasMet.
  classeAnt  =  (clasMet at: 1)
    ifFalse:
      [classeAnt := (clasMet at: 1).
        mo := MetaGerente reflectInstancesOf:
                (Smalltalk at: (classeAnt) asSymbol)
                 on: MetaTeste.
        mo estadoDoObjeto: estadoDoObjeto]].
MetaGerente startReflect
```

LISTAGEM 5.6 - Método *ativaMO*

```
desativaMO
"Desativa meta-objetos"

(MetaGerente reflectedClasses keys) do:
  [:umaClasse |
  MetaGerente unreflectClass: umaClasse.
  MetaGerente unreflect: umaClasse].
MetaGerente stopReflect
```

LISTAGEM 5.7 - Método *desativaMO*

5.5.2.2 Apresentação do estado dos Objetos

A apresentação do estado dos objetos é feita de duas formas:

1. É mostrado o estado dos objetos antes e após a execução dos métodos selecionados para monitoração, e
2. É mostrado o estado dos objetos antes e após a execução de todos os métodos ativados durante a execução da aplicação sob teste, a partir de mensagens enviadas às classes monitoradas.

A solicitação para que estas informações sejam mostradas é realizada através dos métodos *listaCaminhoExecução* e *listaCaminhoMonitorado*.

5.5.3 Gerenciamento de Meta-Objetos - Classe *MetaGerente*

Na classe MetaGerente é implementado o comportamento para associação e ativação de meta-objetos, cujo comportamento genérico é herdado da classe MetaObjectManager.

Esta classe não possui variáveis. A próxima subseção lista os métodos implementados (fig.5.5).

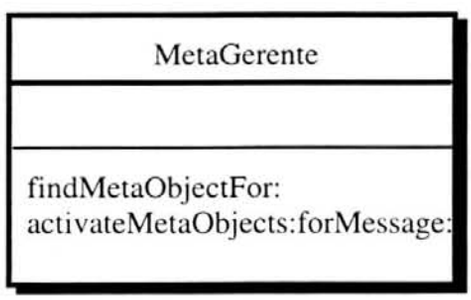

FIGURA 5.5 - Classe MetaGerente

5.5.3.1 Busca e Ativação de Meta-Objetos

Quando um objeto refletido recebe uma mensagem, esta é capturada e desviada para a classe MetaGerente. É realizada então, uma busca para verificar qual meta-objeto está associado com o objeto que recebeu a mensagem. Esta funcionalidade é implementada através do código da listagem 5.8,no método *findMetaObjectFor:*.

```
FindMetaObjectFor: anObject
"Procura meta-objeto"

| mo |
mo := ReflectedClasses at: anObject class
                        ifAbsent: [nil].
^mo
```

LISTAGEM 5.8 - Método *findMetaObjectFor:*

A classe MetaGerente possui, também, como funcionalidade, a transferência do controle para o meta-objeto associado, através do método *activateMetaObjects:forMessage:*, descrito na listagem 5.9, enviando a este meta-objeto a mensagem *handleMessage:*, cujo parâmetro é uma instância da classe ReflectedMessage, contendo as informações sobre a mensagem refletida.

A partir destas informações o meta-objeto pode realizar computações próprias, que devem ser definidas no método *handleMessage:*. O método *handleMessage:* é implementado na classe MetaTeste.

```
activateMetaObjects: mo
forMessage: aReflectedMessage
"Ativa meta-objeto"

mo handleMessage: aReflectedMessage
```

LISTAGEM 5.9 - Método *activateMetaObjects:forMessage:*

5.5.4 Interceptação de Mensagens - Classe *MetaTeste*

A classe MetaTeste, (fig.5.6), herda os comportamentos da classe abstrata MetaObjectClass, que possui como funcionalidade básica o tratamento das mensagens recebidas pelo objeto refletido.

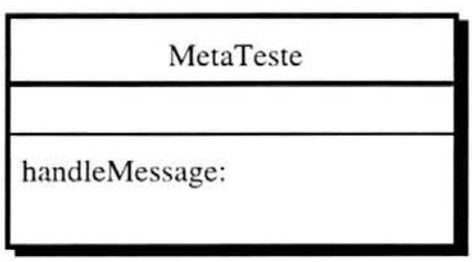

FIGURA 5.6 - Classe MetaTeste

Esta classe especializa as ações a serem executadas quando ocorrer a interceptação de mensagens, que são realizadas através do método *handleMessage*: cuja descrição encontra-se na listagem 5.10. Este método é ativado pela classe MetaGerente quando uma mensagem é refletida, sendo enviado como argumento uma instância da classe ReflectedMessage, contendo as informações acerca da mensagem refletida.

O método *handleMessage*: gerencia o controle das ações adicionais a serem executadas na aplicação sob teste. A fim de executar a monitoração do estado dos objetos, o método transfere o controle para a classe EstadoDoObjeto, enviando-lhe a mensagem *buscaEstado:*, que se encarrega da implementação dos comportamentos requeridos para a execução destas atividades de auxílio ao teste.

```
handleMessage: msg

((msg selector asString) = 'printOn:' )
   ifTrue: [^msg send].
estadoDoObjeto buscaEstado: msg.
```

LISTAGEM 5.10 - Método *handleMessage:*

5.5.5 Busca do Estado do Objeto- Classe *EstadoDoObjeto*

MetaTeste possui, como subclasse, a classe EstadoDoObjeto, ilustrada na fig.5.7, que também herda comportamento da classe abstrata MetaObjectClass.

A funcionalidade básica desta classe é a implementação das atividades de auxílio ao teste, através da monitoração do estado dos objetos e da transformação dos atributos do objeto em dados disponíveis para visualização.

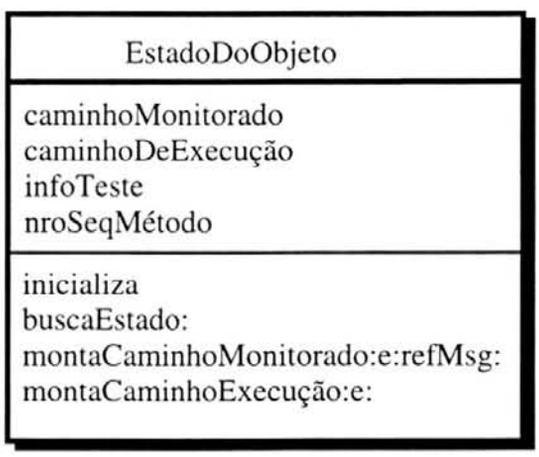

FIGURA 5.7 - Classe EstadoDoObjeto

5.5.5.1 Busca e Mostra do Estado dos Objetos

A partir das classes e métodos selecionados, é apresentado ao testador o estado do objeto antes e após a execução de um dado método, sempre que um objeto de uma classe selecionada receber uma mensagem que ativa um método também selecionado para monitoração.

Adicionalmente é armazenado o estado do objeto antes e após a execução de todos os métodos ativados durante a execução da aplicação sob teste, gerando uma lista com o caminho de teste completo, cuja visualização pode ser solicitada a qualquer momento durante a execução da aplicação sob teste. Este caminho de teste tem por objetivo fornecer ao testador uma visão geral do caminho de execução da aplicação, dando-lhe condições de verificar eventuais desvios ou estados falhos.

A listagem 5.11 mostra o processamento realizado para acesso e disponibilização das informações sobre o estado dos objetos da aplicação sob teste, implementado no método *buscaEstado:*.

Este método ativa o procedimento descrito na listagem 5.12 (método *montaCaminhoMonitorado:e:refMsg:)*, que verifica se uma determinada mensagem interceptada ativa um método selecionado para monitoração. Caso positivo, são adicionadas as informações atuais na lista *caminhoMonitorado*.

A listagem 5.13 apresenta o método *montaCaminhoExecução:e:*, também chamado pelo método *buscaEstado:*, que tem por função adicionar as informações na lista do caminho completo de execução da aplicação sob teste.

A solicitação para que estas informações sejam mostradas é realizada pela classe InfoTeste, através dos métodos *listaCaminhoMonitorado:* e *listaCaminhoExecução:*.

```
buscaEstado: msg
"Busca estado dos objetos monitorados"

|linhaDadosIntercep linhaDadosVar dadosDeVariaveis nomes
valorVar  |
dadosDeVariaveis := OrderedCollection new.
linhaDadosVar := ' **  ANTES DA Execução do Metodo  ** '
                                                printString.
dadosDeVariaveis add: linhaDadosVar.
linhaDadosIntercep:= nroSeqMétodo value printString,
     msg selector, ' - ', msg receiverClass name asString,
     ' / ',msg senderClass name asString,' / ',
     msg homeReceiverClass name asString.
nomes := msg receiverClass instVarNames.
