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

Ferramenta de Apoio ao Teste de Aplicações Java Baseada em Reflexão Computacional

por
FÁBIO FAGUNDES SILVEIRA

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

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

Porto Alegre, dezembro de 2001

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Silveira, Fábio Fagundes

Ferramenta de Apoio ao Teste de Aplicações Java Baseada em Reflexão Computacional / por Fábio Fagundes Silveira. – Porto Alegre: PPGC da UFRGS, 2001.

96 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientadora: Price, Ana Maria de Alencar.

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

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profª. Wrana Panizzi

Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann

Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fensterseifer

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do PPGC: Prof. Carlos Alberto Heuser

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Agradecimentos, ao meu ver, constituem uma pequena, mas sincera, forma de homenagem a pessoas que tantas coisas fizeram e fazem por mim.

Algumas delas, com contribuições valiosas que nortearam o rumo deste trabalho, enquanto que, outras, me apoiam desde meus primeiros momentos de vida.

Em primeiro lugar, gostaria de agradecer à minha família,
pela criação, demonstração de eterno amor, união e cumplicidade.

Um agradecimento especial a meus pais,
pessoas que, com o uso de palavras, seria difícil homenageá-los, pois aprendi com eles, que isto se faz com gestos e atitudes.
Agradeço a eles,
pela formação, apoio em todos os sentidos e, constantes incentivos.

A minha orientadora e amiga, Professora Dra. Ana Maria de Alencar Price,
uma pessoa incentivadora e sempre disposta a ajudar, com sugestões e críticas técnicas extremamente valiosas, que enriquecem a base teórica e prática deste trabalho, sempre expostas de uma maneira muito meiga, a qual lhe é peculiar.
Professora Ana, muito obrigado!

A minha namorada Viviane,
pelo constante apoio e pela paciência a mim dispensada, nas inúmeras vezes que deixamos de estar juntos devido ao trabalho que por mim esperava.

Ao meu amigo Rodrigo Senra,
que conheci no desenvolvimento deste trabalho, e cujo auxílio foi muito importante para a conclusão do mesmo.

Ao Professor e amigo Carlos Abot Gomes,
pela dedicação e boa vontade na revisão da Língua Portuguesa em todos os trabalhos desenvolvidos neste curso.

Ao meu tio, Edval Rosa Fagundes “in memorian”,
cuja vontade de, na vida vencer, sempre foi e será para mim, um referencial de perseverança, cultura e conhecimento.

A todos que, de uma forma ou de outra, contribuíram para a realização deste trabalho,
meus sinceros agradecimentos!

Sumário

Lista de Abreviaturas ... **6**
Lista de Figuras ... **7**
Lista de Tabelas ... **9**
Resumo ... **10**
Abstract ... **11**
1 Introdução ... **12**
1.1 Estrutura do documento ... **13**
2 Teste de *Software* ... **15**
2.1 Teste de *Software* Orientado a Objeto ... **17**
2.2 Teste de *Software* Orientado a Objeto Baseado em Estados ... **19**
2.2.1 Diagrama de Estados ... 20
2.3 O uso de Asserções no Teste de Estados ... **25**
3 Reflexão Computacional ... **28**
3.1 Arquitetura Reflexiva ... **28**
3.2 Modelos de Reflexão ... **29**
3.3 Estilos de Reflexão ... **31**
3.4 Linguagens Reflexivas ... **32**
3.5 Reflexão na linguagem de programação Java ... **33**
3.6 Protocolo Reflexivo CoreJava ... **35**
3.6.1 Obtenção de informações das Classes ... 36
3.7 Protocolo Reflexivo MetaJava ... **38**
3.7.1 Arquitetura do Protocolo ... 40
3.8 Protocolo Reflexivo OpenJava ... **42**
3.8.1 Processo de Tradução ... 43
3.8.2 A API do Protocolo ... 45
3.9 Protocolo Reflexivo Guaraná ... **47**
3.9.1 Meta-Objetos do Protocolo ... 47
3.9.2 Composers ... 48
3.9.3 Estrutura do MOP ... 49
3.9.4 Dinâmica do MOP ... 50
3.9.5 Gerenciamento de Meta-Configuração ... 54
3.9.6 Bibliotecas de Meta-Objetos ... 55
3.9.7 Principais Classes e Métodos do Protocolo Guaraná ... 55
3.10 Comparação entre os protocolos de reflexão ... **58**
3.10.1 Análise dos Protocolos de Reflexão ... 58
4 A Ferramenta *KTest* ... **61**
4.1 Características ... **61**
4.1.1 Verificação de Asserções ... 63
4.2 Implementação da *KTest* ... **64**
4.2.1 Classes desenvolvidas ... 64

4.2.1.1 Classe ClassData 65
4.2.1.2 Classe *ClasseS* 66
4.2.1.3 Classe Construtor 66
4.2.1.4 Classe Atributo 66
4.2.1.5 Classe AtributoS 67
4.2.1.6 Classe *VlrAP* 68
4.2.1.7 Classe *Metodo* 68
4.2.1.8 Classe *MetodoS* 69
4.2.1.9 Classe *KTest* 69
4.2.1.10 Classe *KMeta* 70
4.2.1.11 Classe *KUtil* 72
4.2.1.12 Classe diaEspInvClass 72
4.2.1.13 Classe diaEspPrePos 73
4.2.1.14 Classe CheckAssert 73
4.2.1.15 Classe diaResultAssert 74
4.2.2 Problema Crítico: Recursão Infinita no Meta-nível 75
4.2.3 HashTables: ganho de desempenho em meta-interações 76
4.2.4 Validação e teste das asserções 77
4.3 Funcionamento da Ferramenta 78
4.3.1 Execução da Aplicação: ativação da reflexão 83
5 Conclusões 89
5.1 Principais contribuições 89
5.2 Extensões Futuras 90
Bibliografia 91

Lista de Abreviaturas

API	*Application Programming Interface*
AWT	*Abstract Window Toolkit*
DLL	*Dynamic Linked Library*
fig.	*figura*
JDK	*Java Development Kit*
JEP	*Java Expression Parser*
JRE	*Java Run Time Environment*
JVM	*Java Virtual Machine*
MLI	*Meta-Level Interface*
MOP	*Meta-Object Protocol*

Lista de Figuras

FIGURA 2.1 – Representação da Classe *Quarto* 22
FIGURA 2.2 - Máquina de Estados Finitos da Classe *Quarto* 23
FIGURA 2.3 - Árvore de Transição da Classe *Quarto* 24
FIGURA 3.1 - Arquitetura Reflexiva 29
FIGURA 3.2 - Hierarquia de Classes 30
FIGURA 3.3 - Reflexão de Meta-Classe da classe *Terrestre* 30
FIGURA 3.4 - Reflexão de Meta-Objetos (adaptado de [BER 99]) 31
FIGURA 3.5 - Reflexão Comportamental [PIN 98] 31
FIGURA 3.6 - Obtenção do nome da classe 34
FIGURA 3.7 – Programa *Mosta_Info* 37
FIGURA 3.8 – Resultados da execução do programa *Mosta_Info* 37
FIGURA 3.9 – Modelo computacional de reflexão comportamental [KLE 96] 38
FIGURA 3.10 – Exemplo de ligação entre o nível-base e o meta-nível [GOL 97] 40
FIGURA 3.11 – Arquitetura do Protocolo MetaJava [GOL 97] 40
FIGURA 3.12 – Fluxo de informações do compilador OpenJava [TAT 99] 42
FIGURA 3.13 – Reimplementação de métodos [CHI 99] 43
FIGURA 3.14 – Exemplo de programa de nível-base – *Hello.oj* - [TAT 99b] 43
FIGURA 3.15 – Programa de nível-base – *Sequencia.oj* 44
FIGURA 3.16 – Programa de meta-nível – *MetaSequencia.oj* 44
FIGURA 3.17 – Programa de nível-base traduzido – *Sequencia.java* 45
FIGURA 3.18 – Diagrama de Classe do MOP [SEN 2001] 49
FIGURA 3.19 – Interação entre dois objetos do para-nível [OLI 98d] 51
FIGURA 3.20 – Interação com um objeto reflexivo [OLI 98d] 52
FIGURA 3.21 – Meta-configuração [OLI 98] 55
FIGURA 3.22 – Exemplo de Reconfiguração [OLI 98] 56
FIGURA 4.1 – Diagrama de Funcionamento da *KTest* 62
FIGURA 4.2 –Classes que armazenam informações sobre a aplicação em teste 64
FIGURA 4.3 – Classe *ClassData* 65
FIGURA 4.4 – Classe *ClasseS* 66
FIGURA 4.5 – Classe *Construtor* 66
FIGURA 4.6 – Classe *Atributo* 67
FIGURA 4.7 – Classe *AtributoS* 67
FIGURA 4.8 – Classe *VlrAP* 68
FIGURA 4.9 – Classe *Metodo* 68
FIGURA 4.10 – Classe *MetodoS* 69

FIGURA 4.11 – Classe *KTest* 69
FIGURA 4.12 – *KMeta* - Classe de Meta-Nível da *KTest* 71
FIGURA 4.13 – Classe *KMeta* 71
FIGURA 4.14 – Classe *KUtil* 72
FIGURA 4.15 – Classe *disEspInvClass* 72
FIGURA 4.16 – Classe *disEspPrePos* 73
FIGURA 4.17 – Classe *CheckAssert* 74
FIGURA 4.18 – Classe *disResultAssert* 74
FIGURA 4.19 – Utilizando *HashTables* 77
FIGURA 4.20 – Utilizando *HashTables* 77
FIGURA 4.21 – Exemplo da utilização do pacote *JEP* 77
FIGURA 4.22 – Diagrama de Classes – *Contas Bancárias* 79
FIGURA 4.23 – *KTest* – classes e métodos escolhidos para monitoramento 80
FIGURA 4.24 – *KTest* – janela para especificação de invariantes 81
FIGURA 4.25 – *KTest* – Pré e pós-condições do método *credita* 82
FIGURA 4.26 – *KTest* – Pré e pós-condições do método *debita* 82
FIGURA 4.27 – Aplicação de Contas Bancárias 83
FIGURA 4.28 – Abertura de uma Conta Especial com Bônus 84
FIGURA 4.29 – Consulta à Conta Especial com Bônus após operação “c” 84
FIGURA 4.30 – Consulta à Conta Especial com Bônus após operação “e” 85
FIGURA 4.31 – Violação da Pós-Condição do método *debita* após a operação “h” 85
FIGURA 4.32 – Violação da invariante da classe *ContaBonus* após a operação “h” 86
FIGURA 4.33 – Resultado da Pré-Condição do método *credita* – operação “e” 87
FIGURA 4.34 – Resultado da Pré-Condição do método *debita* – operação “h” 87
FIGURA 4.35 – Histórico dos métodos de *ContaBonus* 88

Lista de Tabelas

TABELA 3.1 – Métodos Reflexivos da JVM padrão 34
TABELA 3.2 – Métodos para vinculação de meta-objetos... 39
TABELA 3.3 – Métodos de Reflexão Comportamental do MetaJava.......................... 41
TABELA 3.4 – Principais métodos da classe *OJClass* 45
TABELA 3.5 – Métodos da Classe Guarana.. 55
TABELA 4.1 – Resumo das principais características dos protocolos de reflexão 60

Resumo

A atividade de teste constitui uma fase de grande importância no processo de desenvolvimento de *software*, tendo como objetivo garantir um alto grau de confiabilidade nos produtos desenvolvidos.

O paradigma da Orientação a Objetos (OO) surgiu com o objetivo de melhorar a qualidade bem como a produtividade no desenvolvimento de aplicações. Entretanto, apesar do aumento constante de aceitação do paradigma OO pela indústria de *software*, a presença de algumas de suas características torna a atividade de teste de programas neste paradigma mais complexa do que o teste de sistemas tradicionais. Entre estas características cita-se a herança, o encapsulamento, o polimorfismo e a ligação dinâmica [EIS 97] [PRE 95] [UNG 97]. Algumas técnicas estão sendo implementadas para auxiliarem a atividade de teste através do uso da tecnologia de reflexão computacional [HER 99]. Estas técnicas permitem a realização de análises de aspectos dinâmicos dos programas, sem a necessidade de instrumentar o código-fonte das aplicações que estão sendo monitoradas.

Com o objetivo de auxiliar o processo de teste de programas orientados a objetos, este trabalho aborda o desenvolvimento de uma ferramenta, a qual automatiza parcialmente o teste de programas escritos em Java. A ferramenta evidencia o teste de estados fazendo uso da tecnologia de reflexão computacional.

Através da especificação de asserções, feitas pelo usuário da ferramenta, na forma de invariantes de classe, pré e pós-condições de métodos, é possível verificar a integridade dos estados dos objetos durante a execução do programa em teste. A ferramenta possibilita também, armazenar a seqüência de métodos chamados pelos objetos da aplicação em teste, tornando possível ao testador, visualizar o histórico das interações entre os objetos criados no nível-base.

Palavras-chaves: Teste de *Software* Orientado a Objetos, Orientação a Objetos, Reflexão Computacional, Protocolos de Reflexão Computacional, Java.