1 to: nomes size do:
    [:cont |
   valorVar := msg receiver instVarAt: (cont + 1).
   linhaDadosVar := (nomes at: cont) asString , ' =' ,
                                valorVar printString.
   dadosDeVariaveis add: linhaDadosVar asString].
self montaCaminhoExecução: linhaDadosIntercep e:
                                dadosDeVariaveis asValue.
self montaCaminhoMonitorado: linhaDadosIntercep e:
                                dadosDeVariaveis refMsg: msg.
   nroSeqMétodo := nroSeqMétodo + 1.
   msg send.
linhaDadosVar := ' **  APÓS  A  Execução do Método  ** '
                                                printString.
dadosDeVariáveis := OrderedCollection new.
dadosDeVariáveis2 add: linhaDadosVar.
linhaDadosIntercep:= msg selector, '- ', msg receiverClass
     name asString,' / ', msg senderClass name asString, msg
     homeReceiverClass name asString,' - '.
1 to: nomes size do:
    [:cont |
   valorVar := msg receiver instVarAt: (cont + 1).
   linhaDadosVar := (nomes at: cont) asString , ' = ' ,
                                valorVar printString.
   dadosDeVariaveis2 add: linhaDadosVar].
self montaCaminhoExecução: linhaDadosIntercep e:
                                dadosDeVariaveis2 asValue.
self montaCaminhoMonitorado: linhaDadosIntercep e:
                                dadosDeVariaveis2 refMsg: msg.
```

LISTAGEM 5.11 - Método *buscaEstado*:

```
montaCaminhoMonitorado: dadosIntercep e:
dadosDeVariaveisv refMsg: msg
"monta uma lista com o caminho de execução dos metodos
selecionados para teste"

|array |
array := Array new: 2.
infoTeste selecParaReflexao do:
   [:umElemento |
   infoTeste buscaElem: umElemento paraArray: array.
   (msg selector  asString =  (array at: 2) and:
                  [(msg senderClass) ~~ AspectAdaptor])
      ifTrue:
         [self caminhoMonitorado add: dadosIntercep.
         self caminhoMonitorado add: dadosDeVariaveis .
         self caminhoMonitorado add:
         '_____________________________________________
_____________'.
         infoTeste listaCaminhoMonitorado:
                              self caminhoMonitorado.]].
```

LISTAGEM 5.12 - Método *montaCaminhoMonitorado:e:refMsg:*

```
montaCaminhoExecução: dadosIntercep e: dadosDeVariaveis
"monta uma lista com o caminho de execução"

self caminhoDeExecução add: dadosIntercep.
self caminhoDeExecução add: dadosDeVariaveis.
self caminhoDeExecução add:
'_____________________________________________
_____________'.
infoTeste listaCaminhoExecução: self caminhoDeExecução
```

LISTAGEM 5.13 - Método *montaCaminhoExecução:e:*

5.6 Funcionamento de *ATeste* para um Estudo de Caso

Para demonstrar o funcionamento de ***ATeste***, é apresentado um estudo de caso. A monitoração de estados é realizada sobre objetos de classes da categoria Examples-VWTutorial.

Com o objetivo de auxiliar o entendimento desta demonstração, inicialmente são descritas a estrutura e funcionalidades principais da aplicação utilizada para este estudo de caso.

A categoria Examples-VWTutorial agrupa as classes Check, Checkbook, e CheckbookInterface, que implementam uma aplicação bancária de controle de cheques.

As principais transações desta aplicação são realizadas de forma interativa com o usuário através de uma janela principal que apresenta uma lista de cheques e o saldo atual da conta, e disponibiliza as ações: depositar valores, eliminar cheques selecionados da lista de cheques, e gravar novos cheques emitidos na lista de cheques. Estas transações são explicitadas a seguir:

- Depósito: Realiza o depósito de um valor, informado pelo usuário, acumulando-o em uma variável de instância (*balance*) da classe Checkbook, que representa o saldo da conta. O saldo da conta é atualizado na janela principal da aplicação.
- Grava novo cheque: Adiciona as informações relativas a um cheque emitido, informadas pelo usuário, em uma lista de cheques (*register*) da classe Checkbook, subtraindo o valor correspondente da variável *balance*. Este novo cheque é adicionado na lista de cheques apresentada na janela principal, e o saldo atualizado.
- Cancela cheque: A partir de um cheque selecionado na lista de cheques da janela principal de comunicação com o usuário, elimina este cheque da lista ordenada *register*, adicionando o valor correspondente na variável *balance*. Paralelamente é atualizada a lista de cheques apresentada ao cliente, retirando-se as informações relativas ao cheque eliminado, e o saldo é atualizado.

Para testar uma aplicação, o usuário de ***ATeste*** deve inicialmente fornecer ao sistema as informações referentes à(s) categoria(s) que agrupam as classes que implementa(m) esta aplicação.

O primeiro passo é ativar a ferramenta no ambiente VisualWorks. Este ambiente encontra-se disponível tanto para estações de trabalho Sun como para uso em microcomputadores PC.

Na opção "Tools" do menu da janela principal do VisualWorks (fig.5.8), a qual é visualizada aberta na fig.5.9, está disponibilizada a chamada para a ferramenta. Após a ativação de ***ATeste***, é apresentada a janela principal de comunicação do protótipo, a qual é mostrada na fig.5.10.

Com a ativação da ferramenta inicia-se a primeira etapa do processo de teste, que refere-se à interação do usuário para fornecer informações para ativação da reflexão.

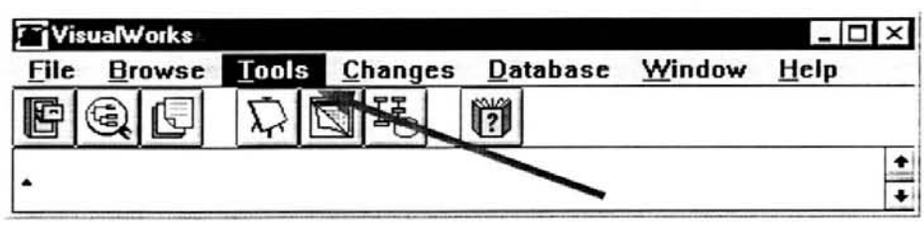

FIGURA 5.8 - Menu Principal do VisualWorks

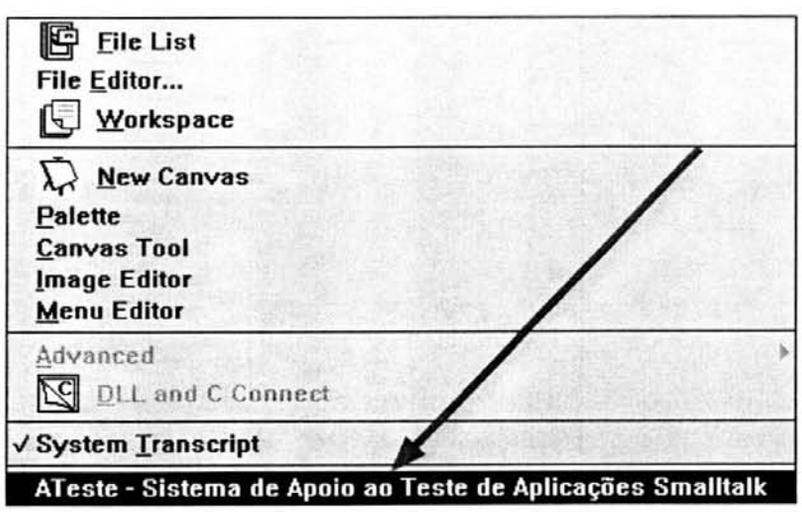

FIGURA 5.9 - Menu da Opção Tools

Deve ser então digitado o nome da categoria no campo de entrada de categorias. Após a verificação da categoria, o que é realizado conforme descrição apresentada na seção 5.5.1.2, se nenhum erro ocorrer, o nome da categoria digitada é acrescentado à lista de categorias informadas desta janela. Todas as classes agrupadas nesta categoria são recuperadas e adicionadas à lista de classes desta mesma janela, na forma de pares categoria/classe para identificar a que categoria pertence cada classe.

No caso de ocorrência de algum dos erros previstos (categoria inválida ou categoria já digitada), é apresentada uma advertência alertando sobre o erro cometido.

Após o procedimento descrito anteriormente, pode ser digitada uma nova categoria, executando-se o mesmo processo, ou ir para o próximo passo.

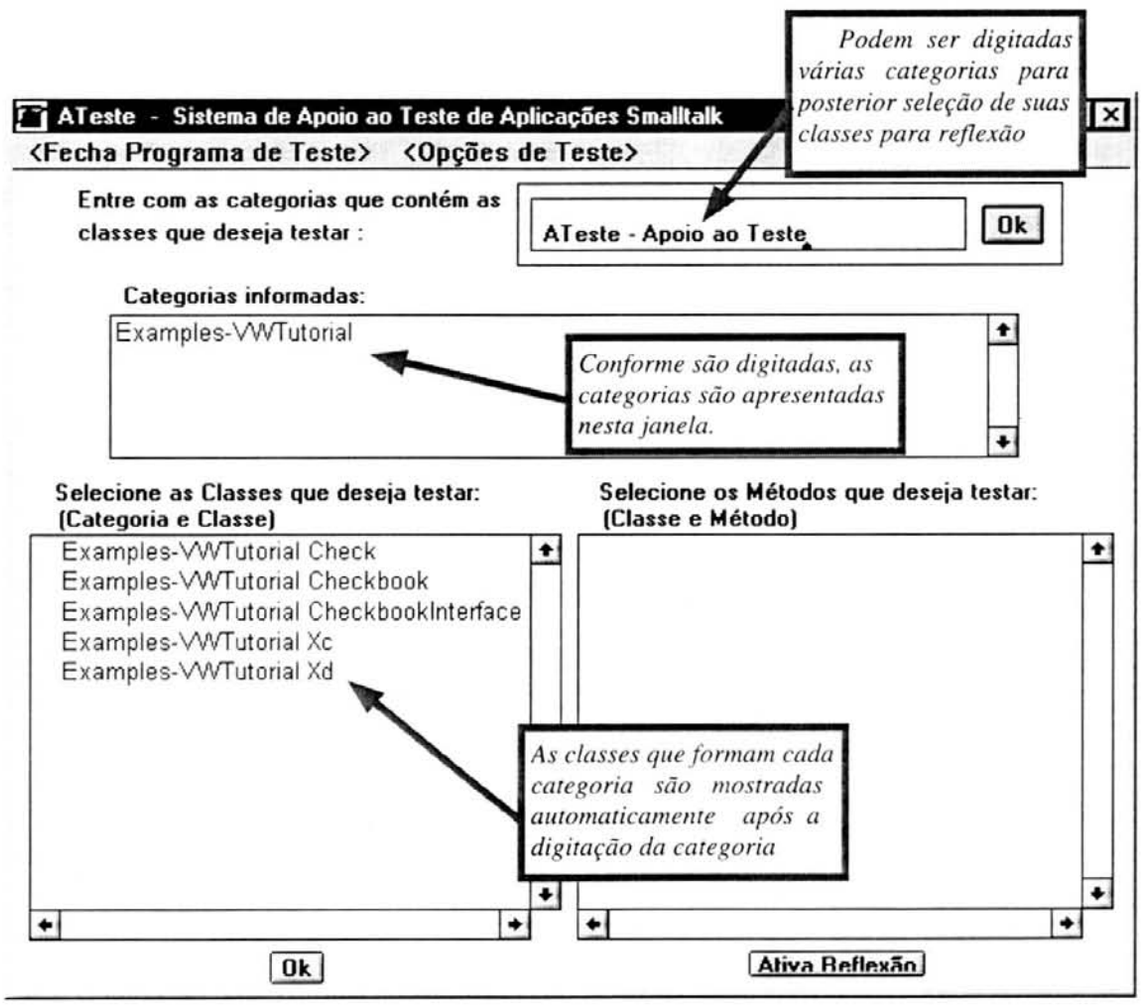

FIGURA 5.10 - Janela de Comunicação Principal de ATeste

A(s) classe(s) que se deseja monitorar são selecionadas pressionando-se o botão de "Ok" correspondente. Este botão de ação ativa o método *selecClasses*, descrito na seção 5.5.1.3.

Se nenhuma classe for selecionada é apresentada uma advertência. Caso contrário, é realizada a busca de todos os métodos encapsulados na(s) classe(s) selecionada(s), sendo estes mostrados na janela de comunicação na forma de pares classe/método, identificando a classe a qual pertence cada método, para a seleção dos métodos que se deseja monitorar. Conforme a figura 5.11, os métodos *recordCheck*: e *deposit*:, da classe Checkbook foram selecionados para serem refletidos.

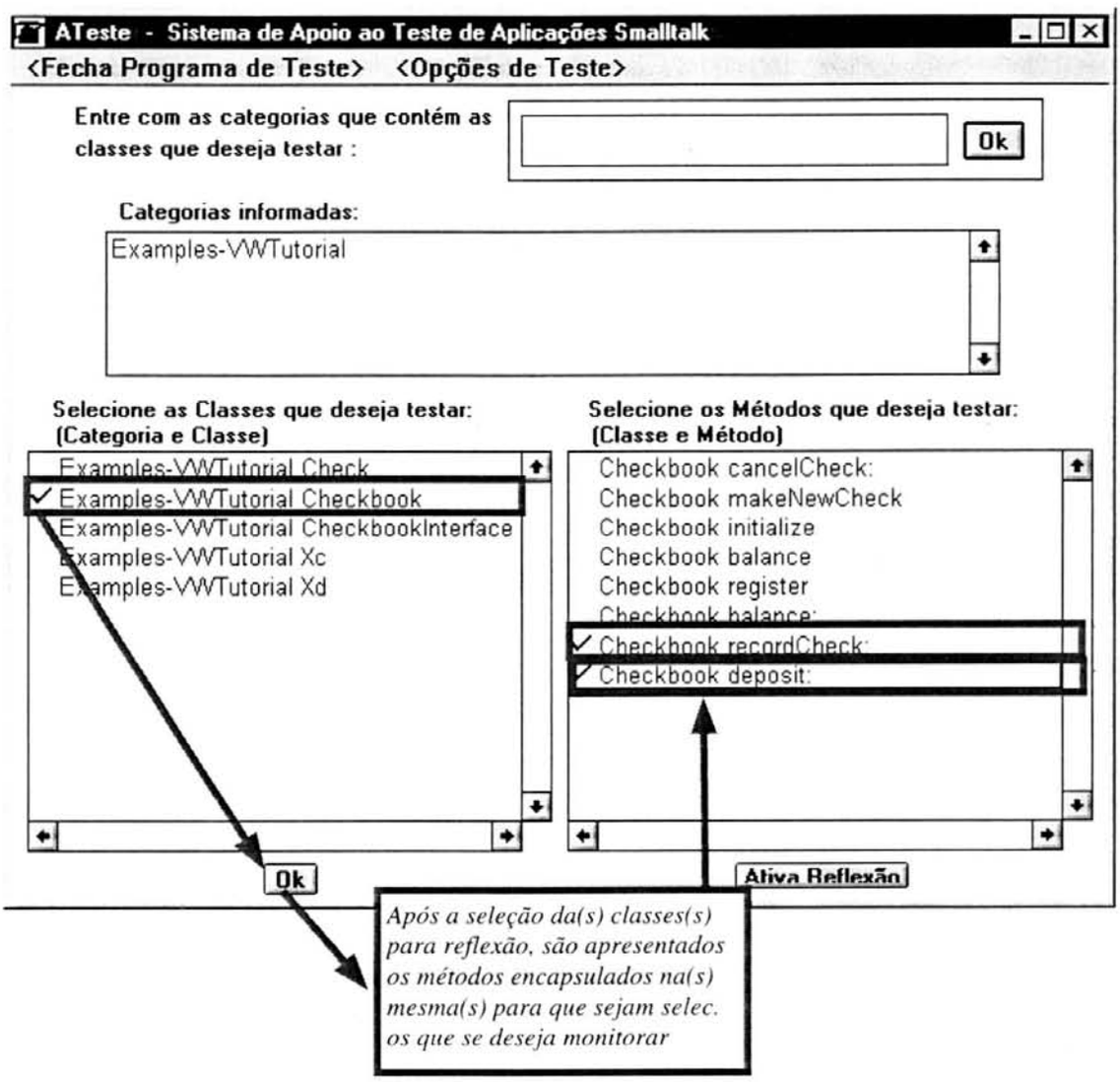

FIGURA 5.11 - Mostra Métodos das Classes Selecionadas

Após a seleção dos métodos, o botão "Ativa Reflexão" deve ser pressionado. Esta ação causa a ativação dos métodos *selecMétodos*, descrito na seção 5.5.1.3, e *ativaMO*, fazendo com que os métodos selecionados sejam armazenados e a reflexão das classes selecionadas seja ativada com base nestas informações. É apresentada uma mensagem informando que a reflexão foi registrada e que a aplicação a ser testada pode ser executada, conforme pode ser visto na figura 5.12. A seção 5.5.1.4 descreve o processo de ativação e desativação da reflexão.