TITLE: "TESTING TOOL FOR JAVA-WRITTEN APPLICATIONS SUPPORTED BY COMPUTATIONAL REFLECTION"

Abstract

The testing activity is one of the most important phases in the software development process, whose goal is to guarantee a high degree of reability to the developed products.

The Object-Oriented Paradigm (OO) comes out with the goal to increase the quality as well as the productivity during applications development. Though such paradigm has been having a constantly acceptation by software developers, some of its characteristics make the testing activity under this paradigm more complex than the testing acitvity under tradicionals paradigms. Among these characteristics, we could list inheritance, encapsulation, polymorphism and dynamic binding [EIS 97] [PRE 95] [UNG 97]. Some techniques has been implemented in order to assist the testing activity through the use of computational reflection [HER 99]. Such techniques allow the analysis of program dynamic aspects, without needing to inspect the applications source-code which are being monitored.

With the aim to help the object-oriented program testing process, this paper focuses on the development of a tool which supports and partially automatizes this phase. The tool is oriented to state-based testing of Java-written programs, supported by the mechanism of computational reflection.

Through the definition of assertions, as class invariants, methods preconditions and posconditions, especified by the tool user, it is possible to verify the state integrity of the objects during the execution of the program being tested. The tool also allows the storage of the methods sequence called by the application objects under testing, giving to the user the possibility of visualizing the history of the interactions that have taken place among the objects that have been instancied into the base-level.

Keywords: Object-Oriented Testing Software, Object-Oriented Software, Computational Reflection, Meta-Object Protocol, Java.

1 Introdução

O teste é uma atividade associada a qualquer processo, cujo objetivo seja produzir um produto [PER 2000], sendo utilizado para determinar a qualidade deste produto durante seu desenvolvimento e, também, após sua construção. Desta forma, o teste é parte integrante do processo e não uma atividade auxiliar durante sua elaboração.

Na busca pela qualidade no desenvolvimento de produtos, com o objetivo de garantir um alto grau de confiabilidade, a atividade de teste tem se caracterizado como de grande importância. O processo de teste de *software* representa um conjunto de atividades realizadas com o objetivo de garantir a maior qualidade possível em produtos de *software*.

De acordo com Pressman [PRE 95], o processo de teste tem ganhado significativa importância, pois pode consumir até 40% do esforço despendido no desenvolvimento de *software*. A atividade de teste é um elemento crítico na garantia de qualidade de um produto de *software* visando torná-lo confiável. Entretanto, é praticamente impossível executar um teste de *software* completo, já que o teste não mostra a ausência de erros, mas apenas defeitos de *software* que estão presentes [MYE 79].

O paradigma da Orientação a Objetos (OO) surgiu com o objetivo de melhorar a qualidade bem como a produtividade no desenvolvimento de *software*, desde que sejam bem exploradas suas características, possibilitando assim, maior reutilização, confiabilidade, modularização e rapidez de desenvolvimento [PRE 95]. Entretanto, apesar do aumento constante de aceitação do paradigma OO pela indústria do *software*, os novos conceitos presentes nesta abordagem introduziram um conjunto de problemas na atividade de teste de programas [KUN 95]. Tais problemas tornam o processo de teste neste paradigma mais complexo do que em sistemas tradicionais, pois esconde informações e dificulta a identificação do código que gerou o erro [EIS 97] [PRE 95] [UNG 97].

Algumas técnicas de teste de *software* OO estão sendo implementadas através do uso da tecnologia de reflexão computacional [HER 99], as quais permitem a realização de análises de aspectos dinâmicos dos programas, de forma a auxiliar o processo de teste e depuração neste paradigma.

A utilização do conceito de reflexão computacional tem atraído cada vez mais a atenção de pesquisadores, sendo vista como uma alternativa de extensão das linguagens de programação. Segundo Maes [MAE 87], reflexão computacional é uma técnica de programação que permite ao programador obter informações a respeito do próprio programa, com o objetivo de monitorá-lo, adicionar novas funcionalidades e mesmo fazer alterações adaptativas em tempo de execução.

Disso resulta que o uso de reflexão é útil em atividades administrativas da aplicação, tais como estatísticas de desempenho, otimização, distribuição, tolerância a falhas e, evidentemente no processo de teste de *software*.

Sob forma de auxiliar no processo de teste de *software* OO, foram analisados diversos protocolos de reflexão computacional disponíveis para a linguagem Java, visando sua aplicação no desenvolvimento de uma ferramenta de teste. Optou-se pelo protocolo reflexivo Guaraná, devido a diversas características benéficas ao teste nele encontrados. Foi então desenvolvida uma ferramenta de teste para aplicações orientadas

a objetos, *KTest*, a qual objetiva fornecer apoio às atividades de teste e validação de aplicações escritas na linguagem Java, dando suporte ao teste de *software* baseado em estados. *KTest* utiliza a tecnologia da Reflexão Computacional para fazer a análise dos estados dos objetos de forma dinâmica, ou seja, durante a execução da aplicação em teste, sem a necessidade de instrumentação do código-fonte da aplicação.

Através da especificação de asserções, introduzidas no *software* em teste pelo usuário (testador) da ferramenta, na forma de invariantes de classe, pré e pós-condições, é possível verificar os estados dos objetos da aplicação em teste. *KTest* possibilita também, armazenar a seqüência de métodos chamados pelos objetos da aplicação em teste, tornando possível ao testador, visualizar o histórico de interações no nível-base

Trabalhos correlatos ao desenvolvido são apresentados por Campo [CAM 97], Pinto [PIN 98b] e Palavro [PAL 2000], implementados na linguagem *Smalltalk*. A ferramenta *ATeste* [PIN 98b] utiliza abordagem reflexiva para a realização do teste de estados, utilizando uma estratégia de teste denominada teste dinâmico de caminho em aplicações desenvolvidas na linguagem *Smalltalk*. *FATOO* [PAL 2000] estende *ATeste*, aceitando pré, pós-condições de métodos e invariantes de classes, gerando diagramas de eventos associados à execução da aplicação e informações relevantes para a aplicação de teste de regressão. Ambas utilizam o *framework LuthierMOPs* [CAM 97], responsável pelo suporte à reflexão computacional, o qual permite monitorar a execução de *frameworks* OO.

1.1 Estrutura do documento

Este documento está organizado conforme descrito abaixo.

No capítulo 2 são introduzidos conceitos referentes ao teste de *software*, e em particular, ao teste de *software* orientado a objetos. É detalhado também neste capítulo o teste de estados, modalidade de teste utilizado pela ferramenta desenvolvida. São apresentados também os conceitos, aplicações e vantagens da utilização das asserções no processo de teste de *software* OO.

O capítulo 3 apresenta os conceitos que envolvem a reflexão computacional, sua arquitetura, modelos e estilos de reflexão. Neste capítulo são apresentados os conceitos referentes a linguagens reflexivas, em especial à linguagem Java. Logo após, são apresentados de forma sucinta os protocolos de reflexão *CoreJava*, *MetaJava* e *OpenJava*. Já o protocolo *Guaraná* é apresentado com um maior grau de detalhamento, incluindo conceitos relativos a sua arquitetura, dinâmica e gerenciamento de meta-configurações, tendo em vista ter sido este o protocolo utilizado na implementação da ferramenta desenvolvida.

Ainda neste capítulo, são também apresentadas considerações sobre a utilização da reflexão computacional no teste de *software*, mostrando também as principais características presentes nos protocolos analisados com relação a sua utilização no teste.

No capítulo 4, é descrita a ferramenta *KTest*, onde são especificadas suas estruturas de classes, suas características e funcionalidades, além de um detalhamento sobre sua implementação. Logo após é explicado o funcionamento da *KTest*, com a submissão de uma aplicação à ferramenta para monitoramento, a fim de avaliar e demonstrar suas principais contribuições ao processo de teste de *software* OO baseado em estados.

Finalmente, no capítulo 5, são apresentadas as conclusões, juntamente com as principais contribuições do trabalho e as futuras extensões da ferramenta desenvolvida. Este capítulo é seguido pela bibliografia utilizada na elaboração deste documento.

2 Teste de *Software*

Muitas são as publicações definindo teste de *software*, entretanto, todas recaem no mesmo princípio. Teste de *software* é o processo de execução de um programa de maneira controlada, com o objetivo de responder à questão: "O *software* se comporta segundo sua especificação?" Segundo Myers [MYE 79], o objetivo principal desta atividade é o de encontrar erros: "A atividade de teste não pode mostrar a ausência de *bugs*; ela só pode mostrar se defeitos de *software* estão presentes".

Uma das razões pela qual o processo de teste tem ganhado significativa importância, é o fato deste consumir até 40% do esforço despendido no desenvolvimento de *software* [PRE 95]. Desta forma, esta atividade constitui um elemento crítico na busca pela garantia de qualidade de um produto de *software*, procurando torná-lo mais confiável.

Ainda sob o ponto de vista econômico, outro fator importante deve ser analisado: os níveis de teste convenientes para uma organização dependerão das conseqüências potenciais de falhas no *software* não detectadas [PRE 95]. Estas, por sua vez, podem variar desde pequenos erros até mesmo situações desastrosas. Uma visão panorâmica por parte da organização é necessária, pois sua credibilidade no mercado pode ser afetada pelo fato da empresa entregar ao mercado *softwares* com erros, resultando disso um impacto negativo em suas perspectivas de negócios. De modo contrário, uma reputação por distribuir *softwares* confiáveis, ajudará certamente na obtenção de negócios futuros.

Segundo Hetzel [HET 91], apenas uma minoria de desenvolvedores praticam a atividade de teste adequadamente, enquanto alguns afirmam que esta é uma atividade irrelevante. Já Siegal [SIE 92] enfatiza que muitas pessoas aceitam a necessidade de teste, sugerindo que esta deve ter custos reduzidos. Todavia, na prática sabe-se que os custos de teste não serão reduzidos.

Além dos aspectos já ressaltados, o fato do incrível aumento no número de usuários de *software*, além do aumento de empresas desenvolvedoras destes produtos enfatiza ainda mais a importância da fase de teste. Então, torna-se mister assegurar que a qualidade e confiabilidade nos produtos de *software* desenvolvidos não constitua um fator de risco e de preocupação.

Conforme descrito em [IPL 96], as atividades geralmente associadas ao processo de teste são: análise estática e análise dinâmica. A primeira investiga o código-fonte do programa, procurando por problemas e reunindo métricas sem efetivamente executar o código. Já a análise dinâmica, tem por objetivo inspecionar o comportamento do programa em teste durante sua execução, provendo com isto, informações de cobertura de teste, como por exemplo, através de execuções passo-a-passo (*trace information)*

O processo de teste não deve ser confundido com o processo de depuração. Por teste, entende-se o processo de analisar e localizar erros quando o *software* não apresenta o comportamento de acordo com o esperado. Depuração é, entretanto, uma atividade que suporta o teste, não podendo, todavia, substituí-lo.

A atividade de teste não termina após finalizada suas conclusões. O *software* deve ser monitorado com o intuito de constatar novos problemas durante seu uso, adaptando-o a novas exigências [IPL 96]. Ressalta-se ainda, que o teste de *software* deve ser repetido, modificado e estendido. Os esforços para revisar e repetir o teste,

constituem o maior custo no processo de construção de um produto de *software*. Todo produto deve ser testado em vários estágios do desenvolvimento, sendo, no entanto, o teste empregado com diversos graus de rigorosidade.

Uma das razões para a grande distância existente entre a prática e teoria com relação à aplicação de técnicas de teste de *software*, segundo Pinto [PIN 96], é o esforço extra, necessário para a realização desta atividade, pois requer que dados e resultados de teste sejam determinados, passando-se então, à execução do sistema com estes dados, analisando ao final os resultados obtidos.

Basicamente, existem três abordagens de teste:

- Teste Estrutural: também conhecido como teste da "*caixa-branca*", derivado da estrutura interna do programa, cuja estratégia concentra-se na tentativa de exercitar todo o código da aplicação, com base no fluxo de controle e/ou no fluxo de dados [MAL 98] [PRE 95]. Entre os problemas desta abordagem, cita-se:

    → Sendo a análise da estrutura interna realizada de maneira estática, programas com laços resultam numa quantidade infinita de caminhos. Segundo Myers [MYE 79], a solução seria a aplicação do teste exaustivo, no entanto, este é impraticável;

    → Desperdícios de tempo e financeiros, na geração de casos de teste para caminhos não executáveis (*infeasible paths*) [VER 97];

    → O fato da execução de um caminho resultar em sucesso, não garante a corretеza do mesmo, pois outros casos de teste podem invalidar o mesmo.

Com o objetivo de minimizar os problemas supra citados, são utilizados critérios de seleção de caminhos, divididos em dois grupos: fluxo de controle e fluxo de dados.

- Teste Funcional: também conhecido como teste da "*caixa-preta*", preocupa-se com a verificação do atendimento das funcionalidades requeridas, baseando-se na especificação dos requisitos do sistema. Assim, não considera a estrutura interna do programa, sendo este teste aplicado quando o sistema já encontra-se em fase final ou completamente construído. De acordo com Pressman [PRE 95], os erros mais evidenciados neste tipo de teste são: erros de *interface*, funções incorretas ou ausentes, erros nas estruturas de dados ou no acesso a bancos de dados externos, erros de desempenho e erros de inicialização e término. A abordagem funcional visa complementar a abordagem que as técnicas de teste estrutural apresentam.