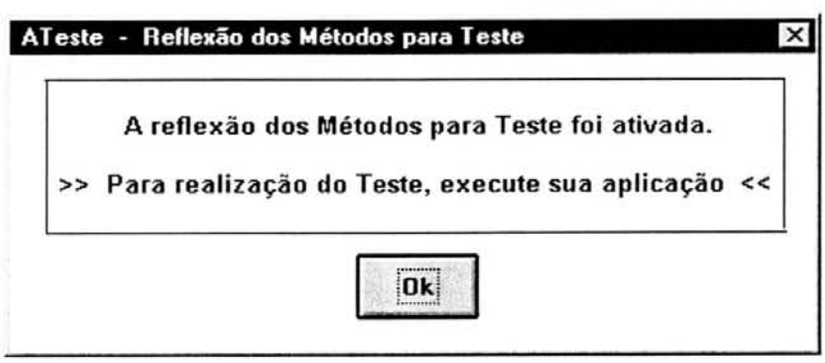

FIGURA 5.12 - Comunicação de Ativação da Reflexão

A janela que mostra a monitoração da execução dos métodos selecionados é, então, apresentada na tela. Ao executar a aplicação, o estado dos objetos, pertencentes às classes selecionadas para teste, que recebem mensagens que ativam métodos, também selecionados, é mostrado nesta janela, antes e após a execução destes métodos, com algumas informações adicionais.

Nesta janela é disponibilizada a opção de visualizar o caminho de execução da aplicação nas classes refletidas, através do botão "Caminho de Execução da Aplicação". Quando esta opção é selecionada, é mostrada outra janela de monitoração da execução da aplicação, a qual contém o estado dos objetos pertencentes às classes selecionadas para teste que receberam mensagens, antes e após a execução dos métodos ativados por estas mensagens, com algumas informações adicionais, da mesma forma que na janela anterior.

As janelas que apresentam o *caminho de execução dos métodos selecionados para teste* e o *caminho de execução da aplicação* são mostradas posteriormente como resultado da monitoração da aplicação de controle de cheques.

A reflexão das classes selecionadas, juntamente com o armazenamento das informações dos métodos a serem monitorados e a apresentação da janela de monitoração, encerram a primeira etapa de utilização da ferramenta.

A segunda etapa independe de qualquer nova interação do usuário com a ferramenta e consiste na execução da aplicação sob teste, e na comparação do estado dos objetos monitorados, que são apresentados nas duas últimas janelas mostradas, com o estado esperado.

Prosseguindo com a demonstração do estudo de caso, é ativada a aplicação de controle de cheques. A figura 5.13 mostra duas formas para realizar esta ativação: pressionando o botão de *start* na janela de recursos do VisualWorks após a seleção da janela principal da aplicação, ou executando o comando "CheckBookInterface open"na área de trabalho, o que causa a abertura da janela principal da aplicação.

A figura 5.14 apresenta a janela de comunicação principal da aplicação de controle de cheques, onde as transações de depósito e de gravação de novos cheque são ativadas, considerando que a classe Checkbook, e os métodos *deposit* (deposita um valor) e *recordCheck* (grava um novo cheque) estão selecionados para monitoração. Nesta figura é demonstrado que foi efetuado um depósito de dez dólares

A gravação de um novo cheque e sua repercussão na lista de cheques e no saldo é demonstrada na figura 5.15. É apresentada a janela de entrada de novos cheques, sendo digitado um cheque para "Uma Pessoa" com o valor de cinco dólares, e a sua repercussão na janela principal, atualizando a lista de cheques e o saldo, após pressionado o botão de "Ok".

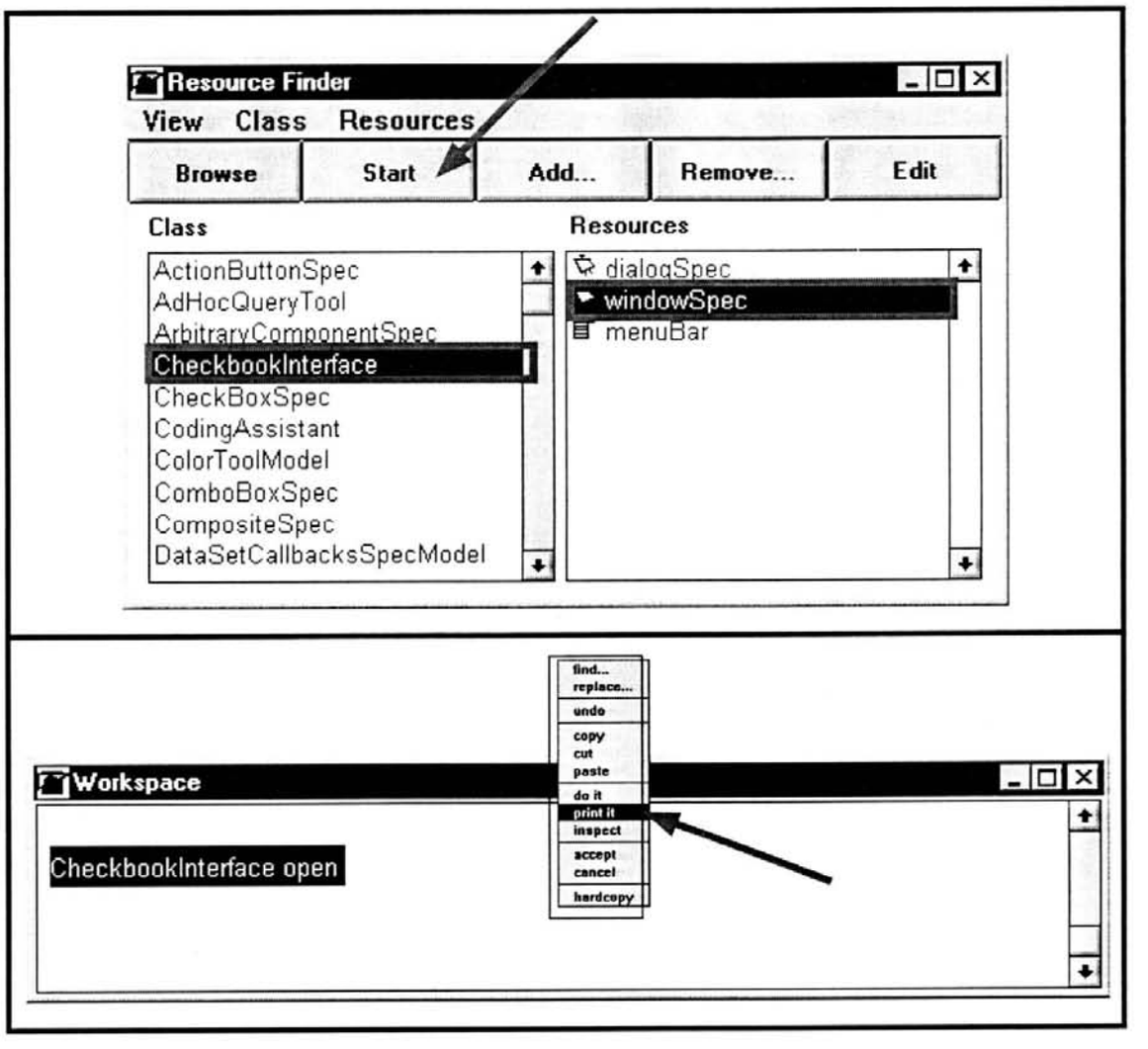

FIGURA 5.13 - Ativação da Aplicação de Controle de Cheques

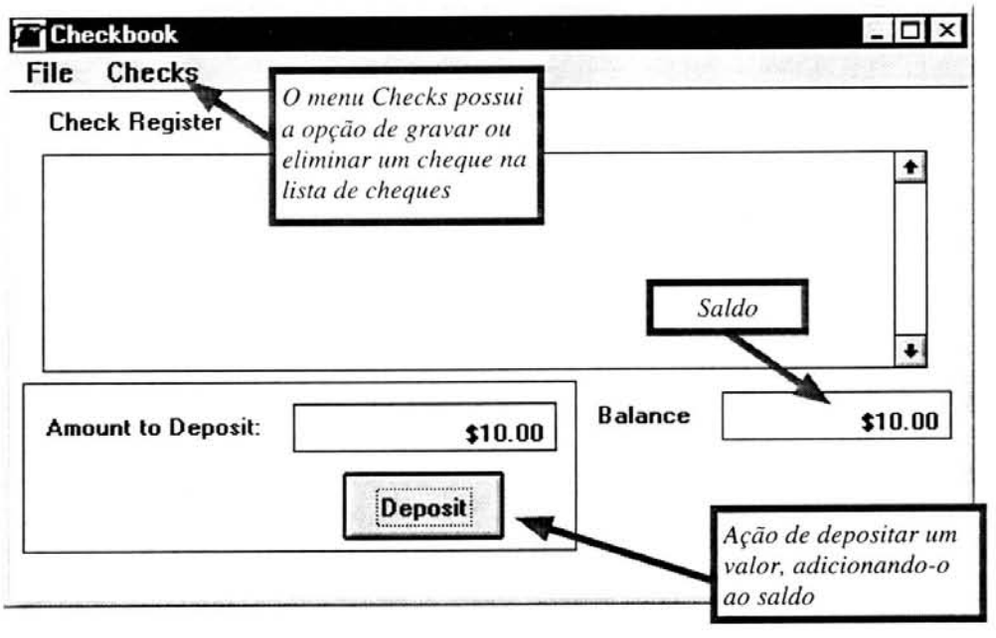

FIGURA 5.14 - Aplicação de Controle de Cheques

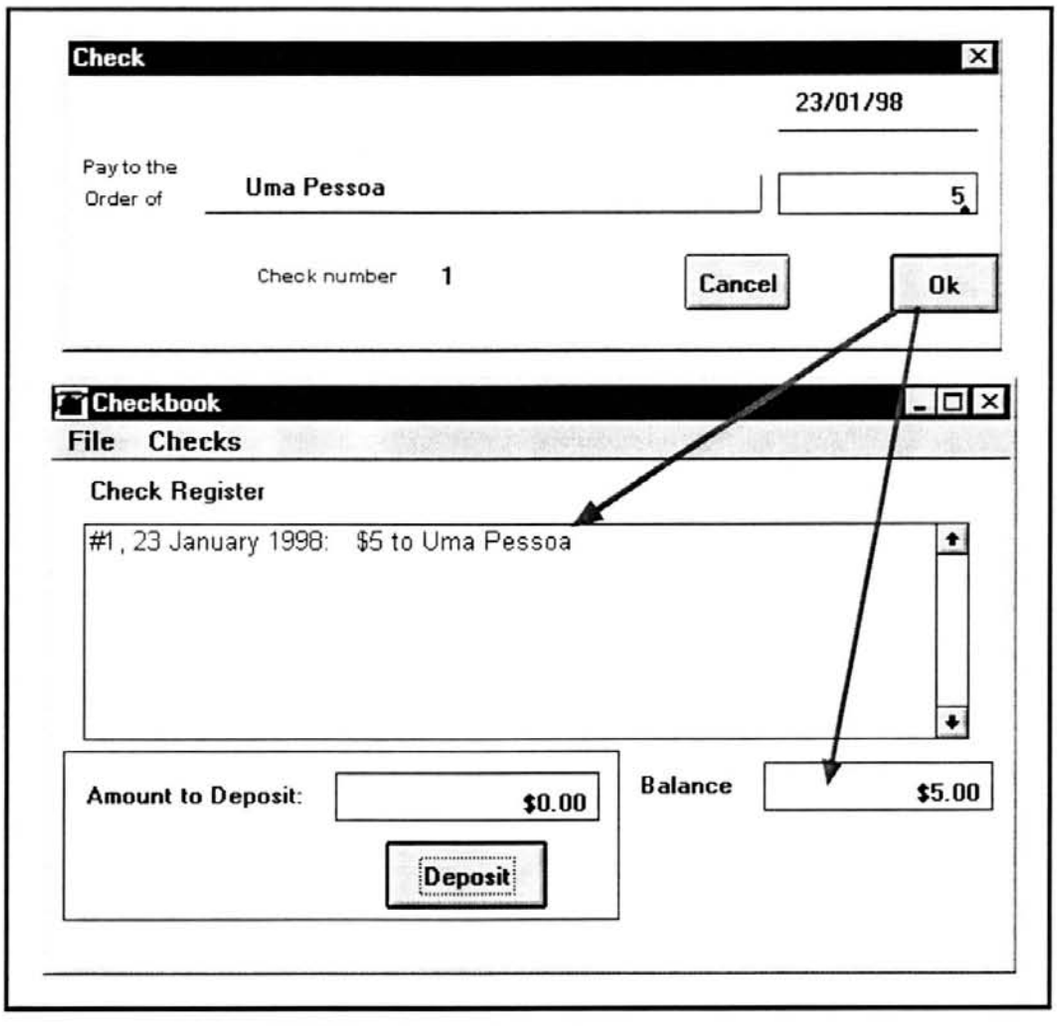

FIGURA 5.15 - Gravação de um cheque

A fig.5.16 mostra o estado dos objetos antes e após a execução dos métodos selecionados na janela "Caminho de Execução dos Métodos Selecionados para Teste". Esta janela fica disponível até que seja solicitado seu fechamento ou o término da execução de ***ATeste***. Esta janela possui o botão de ação "Caminho de Execução da Aplicação", cuja funcionalidade é apresentar a segunda janela de monitoração ao usuário.

A janela "Caminho de Execução da Aplicação" (fig.5.17) apresenta o estado dos objetos no momento de interceptação de qualquer mensagem enviada a uma classe selecionada para monitoração, permanecendo na tela até que o botão de "Ok" seja pressionado.

Cada método monitorado gera um bloco de informações, com duas linhas constantes (ocorrem para todos os métodos) acrescidas de um número variável de linhas (depende do volume de variáveis de instância dos objetos). Estas informações são discriminadas a seguir:

- Primeira linha:
  A primeira linha de cada bloco identifica se as informações referem-se ao estado do objeto antes ou após a execução do método ativado. É apresentada uma mensagem

informando que o estado do objeto que está sendo apresentado foi capturado antes ou após a execução deste método.

- Segunda linha:
  Esta linha contém as informações referentes à mensagem interceptada, descritas a seguir:
  1. Número seqüencial que informa a ordem de ativação dos métodos das classes selecionadas. Esta numeração é seqüencial, e geral para todas as classes refletidas, sendo incrementada a cada mensagem recebida por objetos pertencentes a estas classes. Este número é o mesmo nas duas janelas, servindo como elo de ligação para a localização de um método nestas janelas;
  2. Nome do método ativado pela mensagem interceptada;
  3. Nome da classe receptora da mensagem;
  4. Nome da classe que enviou a mensagem;
  5. Nome da classe que implementa o método ativado.
- Próximas linhas do bloco:
  As próximas linhas do bloco apresentam os subestados do objeto, ou seja, o nome e o valor de cada variável de instância do objeto naquele momento.