- Teste Baseado em Erros (Análise de Mutantes): conforme descrito por Demillo [DEM 87], é gerado um conjunto de programas semelhantes, denominados *mutantes*, a partir de um programa *P*, o qual assume-se como correto. Casos de teste são, então, gerados com o intuito de provocarem diferenças de comportamento em *P* e seus mutantes. Os chamados *operadores de mutação* são utilizados na geração destes mutantes, podendo ser associado a cada operador, uma classe de erros. Segundo Wong [WON 94], o processo de análise dos mutantes consiste em quatro etapas: (i) geração de mutantes; (ii) execução de *P* a partir de um conjunto de casos de teste *T*; (iii) execução dos mutantes a partir de *T* e (iv) análise dos mutantes.

Existe ainda, o teste de regressão, que alguns autores, como Herbert [HER 99b], o descrevem separadamente, devido à importância deste nos projetos de desenvolvimento de *software*. Tais autores, citam que este tipo de teste pode ser

considerado como integrante da categoria de teste funcional. Seu objetivo é assegurar que acréscimos de funcionalidade, aperfeiçoamentos de determinadas características, ou simplesmente correções de erros, não introduzam falhas, inexistentes até então. Isto é, modificações não podem causar erros que não existiam anteriormente [PRE 95].

A seguir, são descritas técnicas de teste para sistemas orientados a objetos, detalhando-se uma destas, o teste baseado em estados, a qual é apoiada pela ferramenta desenvolvida.

2.1 Teste de *Software* Orientado a Objeto

O paradigma da Orientação a Objetos (OO) surgiu trazendo consigo um novo enfoque, comparado aos métodos tradicionais de desenvolvimento de *software*. Entre as vantagens desta abordagem, pode-se citar a adoção de formas mais próximas dos mecanismos humanos com relação ao gerenciamento de complexidades inerentes ao desenvolvimento de produtos de *software*, buscando com isso um aumento de qualidade e maior produtividade, devido a uma de suas principais contribuições: a reutilização de código. Esta contribuição, entretanto, enfatiza que, assegurar que as classes desenvolvidas estejam corretas é essencial, o que deve ser feito o mais cedo possível, pois erros podem propagar-se durante sua reutilização em sub-classes.

O paradigma OO introduz novos conceitos e abstrações, como classes, métodos, mensagens, herança, polimorfismo e encapsulamento. De acordo com Kung [KUN 91], para este paradigma, o mundo real é constituído de objetos autônomos, concorrentes e com interações entre si, no qual cada um destes objetos possui seus próprios estados e comportamentos, semelhantes aos seus correspondentes no mundo real. Pode-se citar aqui, alguns benefícios da utilização deste paradigma: desenvolvimento mais rápido, maior qualidade, manutenção facilitada, estruturas de informação com melhor definição, bibliotecas de classes disponíveis, reutilização, entre outros.

Apesar desta abordagem apresentar várias vantagens em relação ao paradigma procedimental, a atividade de teste constitui um dos principais problemas no desenvolvimento de aplicações OO. Segundo Herbert [HER 99b], existe uma carência de técnicas bem estabelecidas para o teste de aplicações desenvolvidas sobre este paradigma, constituindo-se numa área nova de pesquisa e aplicação.

Do mesmo modo em que algumas das características encontradas em linguagens orientadas a objetos reduzem a probabilidade de determinados erros, outras favorecem o aparecimento de novas categorias dos mesmos [BIN 95]. Entre as características favorecedoras, cita-se o encapsulamento, o polimorfismo e ligação dinâmica.

Algumas facilidades do teste de *software* OO em relação ao procedimental são apresentadas por McGregor [MCG 96]:

- métodos e *interfaces* de classes são explicitamente definidos;
- número menor de casos de testes para cobertura são resultantes, devido ao número reduzido de parâmetros;
- reutilização de casos de teste devido à presença da característica de herança.

McGregor aponta também algumas desvantagens que devem ser consideradas [MCG 96]:

- a avaliação da correteza da classe é dificultada pela presença do encapsulamento de informações;
- o gerenciamento do teste é dificultado pelos múltiplos pontos de entrada (métodos) de uma classe;
- as interações entre os objetos são expandidas pelo polimorfismo e pela ligação dinâmica.

O fato das aplicações OO não serem executadas de forma seqüencial, evidencia uma importante diferença entre o tipo de teste procedimental e o orientado a objetos. Esta tecnologia não é de forma alguma, a motivação básica para a atividade de teste [BIN 95], introduzindo substanciais mudanças. Apesar de haverem similaridades aos testes convencionais, o teste OO possui significativas diferenças, nas quais incluem alguns problemas [BIN 95]:

- até que ponto as características de herança devem ser retestadas?
- quando pode-se confiar em resultados sobre o estado de determinados objetos não testados?

Os seguintes itens, a fim de verificarem a completeza de um sistema de *software* OO, devem estar presentes em sua especificação [MCG 96]:

- a especificação de todas as classes;
- pré-condições, pós-condições e invariantes de um método: as pré-condições representam condições que devem ser verdadeiras para a execução do referido método. Já as pós-condições devem ser verdadeiras após a execução do método. A invariante de uma classe é uma condição que assume-se que esteja verdadeira em todos os momentos, para determinada classe;
- a especificação dos métodos de uma classe, bem como seu modelo de estados.

Um plano de teste, derivado das especificações, deve ser elaborado [HER 99], com o objetivo de especificar a extensão do teste, ferramentas a serem utilizadas, critérios de teste, estimativa do tempo necessário para o teste, descrição dos casos de teste e dados de teste associados.

Uma classificação para os níveis de teste de código OO pode ser [SMI 92]:

- Teste de Classe: a menor unidade a ser testada é a classe [BIN 95];
- Teste de *Cluster*: conjunto de classes que interagem entre si, sendo este o principal fator a ser considerado neste tipo de teste;
- Teste de Sistema: neste nível, a funcionalidade é enfatizada.

Conforme descrito em [HER 99b], as classes são executadas através da execução dos objetos. Abaixo, são descritas as categorias de erros que podem estar presentes em classes, mensagens e métodos:

→ Erros de Classe:
  - não implementação de alguns estados definidos na especificação;
  - não implementação de alguns comportamentos definidos na especificação;

- a especificação da classe não reflete a descrição dos seus requisitos;
- violação do projeto e dos padrões de programação;
- implementação e documentação apresentam inconsistência.

→ Erros de Mensagens:

- parâmetros incorretos;
- os receptores das mensagens retornam valores impróprios ao objeto que enviou a mensagem;
- exceções não são implementadas, como retornos de mensagens;

→ Erros de Métodos:

- problemas de sintaxe na linguagem de programação;
- falhas para alcançar as pós-condições;
- problemas no fornecimento de valores apropriados como retornos;
- tempo de resposta insuficiente (aplicações *real-time*);
- código não executável.

A subseção seguinte descreve o teste de *software* OO baseado em estados, o qual avalia as mudanças de estados sofridos pelos objetos. Este teste é baseado no modelo dinâmico da classe [RUM 97] (diagrama / máquina de estados), a qual é formada por estados, transições, e pré e pós-condições associadas a transições, que são definidas como execução de métodos.

2.2 Teste de *Software* Orientado a Objeto Baseado em Estados

De acordo com Binder [BIN 94], um sistema orientado a objeto pode ser visto como uma sociedade de agentes cooperantes, onde cada agente é responsável por seu estado. Binder [BIN 95] define estado como um subconjunto de todas as combinações possíveis referentes aos valores dos atributos de uma classe.

Binder [BIN 95] define ainda que, comportamento é a seqüência de mensagens e respostas que um classe envia e/ou aceita. Já o comportamento de um sistema é o resultado da interação de comportamentos individuais. Com bases nestas definições, ressalta que para desenvolver sistemas orientados a objetos confiáveis, é necessário um alto nível de segurança, no qual (i) cada componente deverá comportar-se corretamente; (ii) comportamentos coletivos deverão estar corretos e (iii) nenhum comportamento coletivo incorreto será produzido.

Para Binder [BIN 94], o teste baseado em estados pode ser empregado com sucesso sobre sistemas OO por duas razões:

a) o comportamento das classes é bem modelado por máquinas de estados finitos. Métodos de uma classe devem ser usados seqüencialmente, entretanto, existe um número infinito de possibilidades de seqüências de ativação destes. Algumas destas seqüências podem ser proibidas pela especificação ou podem causar uma falha na implementação. Comportamento seqüencial arbitrariamente complexo pode ser modelado por máquinas de estados finitos. Sem dúvida, a maioria das metodologias adaptaram diagramas de estados para representar comportamento, sendo também desenvolvidos modelos formais de estados de objetos;

b) qualquer modelo de *software* usado para teste deverá auxiliar o encontro de falhas. Objetos preservam estado, mas o controle do estado (implementação

de uma seqüência de ativação aceitável) é tipicamente distribuído por um sistema inteiro. Falhas de comportamento (controle de estados) individuais e coletivos são resultados dessa estrutura complexa e implícita.

Neste tipo de teste, podem ser encontradas as seguintes falhas [BIN 95]: estados e transições não implementadas, estados e transições extras implementados, transições com entradas/saídas incorretas e interrupções de execuções.

Conforme descrito por Binder [BIN 95], o conjunto de valores encapsulados que a classe possui num determinado momento, determina o seu comportamento, sendo este controlado por esses valores encapsulados, seqüências de mensagens ou ambos. O principal objetivo deste tipo de teste, é realizar um teste no sistema OO, não tentando realizar todas as combinações possíveis. A seleção destas combinações deve ser feita de forma que estas testem comportamentos representativos do conjunto original [HER 97b].

McGregor [MCG 96], especifica alguns critérios para a cobertura do teste baseado em estados:

- Todos-Métodos: executar todos os métodos da classe, garantindo que todos os métodos necessários foram implementados. Todavia, não garante que todos os estados foram implementados também.
- Todos-Estados: "visitar" todos os estados, garantindo que todos os estados foram implementados, não garantindo, porém, que os métodos também foram;
- Todas-Transições: identificar métodos e estados não implementados, além da existência de transações extras;
- Todas-N-Transições: encontrar interrupções não necessárias, cobrindo combinações de transições;
- Todos-Caminhos: identificar todos os estados e métodos extras. Este critério despende grande esforço de teste, sendo praticamente impossível de ser utilizado [HER 99b], como o critério "todos-caminhos" do teste procedimental estrutural [MYE 79].

É ainda sugerido em [MCG 96], uma lista que o desenvolvedor deverá seguir para certificar-se que o referido teste foi realizado em sua totalidade, sendo esta lista aplicada às demais estratégias de teste de *software* OO:

- através de cada construtor da classe, testar os objetos criados;
- testar os métodos "get" e "set";
- testar as pós-condições de métodos modificadores;
- testar as combinações de métodos que acessam mesmos atributos;
- testar a liberação de memória alocada por cada objeto: este fato não deve ser levando em conta no caso de programa escritos em Java, pois a própria linguagem implementa este serviço (*garbage-collection*).

2.2.1 Diagrama de Estados

A máquina de estados finitos, ou diagrama de estados, faz parte da composição do modelo dinâmico proposto por Rumbaugh [RUM 97]. Sua finalidade é descrever os possíveis padrões dos objetos, atributos e ligações que podem existir em um sistema, ou seja, apresentar o comportamento interno do programa. Neste diagrama, existe a

descrição de como um objeto reage quando recebe um estímulo (uma mensagem, que consiste num evento do sistema), através da representação de estados anteriores e posteriores ao recebimento das mensagens, bem como das operações que são realizadas em transições (modificação do estado causada por um evento). O estado consiste nos valores dos objetos em um determinado momento da execução.

De acordo com Rumbaugh [RUM 97], este modelo dinâmico é formado por um conjunto de máquinas de estados finitos, sendo uma para cada classe, podendo funcionar de maneira concorrente, mudando de estado de forma independente.

Um estado representa a reação de um objeto aos eventos de entrada. É uma abstração dos valores de atributos e ligações de um objeto [HER 97b], sendo que, o estado corresponde ao intervalo entre dois eventos recebidos. Os eventos representam pontos no tempo e os estados intervalos de tempo [RUM 97].

Uma importante observação deve ser feita com relação à diferenciação entre eventos e chamadas a sub-rotinas (no paradigma procedimental): um evento é considerado uma transmissão unidirecional de informação de um objeto para outro [HER 97b], o qual diferencia-se da chamada da sub-rotina pelo fato desta retornar um valor. Embora exista o fato de que um objeto que envia uma mensagem espera uma resposta, esta por sua vez, está a cargo do segundo objeto (receptor da primeira mensagem).

Em uma máquina de estados finitos, os nodos representam estados e os arcos são transições, que são rotuladas com os nomes dos eventos. Ressalta-se que todas as transições que partem de um estado, correspondem a diferentes eventos. As condições para que a transição ocorra são apresentadas como expressões entre colchetes, podendo representar pré (quando a condição aparece antes do nome do método) ou pós-condições (quando a condição aparece depois do nome do método). A seqüência de transformações nos estados é descrita como conseqüência das seqüências de eventos.

O modelo dinâmico proposto por Martin [MAR 95] possui uma representação denominada esquema de eventos. Este é construído a partir de condições de controle, eventos, regras de gatilho e operações, representando a evolução de estados de um objeto, similarmente à máquina de estados finitos, admitindo com isto, estruturação e representação de concorrência.

Com o intuito de exemplificar os conceitos acima expostos, é apresentada na fig. 2.1 uma representação da Classe *Quarto*, a qual é parte de um hipotético sistema de automação de hotéis.

Nome:	***Quarto***	Descrição:
SuperClasse:	***Objeto***	
Operações:	**reservar**	reserva um determinado quarto para futura hospedagem
	ocupar	ocupa um quarto (efetivar hospedagem)
	limpar	coloca o quarto em estado de limpeza
	inc_item_cons	inclui itens consumidos pelos hóspedes
	liberar	libera o quarto para hospedagem
	cadastrar	inclui quarto no sistema
	reformar	coíbe a hospedagem, bloqueando o quarto para reforma
	desativar	cancela a utilização do quarto pelo sistema
Atributos:	Num_Quarto	Número de identificação do quarto
	Tipo	Tipo do quarto (simples, suíte, etc)
	Descrição	Descrição do quarto
	Preço	Valor da diária

FIGURA 2.1 – Representação da Classe *Quarto*

O controle sobre as operações descritas na fig. 2.1. é listado a seguir:

- Todas as transações são aceitas por um quarto, com exceção da transação "inc_item_cons";
- Um quarto estará liberado, desde que tenha sido cadastrado, possibilitando deste modo a efetivação de hospedagens e outras operações;
- Para um quarto poder ser ocupado, ele deverá estar liberado e ser ocupado pelo cliente que efetivou a reserva, na data correta;
- Para um quarto ser reformado, este deverá estar liberado e sem reservas no período;
- Para a realização de uma reserva num determinado período, não poderá haver outra reserva para este mesmo período;
- Para realizar a limpeza num quarto, este deverá estar vazio.
- Um quarto poderá ser removido do sistema sob duas condições: primeiro, deverá estar cadastrado ou estar liberado para hospedagem, desde que sem reservas. Segundo, a remoção só poderá ser feita pelo gerente.
- Um quarto desativado não poderá sofrer mais operações;

Concluída a descrição da classe, pré e pós-condições devem ser definidas para os métodos. As classes que recebem mensagens (classes clientes) devem estabelecer pré-condições, as quais definem regras para que estas mensagens sejam efetivamente aceitas. Já as pós-condições são estabelecidas pelas classes que enviam as mensagens

(classes servidoras), sendo estas condições utilizadas para definir resultados, ou ainda, definir o estado resultante da ativação do método.

Para a construção da máquina de estados finitos da classe Quarto, deve-se primeiramente identificar os estados possíveis dos objetos. São eles: (cadastrado, livre, ocupado, reservado, limpando, reformando e desativado). A fig. 2.2 apresenta a máquina de estados finitos relativa à classe *Quarto*.

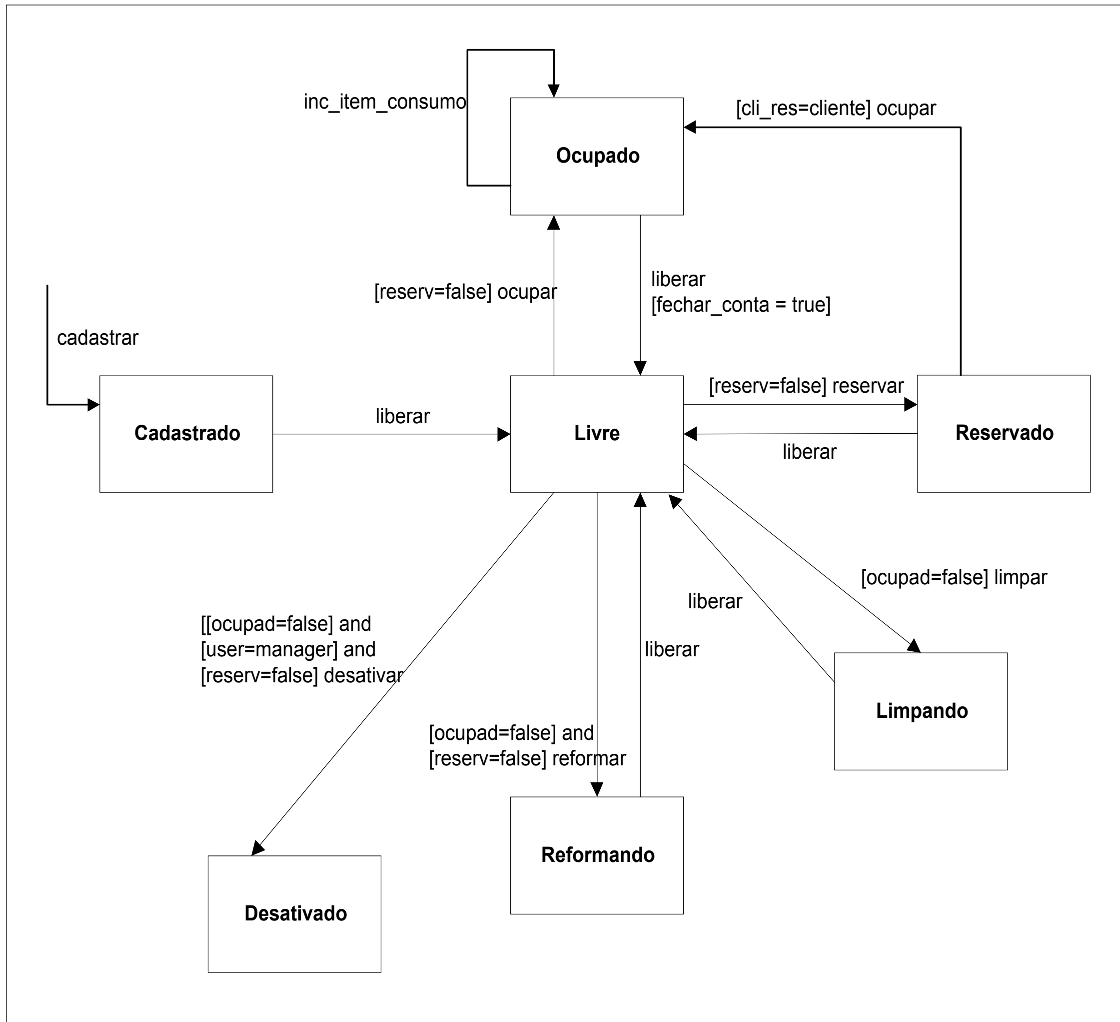


FIGURA 2.2 - Máquina de Estados Finitos da Classe *Quarto*

Com o objetivo de derivar seqüências de teste de transições, Binder [BIN 95] utiliza um procedimento descrito por Chow [CHO 78], sendo necessário à aplicação deste procedimento que exista:

- uma máquina de estados finitos completa;
- uma máquina de estados mínima (sem estados redundantes ou desnecessários);
- um estado inicial;
- todos os estados devem ser alcançáveis.

A seguir, é descrito o procedimento para a criação da árvore de transição: a raíz da árvore origina-se no estado inicial da máquina de estados. Um arco é desenhado para cada transição fora do estado raiz, até um nodo que represente o estado resultante. Este procedimento repete-se para cada nodo do estado resultante, até que este já apareça na

árvore na forma de um nodo anterior ou sendo este um estado final. Na fig. 2.3 é apresentada a árvore de transição da classe *Quarto*.

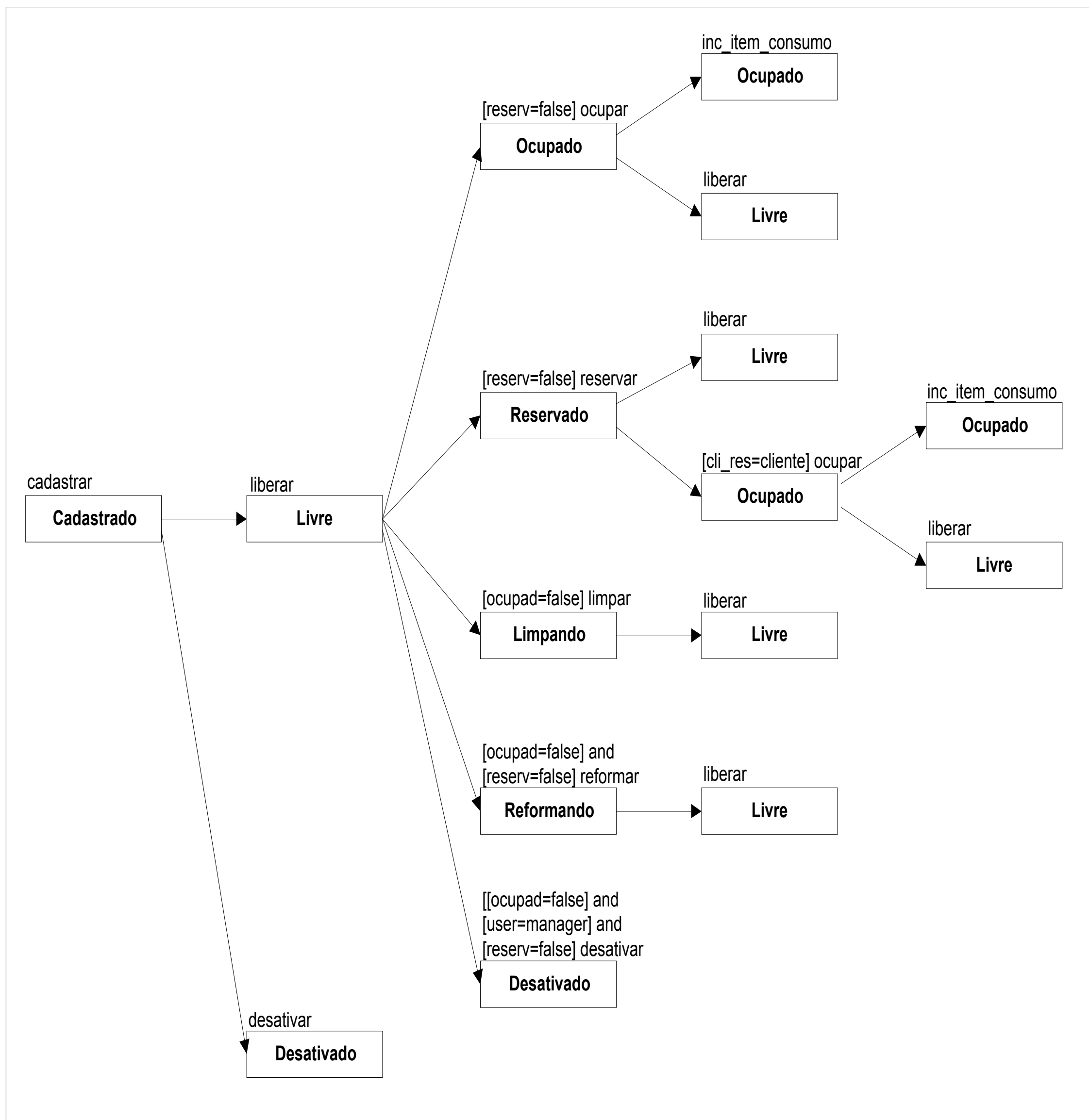


FIGURA 2.3 - Árvore de Transição da Classe *Quarto*

De acordo com Rumbaugh [RUM 97], o próximo passo é a transcrição das seqüências de teste de transições a partir da árvore gerada. Um caso de teste corresponde a cada arco ou cada conjunto de arcos numa mesma direção. Para completar-se o plano de teste, identifica-se os valores de parâmetros dos métodos, estados esperados e exceções de cada transição. Para a execução do teste, inicializa-se o objeto com seu estado inicial, aplicando a seqüência e, logo após, compara-se o estado resultante com o estado esperado.

Ressalta-se que, a árvore de transição, além de servir para determinar dados de teste, serve para dar a idéia, ou seja, identificar os caminhos da máquina de estados finitos que devem efetivamente serem testados. No caso de uma ferramenta de teste que tenha uma máquina de estados como base, o objetivo seria o de verificar se os objetos apresentam-se de forma consistente com relação as assertivas (pré, pós-condições e invariantes de classe associadas), as quais permitirão detectar os possíveis erros.

Conforme descrito por McGregor [MCG 96], os seguintes tipos de erros podem ser encontrados neste tipo de teste: transições perdidas, transições incorretas, ações de saída e estados incorretos.

2.3 O uso de Asserções no Teste de Estados

Nesta seção são apresentados os conceitos e algumas considerações sobre o uso de asserções no teste de estados, face a utilização destas na ferramenta desenvolvida.

O uso de asserções no teste de *software* OO baseado em estados constitui um importante mecanismo de auxílio a este tipo de teste, sendo utilizado por muitos desenvolvedores [VOA 97]. Voas ressalta que o uso das asserções demonstra que sua utilização na monitoração de programas pode determinar se estes comportam-se da maneira desejada.