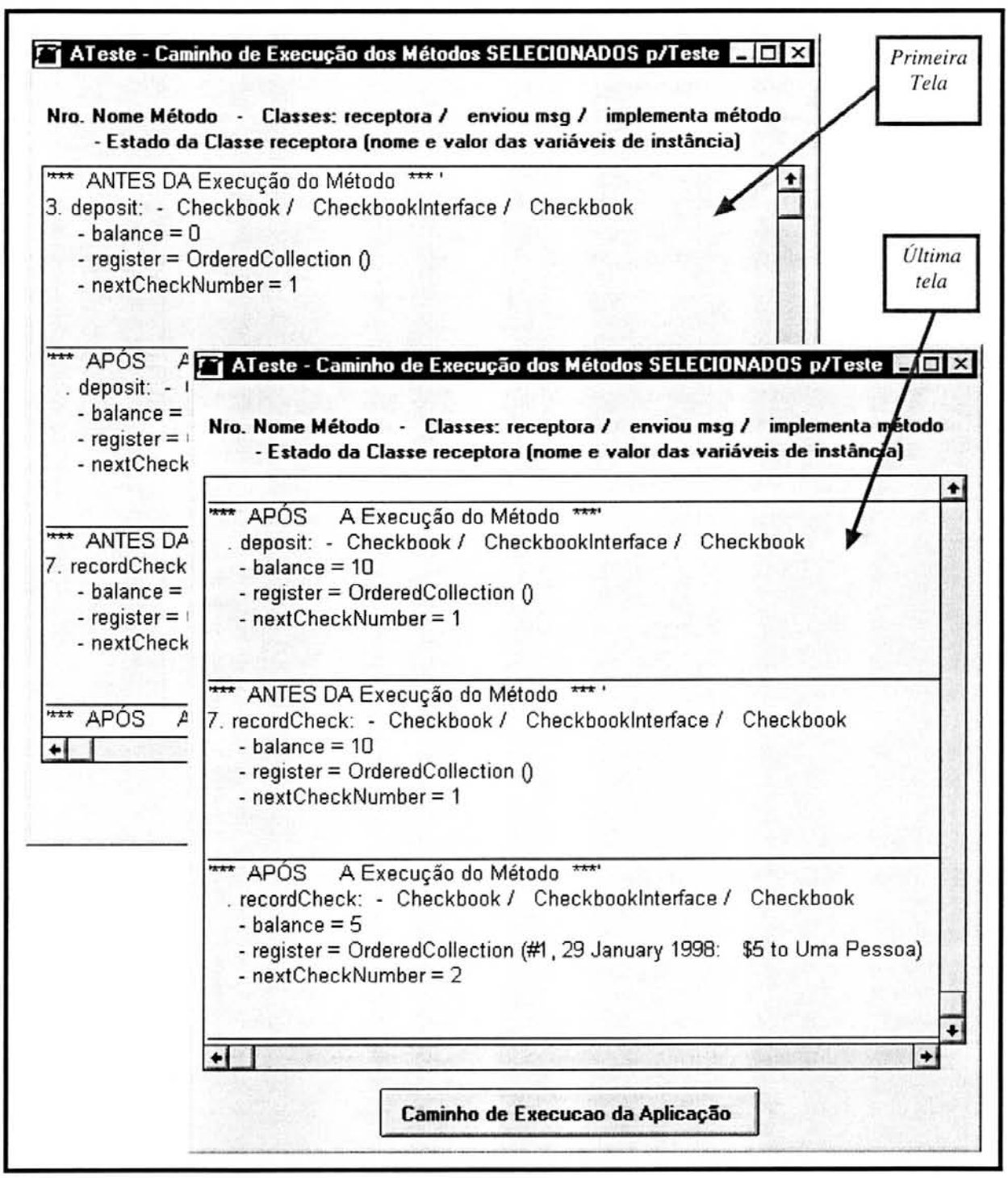

FIGURA 5.16 - Caminho de Execução dos Métodos Selecionados para Teste

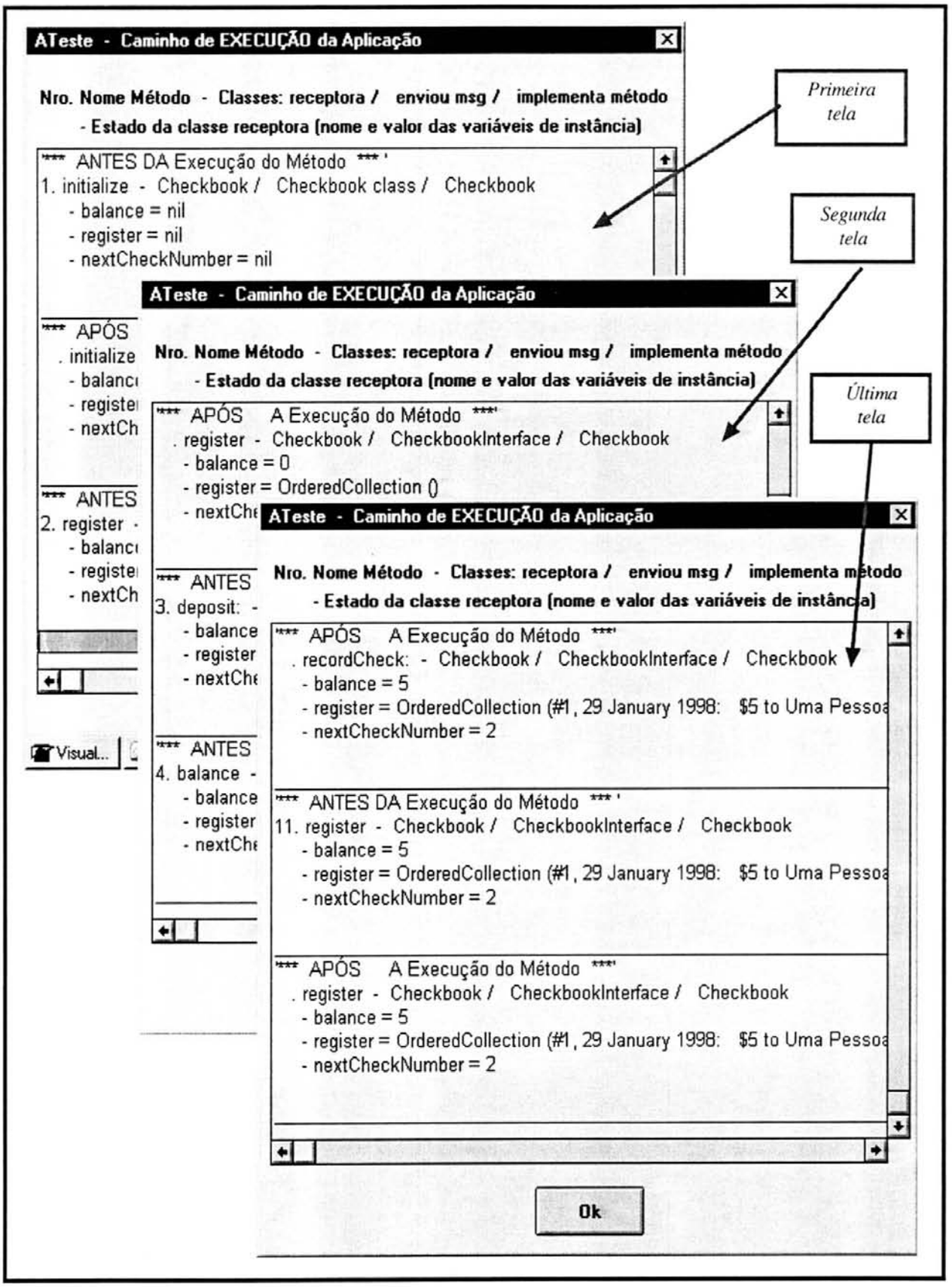

FIGURA 5.17 - Caminho de Execução da Aplicação

6 Conclusões

O estado da arte e a prática do teste de softwares orientados a objetos tem melhorado significativamente nos últimos anos. O teste de software tornou-se um tema de grande importância, com a necessidade da aplicação de métodos que assegurem a qualidade dos produtos finais, a fim de torná-los confiáveis e de fácil manutenção.

Tornando-se eficaz, o teste automatizado vem em auxílio para realizar a promessa da tecnologia orientada a objetos, aumentando o nível de qualidade dos softwares desenvolvidos, e portanto a confiabilidade, diminuindo o esforço de manutenção e propiciando maior reutilização de código. A automação do teste e o reuso, juntos, resultam na diminuição de tempo e custo nas tarefas de desenvolvimento e teste. A curto prazo, o trabalho de desenvolvimento de processos de teste orientado a objetos pode ansiar por avanços na automação da tarefa de teste, mas necessitará conviver com as limitações de uma disciplina de engenharia emergente [BIN95a].

Estratégias de teste OO ainda são imaturas e a atividade de teste ainda se encontra em fase de muitas pesquisas sendo desenvolvidas, e propostas ainda não validadas completamente.

Por estas razões, optou-se pelo desenvolvimento de um projeto embasado na proposta de teste baseado em estados, que apresenta mais informações disponíveis na literatura e fornece uma visão conjunta para o teste de unidade e integração.

Neste trabalho apresentou-se uma abordagem para a construção de uma ferramenta de apoio ao teste de aplicações OO, cujo enfoque central é mostrar as possibilidades de auxílio ao teste, considerando-se os conceitos de teste baseado em estados e utilizando-se uma infra-estrutura reflexiva. Acredita-se que esta combinação gera muitas facilidades para o teste de programas OO.

ATeste tem, como principal objetivo, o auxílio à busca da qualidade de softwares OO, apoiando as atividades de teste e validação de aplicações Smalltalk. Com o suporte provido por esta ferramenta, a análise de aplicações Smalltalk, para averiguação de possíveis erros ao longo do processo de execução, vem ao encontro da exigência de softwares confiáveis.

Nas próximas seções apresenta-se as principais contribuições desta dissertação, possíveis extensões e algumas considerações finais a respeito da mesma.