Segundo Binder [BIN 99], uma asserção é uma expressão *booleana* que define condições necessárias para uma correta execução. O uso das asserções geralmente inclui:

- Verificação de condições que devem ser verdadeiras para a chamada de um método (pré-condição);
- Verificação de condições que devem ser verdadeiras quando do término da execução de um método (pós-condição);
- Verificação de condições que devem ser verdadeiras durante todo o tempo de vida de um objeto (invariante de classe).

As seguintes definições são apresentadas por Binder [BIN 99]:

Uma pré-condição é uma asserção avaliada na chamada de um método, antes da execução de qualquer código presente no corpo deste método. Expressa regras para os valores dos argumentos na chamada da mensagem e um estado requerido para sua correta execução. A pré-condição para uma função que calcule a raiz quadrada de número em ponto-flutuante (*sqrt(float x)*), por exemplo, deve ser: (*x>=0.0*). Desta forma, a pré-condição reflete o fato de que a função não é definida para números negativos.

Uma pós-condição define propriedades que devem ser satisfeitas quando do término da execução de um metódo. É avaliada depois que um determinado método finaliza seus procedimentos e antes que o resultado desta execução seja retornado ao cliente. Considerando a mesma função de cálculo da raiz quadrada, por exemplo, a pós-condição para esta função seria ter dados de saída positivos e menores que o número de entrada.

Uma invariante de classe especifica propriedades que devem ser sempre verdadeiras para todos os objetos da classe. A invariante da classe deve ser verdadeira depois da instanciação, antes e após a execução de cada método, e antes de sua destruição. Para uma classe *Pilha*, por exemplo, o topo da estrutura deve apontar para o elemento mais recente adicionado a esta estrutura.

Quando uma dessas condições torna-se falsa, ocorre o que Binder [BIN 99] denomina *violação de asserção*, significando que uma condição necessária não foi validada. Ressalta ainda que esta situação é geralmente chamada de *falha da asserção*, mas na realidade, a asserção não apresentou falha e sim sua especificação para um

estado que determinaria válido. Esta condição incorreta pode ser causada por uma falha na classe sob teste, em outra parte da aplicação, ou ainda em seu ambiente de execução [BIN 99].

Segundo Rosenblum [ROS 95], asserções foram efetivas na detecção de um amplo conjunto de erros em um estudo realizado. Relata que grande parte das falhas foram detectadas através do uso de asserções. De dezenove falhas, oito foram reveladas por violação de asserções e seis poderiam ter sidos detectadas por asserções que não foram especificadas. Binder [BIN 99] cita que grandes empresas líderes no desenvolvimento de *software*, como *IBM* e *Microsoft* utilizam-se extensivamente do uso de asserções como uma técnica de teste de *software* OO. Cita ainda, outras vantagens na utilização de asserções:

- estados encapsulados, bem como variáveis podem ser diretamente verificados;
- especificações incorretas de classes são evitadas, pois uma violação de uma asserção pode revelar um erro de especificação quando uma classe servidora, por exemplo, sinaliza uma mensagem legal com uma violação;
- uma violação numa pré-condição num método, geralmente revela um erro no método que enviou a mensagem (cliente) e não no método detentor da pré-condição;
- uma violação numa pós-condição num método, geralmente revela um erro no método detentor da pós-condição (servidor) e não no cliente;
- uma pós-condição pode revelar um estado corrompido resultante de um problema na especificação do algoritmo utilizado no método do objeto cliente, uso incorreto de um objeto servidor ou ainda um erro no próprio objeto servidor;
- a verificação das seqüências de ativação de métodos e pré-condições podem identificar tentativas de um objeto cliente enviar mensagens ilegais.

Várias metodologias de análise e projeto encorajam o uso de asserções, possuindo diferentes notações gráficas para representar aplicações OO. Na OMT (*Object Modeling Technique*) [RUM 97], a qual utiliza três tipos de modelos para descrever um sistema (modelo de objetos, dinâmico e funcional), o uso de asserções dá-se na utilização do segundo modelo. Utilizando múltiplos diagramas de estado, o modelo dinâmico mostra o padrão de atividade do sistema em projeto. Condições são utilizadas como guardas nas transições nos diagramas de estados. Estas transições disparam somente quando seus eventos ocorrem e tais condições são verdadeiras. Estas condições são apresentadas como expressões *booleanas* que acompanham o nome do evento.

Já a metodologia UML (*Unified Modeling Language*), a qual representa diferentes aspectos de um sistema OO usando notações gráficas próprias, utiliza-se da OCL (*Object Constraint Language*) para a especificação de restrições e expressões associadas a seus modelos. A OCL não é uma linguagem de programação, mas sim uma linguagem para modelagem de expressões. Uma de suas utilizações é referente à especificação de invariantes de classes, pré e pós-condições de métodos.

Conforme apresentado na Seção 2.1, McGregor [MCG 96] sugere que a especificação de um *software* está completa quando, entre outras coisas, estejam especificadas as pré-condições, pós-condições e invariantes de classe.

Observa-se que, a utilização de asserções na especificação e modelagem de *software* OO é amplamente aceita, da mesma forma que sua utilização no processo de teste de aplicações OO.

3 Reflexão Computacional

Patti Maes [MAE 87] apresentou em 1987 o conceito de reflexão computacional: é a atividade executada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações. Desta maneira, reflexão pode ser entendida como uma forma de introspecção, pois o sistema pode tentar tirar conclusões sobre suas próprias computações, podendo estas serem posteriormente afetadas.

Segundo Lisboa [LIS 97], reflexão computacional é uma técnica de programação que permite ao programador obter informações a respeito do próprio programa, com o objetivo de monitorá-lo, adicionar novas funcionalidades e mesmo fazer alterações adaptativas em tempo de execução. O mecanismo de reflexão computacional pode ser utilizado em qualquer paradigma de programação: funcional, procedural, lógico ou orientado a objetos, sendo neste último melhor empregado devido às características de modularidade e flexibilidades encontradas nas linguagens que o implementam [MAE 87]. O objetivo da reflexão não se refere ao auxílio de atividades referentes ao domínio externo das aplicações, e sim na contribuição para sua organização interna bem como *interface* com o mundo externo [RUB 98]. Disso resulta que o uso de reflexão é útil em atividades administrativas da aplicação, tais como estatísticas de desempenho, otimização, distribuição, tolerância a falhas e, evidentemente no processo de teste e depuração de *software*.

Steel [STE 94] enfatiza que, reflexão computacional é a capacidade de um sistema computacional de interromper o processo de execução (por intermédio de um erro, por exemplo), realizar computações ou fazer deduções no meta-nível e retornar ao nível de execução traduzindo o impacto das decisões, para então retornar o processo de execução.

O conceito de reflexão computacional, introduzido como mecanismo para suportar comportamento reflexivo em linguagens de programação, apresenta um vasto campo de aplicação, entre as quais cita-se: teste e depuração de *software*, concorrência e distribuição, mecanismos para gerenciamento de transações atômicas, tolerância a falhas, entre outras.

A reflexão computacional no modelo de objetos (OO) concentra-se na obtenção de informações sobre propriedades referentes as classes e objetos da aplicação em execução, podendo, entretanto, efetuar modificações nestas propriedades (dados). De acordo com Rubira [RUB 97], a programação reflexiva encoraja uma organização modular de sistemas, pois tem-se a nítida separação entre objetos da aplicação e objetos gerenciadores da aplicação (meta-objetos).

Através da reflexão computacional é possível controlar o comportamento do objeto ao receber uma mensagem, permitindo que se verifique as informações sobre o processo de execução de métodos, com o objetivo de monitorá-la.

3.1 Arquitetura Reflexiva

Durante a execução de uma aplicação, as computações realizadas referem-se a dados do seu próprio domínio, objetivando satisfazer os requisitos da aplicação em particular [LIS 98]. Entretanto, quando há computação reflexiva, as computações

realizadas referem-se a dados do próprio sistema, com o intuito de resolver seus próprios problemas. Por *domínio externo* do sistema ou *domínio da aplicação* entende-se a representação da informação relacionada com a aplicação (parte do mundo externo), e por *domínio interno* do sistema o conteúdo da informação sobre o próprio sistema, no qual inclui também a representação interna do domínio da aplicação [LIS 98].

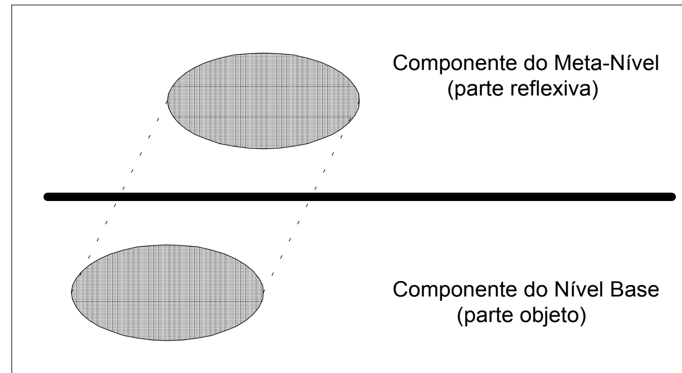


FIGURA 3.1 - Arquitetura Reflexiva

A arquitetura reflexiva é composta por dois níveis, um denominado meta-nível e outro denominado nível-base. No meta-nível estão as estruturas de dados bem como as ações a serem executadas sobre o sistema objeto presente no nível-base. As computações realizadas no meta-nível são feitas sobre dados que representam informações para o programa de nível-base, o qual realiza computações sobre seus dados, atendendo desta forma aos requisitos da aplicação (domínio externo).

A meta-arquitetura deve apresentar as seguintes propriedades [BER 99]:

- Os programas de meta-nível devem ser reutilizáveis;
- Os programas de nível-base devem ser reutilizáveis;
- A arquitetura deve ser capaz de resolver diversos tipos de problemas.

Conforme descrito por Lisboa [LIS 98], um sistema reflexivo possui dois componentes:

a) subsistema objeto, que realiza computações sobre o domínio externo;

b) subsistema reflexivo, o qual realiza computações sobre o sistema objeto. Um fato importante a considerar é que o componente reflexivo pode atuar sobre a estrutura ou sobre o comportamento do sistema objeto.

É descrito por Silva [SIL 97], que uma arquitetura pode ainda ser recursiva, onde o nível superior controla o inferior, isto é, um meta-nível é o nível-base para outro meta-nível, caracterizando uma torre de reflexão. Uma possível aplicação deste conceito, por exemplo, seria a instalação, por parte de uma ferramenta de teste, de um segundo meta-nível a uma aplicação já reflexiva, com o objetivo de tornar possível a realização de testes sobre esta.

3.2 Modelos de Reflexão

A computação reflexiva, no modelo de objetos, pode ser realizada sobre classes ou objetos. Quando a reflexão é realizada sobre classes (denominado modelo de meta-

classes), o meta-nível é composto por meta-classes, contendo as informações estruturais sobre os componentes do nível-base. Com o objetivo de ilustrar os modelos de reflexão, será utilizada a estrutura de classes com seus respectivos métodos e atributos apresentada na fig. 3.2:

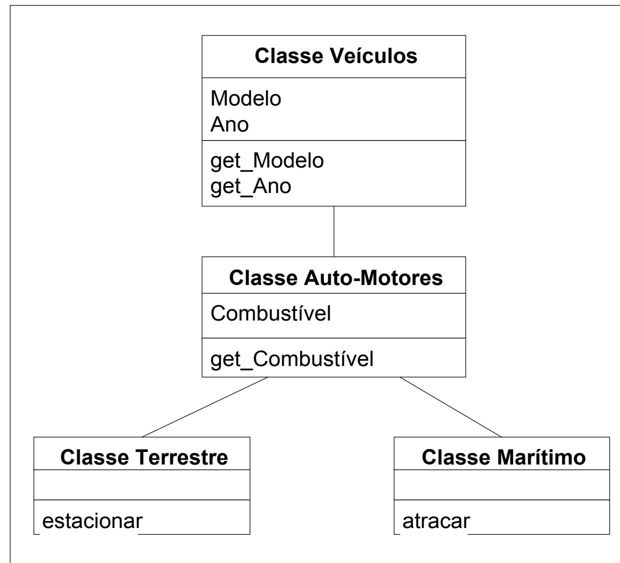


FIGURA 3.2 - Hierarquia de Classes

A fig. 3.3 exemplifica a reflexão por meta-classes ilustrada pela meta-classe da classe *Terrestre*. Segundo Lisboa [LIS 96], este modelo apresenta menor flexibilidade, pois um único meta-objeto é compartilhado por todos os objetos da mesma classe. No segundo caso, (denominado modelo de meta-objetos), o meta-nível é composto por meta-objetos, contendo as informações (descrições) relacionadas ao comportamento dos componentes do nível-base. Neste modelo, a flexibilidade é maior, pois o meta-objeto possuirá as informações de um objeto específico [SIL 97]. A fig. 3.4 exemplifica o modelo de reflexão por meta-objetos.

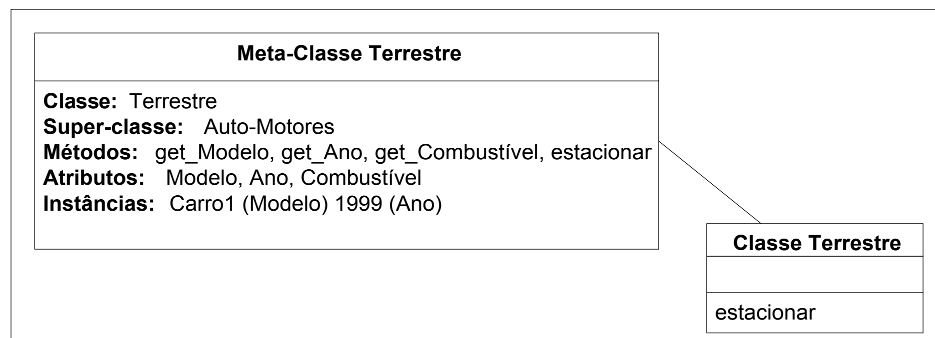


FIGURA 3.3 - Reflexão de Meta-Classe da classe *Terrestre*

Booch [BOO 94] e Graube [GRA 89] apresentam uma definição formal para meta-classe: uma meta-classe MC_X, é uma classe na qual descreve a estrutura de classe “x”, e cujas instâncias também são classes. Analisando-se tal definição, é possível afirmar que uma meta-classe é a classe de uma classe. Foote [FOO 93] apresenta uma definição para meta-objeto: meta-objetos são objetos que definem, implementam, dão

suporte ou participam de alguma forma na execução da aplicação, ou dos objetos de nível-base.

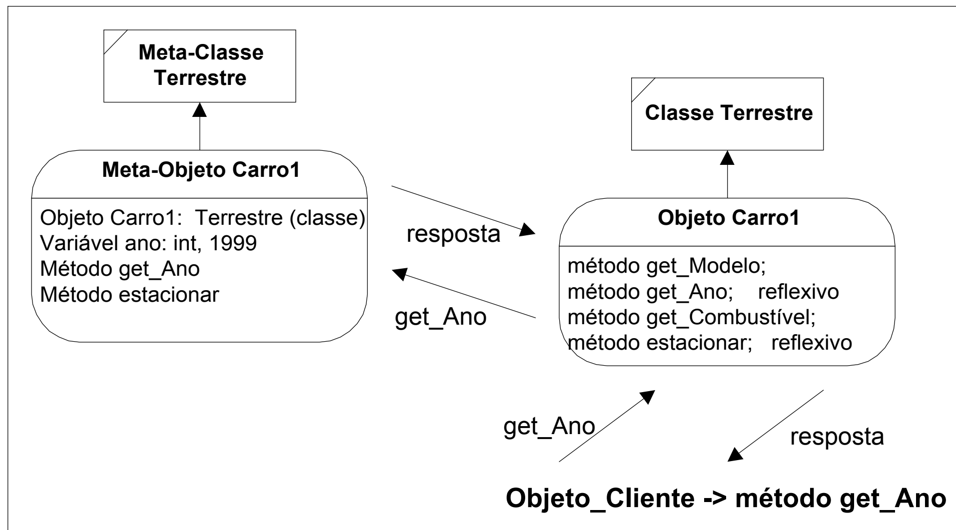


FIGURA 3.4 - Reflexão de Meta-Objetos (adaptado de [BER 99])

3.3 Estilos de Reflexão

No modelo de reflexão de meta-classes, ocorre a reflexão estrutural, a qual permite obter informações (permitindo também alterações) sobre a estrutura da classe refletida. Entre as informações e alterações suportadas pela reflexão estrutural estão [BER 99]: identificar sub-classes, superclasses, atributos, métodos e *interfaces* de uma classe, alterar classes (métodos e atributos), além de possibilitar criar novas classes e redefinir classes existentes, bem como eliminá-las.

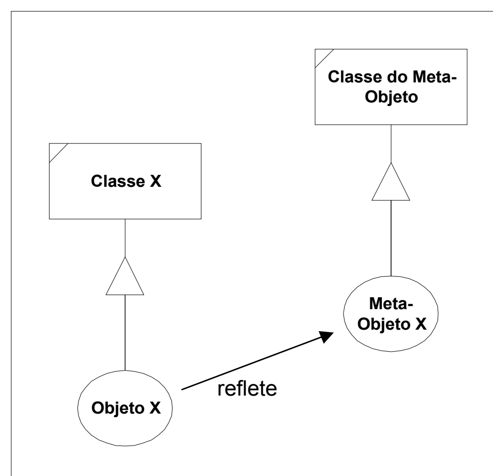


FIGURA 3.5 - Reflexão Comportamental [PIN 98]

Já o modelo de reflexão de meta-objetos utiliza a Reflexão Comportamental (fig. 3.5), permitindo que um meta-objeto interfira no comportamento do objeto refletido. A reflexão comportamental de um objeto consiste na atividade realizada pelo seu meta-objeto, com o intuito de obter informações e realizar transformações sobre o

comportamento do objeto. Isto é, a função principal do meta-objeto é explicitar a reação de um objeto frente ao recebimento de uma mensagem, permitindo desta forma a intervenção da computação. Através da busca e coleta destas informações sobre o processo de execução, pode-se obter: estatísticas de desempenho, informações para fins de depuração e monitoração, entre outras.

A conexão entre o meta-nível e o nível-base é estabelecida através do protocolo de meta-objetos (MOP – *MetaObject Protocol*). Este protocolo define a *interface* entre objetos e meta-objetos. Conforme descrito por Ferber [FER 89], este protocolo preocupa-se com questões como:

a) Quais entidades devem ser transformadas em algo que possa sofrer operações no meta-nível?
b) De que forma é implementado o relacionamento entre o nível-base e o meta-nível ?
c) Quando o sistema passa para o meta-nível ?

Com relação à primeira questão, a dúvida refere-se a que dados podem ser manipulados no meta-nível. De acordo com Lisboa [LIS 98], a atividade computacional do nível-base é transformada em dados para o nível superior (meta-nível), o qual determina quais computações serão ou não realizadas. A esta operação dá-se o nome de reificação ou materialização [FER 89] [NAK 92]. Reificação é a transformação de informações sobre a execução de um programa orientado a objetos em dados disponíveis ao próprio programa [LIS 98]. Deste modo, as computações reflexivas são realizadas sobre os objetos reificados, os quais constituem as meta-informações.

As meta-informações subdividem-se em estáticas e dinâmicas. Estáticas quando envolvem informações estruturais em tempo de compilação, e dinâmicas, quando estas informações estão relacionadas com o processo de execução.

Com relação à segunda questão (b), os meta-objetos possuem meta-*interfaces*, nas quais são oferecidos os serviços genéricos da aplicação. Eles podem ainda, possuir uma *interface* operacional, contendo os serviços oferecidos ao nível-base ou a outros meta-objetos.

A última questão (c), refere-se ao fato de quem e como inicia o processo de reflexão. De acordo com Maes [MAE 88], duas soluções são possíveis: (i) atribuição da responsabilidade ao objeto de nível-base, que contém neste caso código mencionando seu meta-objeto e (ii) atribuição da responsabilidade ao sistema. Neste último, a ativação do meta-objeto pelo sistema ocorre quando um evento envolvendo o objeto referente ocorre, podendo ser uma mudança de estado de variáveis de instância (variáveis reflexivas) ou recebimento de mensagens dirigidas a determinados métodos (métodos reflexivos) [LIS 98].

3.4 Linguagens Reflexivas

Uma linguagem de programação que possui a habilidade de permitir a modificação em sua própria implementação é denominada reflexiva [SIL 97]. Através da utilização de protocolos de meta-objetos (MOP), que constituem um conjunto de regras, estas linguagens alteram a implementação de objetos com a utilização de meta-objetos. Entretanto, para uma linguagem ser considerada reflexiva, ela deve suportar os conceitos de reflexão anteriormente apresentados, bem como, apresentar mecanismos

que possibilitem a conexão entre o meta-nível e o nível-base, tornando disponível no meta-nível os dados obtidos no nível-base.

De acordo com Lisboa [LIS 98], para que exista reflexão computacional, informações (para implementação de características reflexivas) que o processador da linguagem detém durante o processo de compilação como por exemplo, nomes de todas as classes e sua hierarquia de herança, nome de métodos e atributos, entre outros, não devem ser descartadas, devendo estas estarem disponíveis ao programador na forma de uma *interface* ou processador da linguagem.

São encontrados três tipos de categorias de reflexão computacional [BER 99]:

a) Reflexão no momento da compilação: certas expressões são controladas durante a compilação por um programa de meta-nível, modificando desta forma, este processo. Normalmente esta categoria de reflexão trabalha com entidades como classes e tipos, não fornecendo informações dinâmicas.
b) Reflexão em tempo de carga: ocorre reflexão antes da aplicação ser executada, durante o processo de carga (*load*).
c) Reflexão em tempo de execução: durante a execução do programa de nível-base, este poderá estar sendo modificado ou monitorado de acordo com a especificação do programa de meta-nível. Esta categoria possui alta flexibilidade, permitindo a adaptação do comportamento do programa de nível-base de forma dinâmica.

A utilização de linguagens reflexivas, introduz um novo conceito ao processo de desenvolver aplicações, sendo necessária a adoção de um novo modelo de abstração [LIS 98]: a separação de domínios. Programas reflexivos possuem sua estrutura na forma de dois programas: um programa de meta-nível e outro de nível-base. A explicação para ambos é encontrada em Kiczales [KIC 92]. O comportamento desejado pelo cliente é expresso pelo programa de nível-base, em termos de funcionalidades fornecidas pelo ambiente de suporte. Já o programa de meta-nível permite a adequação de aspectos particulares de implementação do ambiente de suporte, de uma forma que melhor atenda as necessidades do programa de nível-base.

Nas seguintes subseções, é abordado o tema reflexão computacional na linguagem Java, sendo detalhados alguns protocolos de reflexão definidos para tal linguagem. Será dada ênfase ao protocolo reflexivo Guaraná, utilizado na implementação da ferramenta *KTest*.

3.5 Reflexão na linguagem de programação Java

A linguagem Java implementa o conceito de reflexão computacional. Através da especificação de sua máquina virtual (JVM – Java Virtual Machine), Java estabelece que todas as classes e interfaces são objetos do tipo Class. Estes objetos são automaticamente criados quando as classes são carregadas para serem executadas, contendo as seguintes informações: nome da classe, a superclasse, interfaces implementadas, entre outras. Entretanto, a classe Class caracteriza uma reflexão introspectiva, pois esta apenas atua com um descritor de classes, não permitindo alterações, quer seja na sua estrutura, quer seja nos seus atributos.

A fig. 3.6, exemplifica a utilização de reflexão computacional para a obtenção do nome da classe.

```
public class Exemplo1 {

    public static void main ( String args[] ) {

        Integer i = new Integer(5);
        Class   c = i.getClass();
        System.out.println ("Classe do objeto: " + c.getName() );
    }
}
```

FIGURA 3.6 - Obtenção do nome da classe

Introspecção é a possibilidade de um sistema obter informações a respeito de seus componentes, não sendo possível a alteração dos mesmos.

A seguinte tabela apresenta os métodos reflexivos da JVM padrão (JDK 1.0):

TABELA 3.1 – Métodos Reflexivos da JVM padrão

Métodos
`public static Class forName (String ClassName)` → retorna o objeto *Class* associado com a classe dada como parâmetro.
`public Object newInstance()` → cria uma nova instância da classe
`public String toString()` → converte o objeto para uma string
`public boolean isInterface()` → determina se o objeto Class especificado representa um tipo interface
`public String getName()` → retorna o nome do objeto juntamente com o seu tipo (Class, Interface, etc.)
`public Class getSuperclass()` → retorna a superclasse do objeto
`public Class[] getInterfaces()` → retorna as interfaces implementadas pela classe
`Public Class[] getClass()` → retorna um vetor contendo objetos Class os quais representam todas as classes e interfaces que são membros da classe representada pelo objeto Class.

Na versão 1.0, a reflexão era realizada somente pela classe *Class*. A partir de 1996, novos mecanismos de reflexão computacional foram incorporados à linguagem Java (discutidos na Seção 3.6), com a inclusão de protocolos de meta-objetos, inseridos em seu ambiente de execução, dando maior flexibilidade à utilização de reflexão nesta linguagem.

3.6 Protocolo Reflexivo CoreJava

Este protocolo foi desenvolvido pela Sun Microsystems®, cuja primeira versão apresentada em 1996 vem sofrendo várias alterações. Através deste protocolo foram ampliadas as informações obtidas sobre o programa objeto, durante o processo de execução. Com a inclusão de novos métodos à classe *Class*, bem como a inclusão de novas classes no ambiente JDK 1.1, outras características reflexivas estão disponíveis. Foi adicionado o pacote *java.lang.reflect*, que é composto por cinco classes: *Array*, *Field*, *Method*, *Constructor*, *Modifier* e uma *interface Member* [COR 98]

Este protocolo constitui uma API (*Applications Programming Interface*), a qual suporta introspecção sobre classes e objetos da JVM.