6.1 Contribuições

Nos capítulos precedentes foi apresentado um estudo sobre os principais aspectos do paradigma OO, teste OO, e mecanismos de reflexão computacional, objetivando fundamentar a proposta apresentada. Foi também realizado um levantamento de diversas propostas/técnicas para teste de programas OO que viessem a contribuir no desenvolvimento desta proposta.

O framework *Luthier MOPs*, recentemente desenvolvido no Instituto de Informática como objeto de uma Tese de Doutorado, também foi alvo de estudos e testes para verificar sua viabilidade como base da estrutura de desenvolvimento do protótipo proposto.

Esta dissertação apresenta como principal contribuição a construção de um protótipo de ferramenta de apoio à atividade de teste de aplicações OO, fornecendo uma infra-estrutura para capturar informações, realizar verificações, e monitorar a execução das aplicações, de forma dinâmica, sem necessidade de alteração do código fonte. Isto é possível graças à composição de um conjunto de meta-objetos coordenados por um gerenciador, que monitoram a execução da aplicação, dinamicamente, realizando a análise e visualização dos resultados sem interferir no seu código.

Os meta-objetos chamados pelo gerenciador no momento em que a mensagem que ativa o método refletido é enviada à classe sob teste, interceptam a execução do método, possibilitando a captura das informações acerca do comportamento do objeto (no caso, o estado atual do objeto) em tempo de execução. Através da visualização dos valores dos atributos do objeto (seu estado atual) antes e após a execução do método interceptado, e a sua comparação com os estados de entrada e saída esperados, tem-se a possibilidade de informar ao testador o erro no momento de sua ocorrência.

O mecanismo utilizado pela maior parte das ferramentas atuais para capturar informações para análise de programas de forma dinâmica é a instrumentação do código fonte. O mecanismo de reflexão computacional utilizado por ***ATeste*** possibilita que interceptações de métodos sejam estabelecidas e suprimidas de forma dinâmica, sem afetar o funcionamento próprio das classes interceptadas, eliminando assim a possibilidade de imiscuir-se erros nas aplicações durante o processo de teste.

A implementação da estratégia de *teste dinâmico de caminho*, aplicando conceitos de teste baseado em estados, fornece ao testador uma visão do estado da classe sob teste, antes e após a execução dos métodos requeridos para teste, bem como uma visão textual de todo o caminho executado pela aplicação, mostrando todos os métodos executados e os respectivos estados, de maneira ordenada.

6.2 Extensões Futuras

Este trabalho apresenta um vasto horizonte de possíveis extensões para prover um apoio mais amplo à automatização da tarefa de teste de softwares OO. A seguir são descritas as principais extensões e pesquisas que podem ser realizadas, visualizadas a partir do desenvolvimento de ***ATeste***.

Sem um conjunto de casos de teste para controle dos resultados, que proverá uma base para comparar os resultados do teste, qualquer atividade de teste é inútil. Uma lista de casos de teste e suas saídas esperadas deve ser preparado, manualmente ou de forma automatizada.

A geração automatizada de dados para teste representa uma extensão necessária, mas que não é de fácil aplicação nas linguagens OO, principalmente devido à ligação dinâmica de mensagens, polimorfismo e herança.

Para implementar-se esta funcionalidade, é necessário que se conheça o fluxo de controle de toda a classe (incluindo todos os seus métodos) e o fluxo de dados dos atributos da classe. Estão sendo analisadas as técnicas de análise estrutural, proposta por [BIN96], para geração da árvore de transições com o objetivo de verificar a viabilidade de sua implementação, a fim de que ***ATeste*** possa auxiliar o desenvolvedor na geração de pré e pós condições, automatizando a tarefa de geração de casos de teste.

O protótipo pode também ser estendido, aproveitando-se sua característica incremental, para a adição de novas estratégias de teste que venham a complementar a estratégia já implementada.

A técnica de teste incremental hierárquico de classes proposta em [HAR92], por exemplo, tem sido objeto de estudo para sua adaptação afim de ser utilizada em conjunto com a estratégia presentemente proposta, utilizando a herança para a geração de uma história de teste, que contribuiria na geração de casos de teste.

O teste incremental de estruturas de classe explora a natureza hierárquica da relação de herança para testar grupos de classes relacionadas através do reuso das informações de teste da classe pai. Para se testar uma subclasse utilizar-se-ia a história de teste da classe pai somado à definição de um modificador e um mapeamento de herança da classe.

O principal benefício desta técnica é evitar o teste completo de cada subclasse, pois a história de teste da superclasse é reutilizada para projetar os casos de teste da subclasse. Somente os atributos novos ou redefinidos na subclasse são retestados. Desta forma, há uma diminuição do tempo despendido no teste e, sendo uma técnica automatizada, limita a intervenção do usuário no processo de teste

Outra extensão possível, seria a transformação de ***ATeste***, de uma ferramenta de apoio ao teste de aplicações Smalltalk, para um ambiente visual de teste. Através de visualizações gráficas da dinâmica das aplicações, poderia disponibilizar-se a navegação e exploração destas visualizações, bem como a seleção de pontos específicos para verificação e análise.

A portabilidade de ***ATeste*** para implementação em outras linguagens OO é dependente da existência de facilidades reflexivas nas linguagens hospedeiras, mas pode ser estudado o uso de mecanismos específicos próprios destas linguagens para sua viabilização.

6.3 Considerações Finais

Na implementação de códigos OO, o teste de software possui grande importância para a construção de sistemas de alta qualidade. Nos sistemas convencionais as verificações estáticas podem ser muito mais efetivas e eficientes na remoção de falhas que o teste. Entretanto, o código fonte OO pode ser consideravelmente mais difícil de entender que o código fonte convencional bem estruturado, ao menos por três motivos:

- a fragmentação das funcionalidades, causada pela herança,
- as ligações dinâmicas, e
- as estratégias de controle cooperativas, onde o fluxo de controle e o controle de estados estão distribuídos em várias classes.

Com tais obstáculos para seu entendimento, as técnicas estáticas são menos efetivas, necessitando de mais testes para obter o mesmo nível de qualidade [BIN95].

Entretanto, para iniciar-se o processo de análise dinâmica é necessária a recuperação de informações estáticas, ou seja, conhecimento prévio da aplicação a ser testada. Tal limitação é inerente a qualquer ferramenta de análise, e no caso de ***ATeste***, a análise estática é necessária para a determinação das classes que devem ser refletidas, os métodos a serem testados e seus possíveis estados de entrada e saída.

ATeste apresenta a aplicação de programação reflexiva combinada aos conceitos de teste baseado em estados. O resultado desta combinação foi uma proposta voltada ao suporte do teste dinâmico de aplicações orientadas a objetos para a monitoração destas aplicações.

Uma característica notável em ***ATeste*** é proporcionar um ambiente de teste adaptável, onde o comportamento das aplicações sob teste pode ser alterado, e novos testes realizados, de forma dinâmica.

Bibliografia

[AMA97] AMANDI, A. A. **Programação de Agentes Orientada a Objetos**. Porto Alegre: CPGCC da UFRGS, 1997. Tese de Doutorado.

[BAL96] BALLISTA, A. L. C. **MAROLA: Um Framework para sonorização de Aplicações Orientadas a Objetos**. Porto Alegre: CPGCC da UFRGS, 1996. Dissertação de Mestrado.

[BIN95] BINDER, R. V. **Testing Object-Oriented Systems: A Status Report**. Disponível por WWW em http://www.rbsc.com/pages/site_map.html (dez.1995).

[BIN95a] BINDER, R. V. Trends in Testing Object-Oriented Software. **Computer**, New York, v.28, n.10, p.68-69, Oct.1995.

[BIN95b] BINDER R. V. State-Based Testing. **Object**, New York, v.5, n.6, p.75-78, July/Aug.1995.

[BIN96] BINDER, R. V. **Overview of the FREE Aproach to Testing Object-Oriented Systems**. Disponível por WWW em http://www.rbsc.com/pages/FREE.html (jun.1996).

[CAM96] CAMPO, M. R.; PRICE, R. T. Um Framework Reflexivo para Ferramentas de Visualização de Software. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 10., 1995, São Carlos. **Anais...** São Carlos: SBC, 1996. p.153-169.

[CAM97] CAMPO, M. R. **Compreensão Visual de Frameworks através da Introspecção de Exemplos**. Porto Alegre: CPGCC da UFRGS, 1997. Tese de Doutorado.