Com a utilização desta API, é possível [GRE 97]:

- identificar a classe de um objeto;
- obter informações sobre modificadores, campos, métodos, construtores e superclasses de uma determinada classe;
- encontrar constantes e declarações de métodos em uma *interface*;
- criar instâncias de classes dinamicamente;
- obter e determinar valores de campos de objetos dinamicamente;
- invocar dinamicamente métodos de um objeto;
- criar matrizes dinamicamente, podendo também, redimensioná-las.

Conforme Niemeyer [NIE 97], o gerenciador de segurança é que realiza o acesso à API de reflexão. Deste modo, sendo uma aplicação executada localmente, ela terá acesso a todas as funcionalidades fornecidas por esta API. Caso contrário, se o código for carregado de algum *site*, por exemplo, apenas os membros públicos de classes públicas podem ser acessados. A violação de segurança nunca pode ser afetada, mesmo utilizando-se desta técnica. Um objeto não poderá utilizar reflexão para acessar métodos e/ou dados privados nos quais ele normalmente não teria acesso [NIE 97].

Neste protocolo, as meta-classes estão no meta-nível e os objetos *Class*, responsáveis pelo mapeamento das classes no nível-base. As meta-informações são disponibilizadas através de classes específicas. Variáveis, construtores e métodos são acessados através do pacote *java.lang.reflect*.

O protocolo CoreJava não apresenta característica de interceptação de chamadas a métodos, resultando disto que o meta-nível não dispõe de informações dinâmicas, como por exemplo, os argumentos usados na chamada de um método [LIS 98].

Conforme sugere Lisboa [LIS 98], a flexibilidade da reflexão pode ser utilizada na construção de um programa que apresente as seguintes características:

- solicita ao usuário o nome de uma classe;
- caso exista, esta classe é carregada dinamicamente;
- informações são recuperadas desta classe, como nome, atributos, métodos e argumentos
- objetos desta classe podem ser instanciados, executando-se seus métodos.

3.6.1 Obtenção de informações das Classes

Para cada classe, o ambiente de execução Java (JRE – *Java RunTime Environment*) mantém um objeto *Class* que contém informações sobre a classe. Este objeto também representa, ou reflete, a classe. Utilizando a API de reflexão, é possível utilizar estes objetos com o intuito de obter informações relativas aos construtores, métodos e campos definidos na classe, além de representarem as *interfaces* implementadas na classe [GRE 97].

Com a finalidade de exemplificar os conceitos abordados, foi desenvolvida uma classe, baseada nas especificações da fig. 3.2, e um programa, denominado *Mostra_Info*, cuja finalidade é recuperar informações da classe definida em tempo de execução, utilizando o processo de reflexão estrutural. O código do programa *Mostra_Info* é apresentado na fig. 3.7 e, o resultado de sua execução na fig. 3.8. O modelo adotado neste trabalho para a explicação do código-fonte dos programas exemplos será o seguinte: linhas numeradas para os códigos dos programas e suas respectivas saídas são fornecidas. As linhas dos códigos-fonte serão referenciadas por números entre chaves "{ }", e suas saídas por sinais de menor e maior "< >", respectivamente. Demais exemplos sobre os conceitos abordados podem ser encontrados em [SIL 2000].

```
1   import java.lang.reflect.*;
2   public class Mostra_Info {
3       public static void main(String args[])
4     {
5       boolean finall = true;
6       try {
7             Class cls = Class.forName(args[0]);
8             do {
9             System.out.print  ("CLASSE:" + cls.getName() );
10            System.out.println ("        SUPER-CLASSE:"
11                + cls.getSuperclass().getName() );
12
13            Constructor ctorlist[]  =
14                 cls.getDeclaredConstructors();
15            System.out.println ("Construtores:");
16            for (int i = 0; i < ctorlist.length; i++) {
17                Constructor ct = ctorlist[i];
18                System.out.println("             Nome: " +
19                     ct.getName());
20                System.out.println("             Classe
21                     Declarado: " + ct.getDeclaringClass());
22                Class pvec[] = ct.getParameterTypes();
23                for (int j = 0; j < pvec.length; j++)
24                    System.out.println("             Param #" +
25                        j + " " + pvec[j]);
26                Class evec[] = ct.getExceptionTypes();
27                for (int j = 0; j < evec.length; j++)
28                    System.out.println( "            Tipo Exc #"
29                        + j + " " + evec[j]);
30                System.out.println("             --------");
31        }
32
33        Method methlist[]   = cls.getDeclaredMethods();
34        System.out.println ("Metodos:");
35        for (int i = 0; i < methlist.length; i++) {
36           Method m = methlist[i];
37           System.out.println("            Nome: "
38               + m.getName());
39              System.out.println("            Classe Declarada:
```

```
40                     " + m.getDeclaringClass());
41               Class pvec[] = m.getParameterTypes();
42               for (int j = 0; j < pvec.length; j++)
43                  System.out.println("              Param #" + j
44                            + " " + pvec[j]);
45               Class evec[] = m.getExceptionTypes();
46               for (int j = 0; j < evec.length; j++)
47                  System.out.println("              Tipo Exc #" + j
48                            + " " + evec[j]);
49               System.out.println("              Tipo de Retorno = "
50                        + m.getReturnType());
51               System.out.println("              --------");
52            }
53
54               Field fieldlist[] = cls.getDeclaredFields();
55               System.out.println ("Campos:");
56               for (int i = 0; i < fieldlist.length; i++) {
57                   Field fld = fieldlist[i];
58                   System.out.println("              Nome: "
59                       + fld.getName());
60                   System.out.println("              Classe
61                       Declarado: " + fld.getDeclaringClass());
62                   System.out.println("              Tipo: "
63                       + fld.getType());
64                  int mod = fld.getModifiers();
65                  System.out.println("              Modificadores: "
66                           + Modifier.toString(mod));
67                  System.out.println("              --------");
68            }
69
70               Class super_cls = cls.getSuperclass();
71               if ( super_cls == null
72                      || (super_cls.getName().indexOf(".") != -1) )
73                   finall = false;
74               else
75                   cls = super_cls;
76
77            } while ( finall == true );
78
79         }
80         catch (Throwable e) {
81            System.err.println(e);
82         }
83      }
84   }
```

FIGURA 3.7 – Programa *Mosta_Info*

```
1    CLASSE:Terrestre         SUPER-CLASSE:Auto_Motor
2        Construtores:
3                  Nome: Terrestre
4                  Classe Declarada: class Terrestre
5                  Param #0 class java.lang.String
6                  Param #1 int
7                  Param #2 class java.lang.String
8                  --------
9        Metodos:
10                 Nome: estacionar
11                 Classe Declarada: class Terrestre
12                 Tipo de Retorno = void
13                 --------
14       Campos:
15   CLASSE:Auto_Motor        SUPER-CLASSE:Veiculo
16       Construtores:
17                 ...
18
```

FIGURA 3.8 – Resultados da execução do programa *Mosta_Info*

3.7 Protocolo Reflexivo MetaJava

Desenvolvido por Michael Golm [GOL 97], o protocolo reflexivo MetaJava tem por objetivo, além de permitir reflexão estrutural, prover também meios para a realização de reflexão comportamental [KLE 97]. O primeiro tipo de reflexão realiza a reificação de aspectos estruturais da aplicação, conforme descrito anteriormente, como herança, estrutura das classes, métodos implementados, entre outros. O segundo tipo refere-se à reificação de computações e seus comportamentos. O protocolo MetaJava foi implementado através de uma extensão da máquina virtual (JVM).

Neste protocolo, as computações realizadas no nível-base causam eventos, os quais constituem o processo de transferência de controle do nível-base para o meta-nível, conforme mostra a fig. 3.9. O meta-nível avalia os eventos e reage de maneira específica. Segundo Golm [GOL 97], a passagem de eventos é realizada de forma síncrona, ou seja, as computações no nível-base são suspensas até que os meta-objetos processem os eventos, dando com isto ao meta-nível um completo controle sobre as atividades do nível-base. Se um meta-objeto recebe um evento *method-enter*, por exemplo, seu comportamento padrão seria o de executar tal método. Entretanto, este meta-objeto pode também sincronizar a execução deste com outro método do objeto de nível-base, podendo ainda, colocar este método em uma fila para posterior execução, retornando imediatamente ao objeto chamador, além de possibilitar que a execução do método seja realizada em outro servidor [KLE 96]. É possível também, que um método de um objeto do nível-base invoque um método de um meta-objeto, sendo isto chamado de *meta-interação explícita*, a qual é usada para controlar o meta-nível a partir do nível-base [KLE 97].

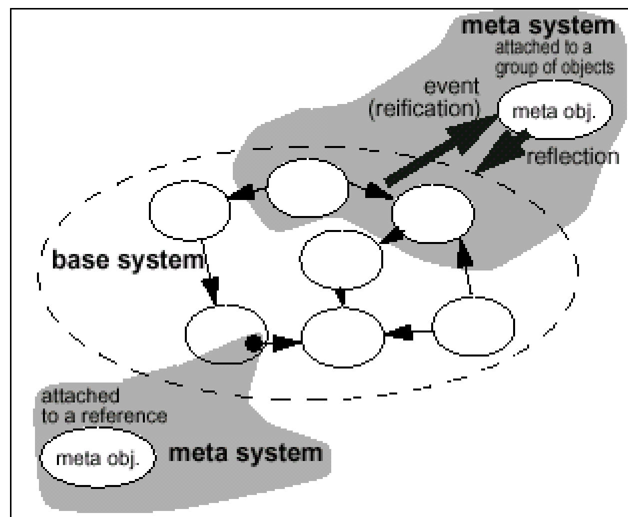


FIGURA 3.9 – Modelo computacional de reflexão comportamental [KLE 96]

Foram definidos diversos eventos para os mecanismos da máquina virtual (JVM), tais como invocações de métodos, acesso a variáveis, operações com monitores e criação de objetos. Deste modo, o evento manipulador de um meta-objeto intercepta um evento de um objeto que contém as informações necessárias para realizar as operações requisitadas.

Os seguintes aspectos comportamentais são implementados por este protocolo:

- invocação de métodos;
- acessos a variáveis de instância;
- acesso a *locks*;
- mapeamento dos nomes das classes;
- carga (*loading*) de classes;
- instanciação de objetos, podendo alterar a representação destes.

Através da reificação dos mecanismos acima citados, é provida a capacidade de reflexão comportamental do protocolo. Se o programa de meta-nível pode implementar estes mecanismos, ele deve ser capaz também de extrair informações do programa de nível-base, enfatizando que é necessário suportar reflexão estrutural.

No protocolo MetaJava, um meta-objeto pode ser ligado não somente a objetos, mas também a referências ou classes. Quando ocorre a ligação a um objeto, serão entregues a este meta-objeto todos os eventos gerados pelo objeto a ele ligado. No caso de um meta-objeto estar atribuído a uma determinada classe, então todos os eventos gerados pelos objetos instanciados desta classe serão entregues ao meta-objeto. Já no caso de um meta-objeto estar ligado a uma referência, somente os eventos gerados por operações usando esta referência serão repassados ao meta-objeto.

Para realizar as ligações acima descritas, os meta-objetos devem acessar um conjunto de métodos nos quais podem manipular o estado interno da máquina virtual. Estes são chamados de métodos da *meta-interface* da máquina virtual. De acordo com Golm [GOL 97], somente a classe *MetaObject*, suas sub-classes e membros do pacote *meta* podem invocar tais métodos. Uma lista com os principais métodos da meta-*interface* são apresentados na tabela 3.2

TABELA 3.2 – Métodos para vinculação de meta-objetos

Métodos
`void attachObject(Object obj)` → vincula um meta-objeto a um objeto
`Object attachReference(Object obj)` → vincula um meta-objeto a uma referência
`void attachClass(Class c)` → vincula um meta-objeto a uma classe
`MetaObject findMeta(Object base)` → encontra o meta-objeto responsável por um objeto de nível-base
`Object doExecuteObject(EventMethodCall event)` → executa um método, como se o método fosse invocado por um objeto de nível-base
`Object createNewInstace(EventObjectCreation event)` → cria uma nova instância da classe
`void createStubs(Object obj, String methods[])` → cria métodos *stubs* para o objeto de nível-base, no qual substitui o método especificado.
`Object cloneReference(Object ref)` → cria uma nova referência que aponta para o objeto ref, sendo usado para atribuir meta-objetos a referências.
`Class cloneClass(Object ref)` → cria um *clone* da classe do objeto referência. O *clone* torna-se a classe deste objeto, sendo usado quando um meta-objeto é atribuído a um objeto evitando interferência com outras instâncias da mesma classe.