[CAS94] CASTRO, J. F. B. **Análise e Projeto Orientado a Objeto**. Caxambu/MG: SBC, 1994. 53f. Trabalho apresentado na Jornada de Atualização em Informática, 13., 1994, Caxambu, MG.

[CHA97] CHAIM, M. L.; MALDONADO, J. C.; JINO, M. Ferramentas para Teste Estrutural de Software baseado em Análise de Fluxo de dados: o caso POKE-TOOL. In: WORKSHOP DO PROJETO, VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia, SP. **Anais...** Águas de Lindóia/SP: ICMS/USP, 1997. p.28-30.

[COA92] COAD, P.; YOURDON, E. **Análise Baseada em Objetos**. Rio de Janeiro: Campus, 1992.

[COA93] COAD, P.; YOURDON, E. **Projeto Baseado em Objetos**. Rio de Janeiro: Campus, 1993.

[DEL93] DELAMARO, M. E. **Proteum - Um Ambiente de Teste Baseado na Análise de Mutantes**. São Carlos: ICMSC da USP, 1993. Dissertação de Mestrado.

Engineering Department of University of Texas at Arlington. Arlington. Nov.1991. Disponível por WWW em http://www.cse.uta.edu/index.html.

[KUN95] KUNG, D, et al. Developing an Object-Oriented Software Testing and Maintenance Environment. **Communications of ACM**, New York, v.38, n.10, p.75-87, Oct.1995.

[DEL97] DELAMARO, M. E.; MALDONADO, J. C.; NAKAGAWAU, E. Y. PROTEUM/IM: Uma Ferramenta de Apoio ao Teste de Integração. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 11., 1997, Fortaleza. **Anais...** Fortaleza: UFC, 1997. p.487-490.

[DEL97a] DELAMARO, M. E.; MALDONADO, J. C. Teste de Integração: Projeto de Operadores para o Critério Mutação de Interface. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 11., 1997, Fortaleza. **Anais...** Fortaleza: UFC, 1997. p.413-428.

[DEM87] DeMILLO, R. A. Et al. **Software Testing and Evaluation**. Menlo Park:The Benjamin/Cummings, 1987.

[DOO94] DOONG, R.; FRANKL, P.G. The ASTOOT Approach to Testing Object-Oriented Programs. **ACM Transations on Software Engineering and Methodology**, New York, v.3, n.2, p.101-130, Apr.1994.

[DRA96] DRAKE, T. Measuring Software Quality: A Case Study. **Computer**, New York, v.23, n.11, p.78-87, Nov.1996.

[FAY95] FAYAD, M. E.; TSAI, W. Object-Oriented Experiences. **Communications of ACM**, New York, v.38, n.10, p.51-53, Oct.1995.

[FER89] FERBER, J. Computational Reflection in Class based Object Oriented Languages. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, 1989. **Proceedings...** New York: ACM Press, 1989. p.317-326.

[FIE89] FIEDLER, S.P. Object-Oriented Unit Testing. **Hewlett-Packard Journal**, Palo Alto, v.40, n.2, p.69-75, Apr.1989.

[HAL93] HALLADAY, S.; WIEBEL, M. **Object-Oriented Software Engineering**. Lawrence: R&D Publications Inc, 1993. p.189-243.

[HAR92] HARROLD, M. J.; MCGREGOR, J. D.; FITZPATRICK, K. J. Incremental Testing of Object-Oriented Class Structures. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 14., 1992, Melbourne, Austrália. **Proceedings...** Los Alamitos: IEEE, 1992. p.68-80.

[HET87] HETZEL, W. **Guia Completo ao Teste de Software**. Rio de Janeiro: Campus, 1987.

[KUN91] KUNG, C. **The Object-Oriented Paradigm**. Computer Science Engineering Department of University of Texas at Arlington. Arlington. Nov.1991. Disponível por WWW em http://www.cse.uta.edu/index.html.

[KUN95] KUNG, D, et al. Developing an Object-Oriented Software Testing and Maintenance Environment. **Communications of ACM**, New York, v.38, n.10, p.75-87, Oct.1995.

[LIS93] LISBOA. P. H. C.; TEPETINO, J. F.; MEIRA, S. L. Reflexão Computacional em SMALLTALK. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 13., 1993, Florianópolis. **Anais...** Florianópolis: SBC, 1993. v.2, p.382-396.

[LIS95] LISBOA. M. L. B. **MOTF: Meta-Objetos para Tolerância a Falhas**. Porto Alegre: CPGCC da UFRGS, 1995. Tese de Doutorado.

[MAE87] MAES, P. Concepts and Experiments in Computational Reflection. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS CONFERENCE, 1987. **Proceedings...** New York: ACM Press, 1987. p.147-155.

[McG94] McGREGOR, J. D.; KORSON, T. Integrated Object-Oriented and Development Processes. **Communications of the ACM**, New York, v.37, n.9, p.59-77, Sept.1994.

[McG96] McGREGOR, J. D. Testing Object-Oriented Components. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1996, Linz. **Tutorial Notes**. Berlin:Springer-Verlag, 1996.

[MYE79] MYERS, G. T. **The Art of Software Testing**. New York: John Wiley & Sons, 1979.

[PRE95] PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books do Brasil, 1995.

[PRI94] PRICE, A. M. de A.; SILVA, J. B. PROTESTE+: Ambiente de Validação Automática de Qualidade de Software através de Técnicas de Teste e de Métricas de Complexidade. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE - WORKSHOP DE QUALIDADE DE SOFTWARE, 8., 1994, Curitiba. **Anais...** Curitiba: Centro Internacional de Tecnologia de Software, 1994. v.3, p.31-33.

[RUM97] RUMBAUGH, J. et al. **Modelagem e Projetos baseados em Objetos**. Rio de Janeiro: Campus, 1997.

[SIC97] SICHONANY, O. R. de A. **Métricas de Complexidade de Sistemas Orientados a Objetos**. Porto Alegre: CPGCC da UFRGS, 1997. Dissertação de Mestrado.

[SIL95] SILVA, J. B. **PROTESTE+: Ambiente de Validação Automática de Validação de Software através de Técnicas de Teste e de Métricas de Complexidade**. Porto Alegre: CPGCC da UFRGS, 1995. Dissertação de Mestrado.

[SMI92] SMITH, M. D.; ROBSON, D. J. A Framework for Testing Object-Oriented Programs. **Journal of Object-Oriented Programming**, [S.l.], v.5, n.3, p.45-53, June 1992.

[TUR93] TURNER, C. D.; ROBSON, D. J. **The Testing of Object-Oriented Programs**. England: University of Durham, Computer Science Division /School of Engineering and Computer Science (SECS), Feb.1993. (Technical Report TR-13/92).

[TUR93a] TURNER, C. D.; ROBSON, D. J. **State-Based Testing and Inheritance**. England: University of Durham, Computer Science Division/School of Engineering and Computer Science (SECS), Apr.1993. (Technical Report TR-1/93).

[TUR93b] TURNER, C. D.; ROBSON, D.J. **A Suite of Tools for the State-Based Testing of Object-Oriented Programs**. England: University of Durham, Computer Science Division/School of Engineering and Computer Science (SECS), Apr.1993. (Technical Report TR-14/92).

[VER92] VERILOG. **LOGISCOPE Technical Representation**. [S.l.]: VERILOG, 1992. 19p.

[VIE97] VIEIRA, M. E.R.; TRAVASSOS, G. H. **Testes em Sistemas Orientados a Objetos: Sua Aplicação e Integração a um Ambiente de Desenvolvimento de Software**. Rio de Janeiro: COPPE/UFRJ, 1997. (Relatório Técnico ES-422/97).

[WIL92] WILDE, N.; HUITT, R. Maintenance Support for Object-Oriented Programs. **IEEE Transactions on Software Engineering**, New York, v.18, n.12, p.1038-1044, Dec.1992.

Coordenadoria do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS

Informática **UFRGS**

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

"Um Sistema de Apoio ao Teste de Aplicações Smalltalk

por

Ivete Martins Pinto

Dissertação apresentada aos Senhores:

Prof. Dr. Carlos Alberto Heuser

Prof. Dr. José Carlos Maldonado (USP)

Profa. Dra. Maria Lúcia Blanck Lisbôa

Vista e permitida a impressão.
Porto Alegre, 14/05/98.

Profa. Dra. Ana Maria de Alencar Price,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora do Curso de Pós-Graduação
em Ciência da Computação
Instituto de Informática - UFRGS