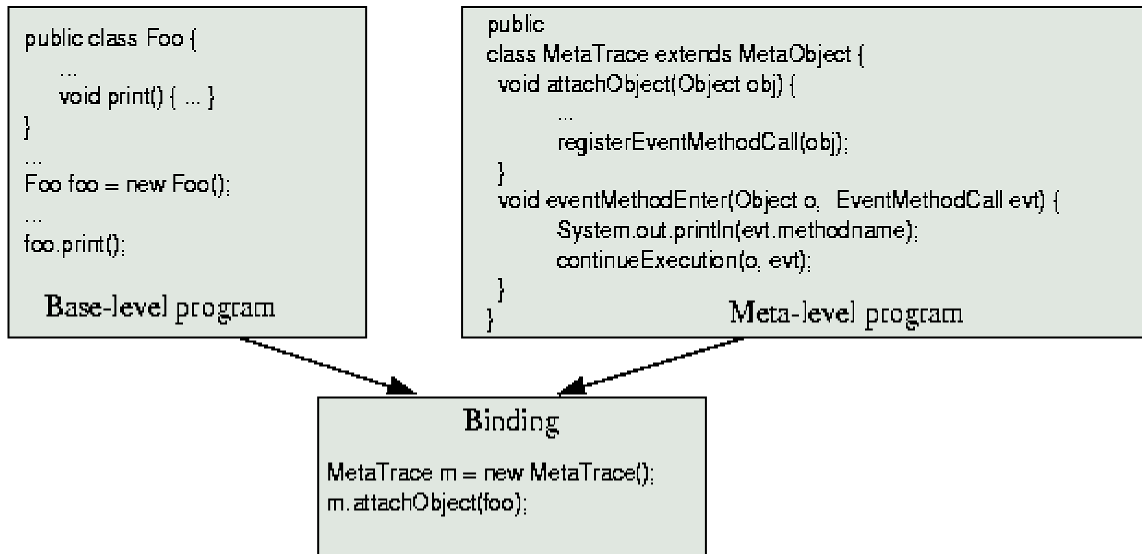


FIGURA 3.10 – Exemplo de ligação entre o nível-base e o meta-nível [GOL 97]

A fig. 3.10 mostra um exemplo de como realizar a ligação entre o meta-nível e o nível-base, sem modificações nos códigos dos níveis envolvidos.

3.7.1 Arquitetura do Protocolo

A arquitetura do protocolo MetaJava consiste em duas camadas [GOL 97], conforme demonstra a fig. 3.11: (i) a camada inferior, na qual estende a máquina virtual Java (JVM), chamada de *interface* de meta-nível (MLI - *meta-level interface*); (ii) a camada superior, escrita em Java, a qual provê os mecanismos dos métodos descritos na tabela acima, fornece uma abstração de eventos, e gera código em tempo de execução.

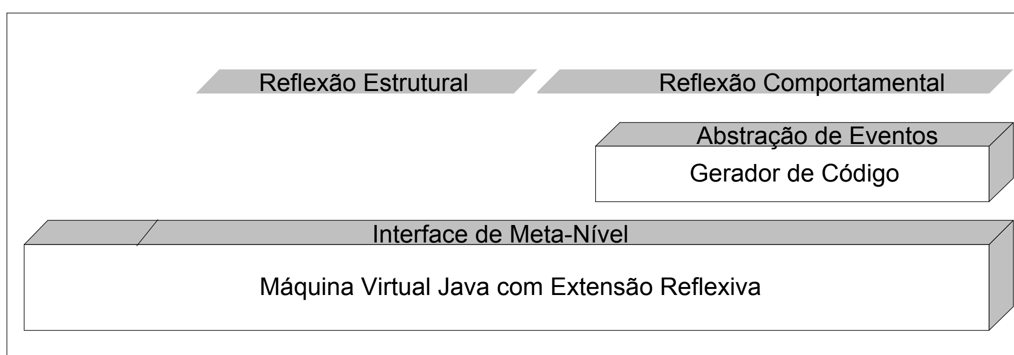


FIGURA 3.11 – Arquitetura do Protocolo MetaJava [GOL 97]

Os meta-objetos acessam a máquina virtual através da *interface* de meta-nível (MLI). Esta contém métodos para reflexão estrutural, os quais são implementados como métodos nativos e ligados dinamicamente pelo interpretador Java. A MLI possui as seguintes funções [KLE 97], sendo que seus métodos são separados em grupos, de acordo com suas funcionalidades:

- manipulação de informações de objetos;
- suporte a modificação de código;

- ligação entre o nível-base e o meta-nível;
- modificação nas estruturas das classes;
- modificação de constantes *pool;*
- suporte a código nativo.

Na *interface* de meta-nível são encontrados também, métodos auxiliares, que podem ser usados, por exemplo, para criar objetos cujos estados são completamente gerenciados pelo meta-objeto associado. Acima da MLI, encontra-se a camada de geração de eventos, constituída por um conjunto de métodos herdados por meta-objetos da classe *MetaObject*, na qual é escrita em Java e utiliza a MLI para reflexão estrutural.

Os seguintes métodos, mostrados na tabela 3.3, devem ser usados nas situações em que um meta-objeto interessa-se pelo comportamento de objeto do nível-base, tendo o meta-objeto que registrar os eventos [BER 99]:

TABELA 3.3 – Métodos de Reflexão Comportamental do MetaJava

Métodos
`void registerEventMethodCall(Object ojb, int notificationTypem, String Methods)` → `registro da reificação de mensagens que chegam`
`void registerEventMethodCallOut(Object obj, String Methods)` → `reificação de mensagens que saem`
`void registerEventFieldAccess(Object obj, String Methods)` → `reificação do acesso às variáveis`
`void registerEventObjectLock(Object obj)` → `reificação de` *`locking`* `do objeto`
`void registerEventObjectCreation(Class c)` → `reificação da criação de objetos`
`void registerEventClassLoading(String classname)` → `reificação do` *`loading`* `da classe`

A criação de uma coleção de métodos nativos residentes em DLLs (*Dynamic Linked Library*) constitui uma das práticas mais comuns para estender a JVM [BER 99], Esta técnica é utilizada pela *Sun MicroSystems* no pacote *java.net* e na biblioteca *unix libnet.so*, permitindo assim, que o protocolo seja integrado a JVM.

Uma descrição completa dos métodos deste protocolo pode ser encontrada em [GOL 97].

3.8 Protocolo Reflexivo OpenJava

O protocolo de reflexão OpenJava foi desenvolvido por Michiaki Tatsubori, na Universidade de Tsukuba, em 1999. Consiste em uma linguagem estendida baseada em Java. As características do protocolo OpenJava são especificadas por um programa de meta-nível em tempo de compilação [TAT 99]. O programa de meta-nível estende o OpenJava através de uma *interface* chamada OpenJava MOP (*MetaObject Protocol*). O compilador OpenJava é formado por três estágios:

a) pré-processador;

b) tradução do código OpenJava para código Java;

c) compilação do código nativo Java.

Este protocolo foi totalmente escrito em Java (JDK 1.1), podendo ser executado em qualquer plataforma que suporte a máquina virtual versão1.1 ou 1.2.

O protocolo de meta-objetos OpenJava consiste numa *interface* para controlar a tradução que acontece no segundo estágio (b), no qual permite especificar de que maneira uma característica estendida do protocolo OpenJava pode ser traduzida para código regular Java.

De acordo com Tatsubori [TAT 99], a característica de extensão do protocolo é fornecida com a incorporação de uma espécie de *plug-in* para o compilador. Este *software* incorporado consiste não somente pelo programa de meta-nível, mas contém também código para suporte durante a execução, provendo as classes usadas pelo programa de nível-base traduzido para Java. O programa de nível-base em OpenJava é primeiramente traduzido para código Java pelo programa de meta-nível sendo dinamicamente ligado com o código de suporte em tempo de execução, conforme demonstra a fig. 3.12.

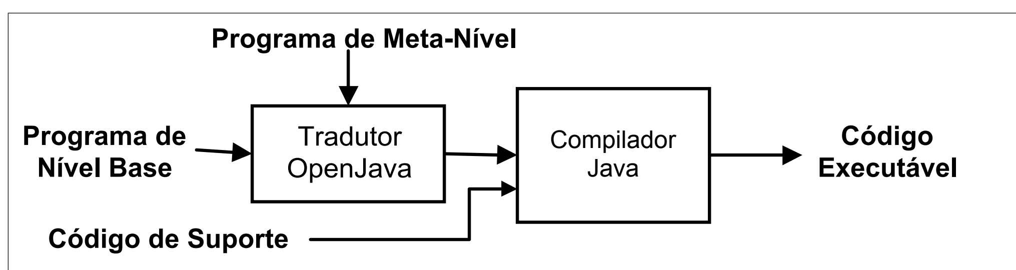


FIGURA 3.12 – Fluxo de informações do compilador OpenJava [TAT 99]

O programa de meta-nível define novos meta-objetos para controlar o processo de tradução de OpenJava para Java. Os meta-objetos são a representação no meta-nível dos objetos do programa de nível-base, sendo que eles realizam o processo de tradução, sendo os detalhes dos meta-objetos especificados pelo OpenJava MOP.

De acordo com Chiba [CHI 99], com a finalidade de evitar degradação de performance foi utilizada uma técnica chamada de reflexão em tempo de compilação. Esta técnica faz com que a maioria das customizações sejam estaticamente aplicadas ao

programa Java em tempo compilação, fazendo com que o código compilado seja executado pela JVM com um número de *overheads* significativamente limitado.

Um exemplo da utilização do protocolo é mostrado através de um pseudo-código [CHI 99] na fig. 3.13, no qual o comportamento de acesso a campos de uma classe é reimplementado, de modo que uma mensagem seja impressa sempre que o campo é lido nos objetos da classe por qualquer método.

```
1   class VerboseClass extends Class {
2       Object fieldRead(Object o, Field f){
3       System.out.println(f.getName() + " has been read.");
4       return super.fieldRead(o, f);
5       }
6   }
```

FIGURA 3.13 – Reimplementação de métodos [CHI 99]

De acordo com a fig. 3.13, se um campo é lido em objetos cuja sua meta-classe seja *VerboseClass*, então o método *fieldRead( )* é chamado, sendo mostrada uma mensagem para uma depuração, por exemplo. [CHI 99].

Este protocolo implementa uma classe chamada *openjava.mop.OJClass* para meta-objetos, além de classes associadas, tais como *OJMethod*, que são uma reimplementações das classes *java.lang.Class* e *Method*, acrescentadas a seus nomes o sufixo "OJ". Estas classes provêem as habilidades de introspecção e customização da linguagem, sendo a última, implementada com a reflexão em tempo de compilação em cooperação com o compilador OpenJava. A classe *OJClass*, fornece todos os métodos contidos na classe *Class*.

3.8.1 Processo de Tradução

Uma extensão é definida, de acordo com Tatsubori [TAT 99b], como uma meta-classe, e uma classe corresponde a uma instância de uma meta-classe. Em OpenJava, a meta-classe default é provida com a classe *OJClass*, não sendo esta implementada para realizar o processo de tradução. O meta-programa deve definir uma sub-classe de *OJClass*, com a finalidade de implementar as extensões almejadas e especificar a relação desta com as classes apropriadas. Logo após, o sistema traduzirá somente as partes do programa, no qual os objetos instanciados da classe estejam relacionados com esta meta-classe. A fig. 3.14 mostra um pequeno exemplo de um programa de nível-base escrito em OpenJava.

```
1   public class Hello instantiates Verbose {
2       public String say() {
3       return "Hello World.";
4       }
5   }
```

FIGURA 3.14 – Exemplo de programa de nível-base – *Hello.oj* - [TAT 99b]

Ressalta-se aqui, a notação: *class C instantiates M*. A classe *C* é ligada com a meta-classe *M*. Isto significa que a tradução dos objetos do tipo *C* será realizada de acordo com as definições da meta-classe *M*. Analisando-se o exemplo da fig. 3.14, nota-

se que a classe *Verbose* será definida para mudar o comportamento na chamada do método, com uma finalidade específica, como por exemplo, realizar uma depuração.

Sob forma de exemplificar a utilização do protocolo, será apresentado um exemplo, adaptado de [TAT 99b]. A fig. 3.15 apresenta a listagem do programa de nível-base, cuja classe *Seqüência* está ligada à meta-classe *MetaSequencia*.

```
1   public class Sequencia instantiates MetaSequencia {
2       public static void main( String[] args ) {
3           Imp_Msg();
4       }
5       static void Imp_Msg() {
6           System.out.println( "Mensagem impressa na tela." );
7       }
8   }
```

FIGURA 3.15 – Programa de nível-base – *Sequencia.oj*

A implementação da meta-classe *MetaSequencia* tem por finalidade traduzir o código da classe de nível-base, a qual passará a imprimir a relação dos métodos chamados quando o programa for executado. A fig. 3.16 lista a implementação da meta-classe.

```
1   import openjava.mop.*;
2   import openjava.ptree.*;
3   public class MetaSequencia instantiates Metaclass extends OJClass
4       {
5       public void translateDefinition() throws MOPException {
6           OJMethod[] methods = getDeclaredMethods();
7           for (int i = 0; i < methods.length; ++i) {
8               Statement printer = makeStatement(
9                   "System.out.println( \"" + methods[i]
10                      + " foi chamado.\" );" );
11              methods[i].getBody().insertElementAt( printer, 0 );
12          }
13      }
14  }
```

FIGURA 3.16 – Programa de meta-nível – *MetaSequencia.oj*

A tradução, na prática, do código da classe *Sequencia*, será realizada pelo objeto da meta-classe *MetaSequencia*. O objetivo desta implementação, como descrito, é o de apresentar na tela, a seqüência dos métodos chamados durante a execução do programa. A meta-classe traduzirá o código da classe de nível-base, adicionado ao corpo do método *Imp_Msg( )* da classe *Sequencia* uma função na qual seja possível listar os métodos chamados. A listagem com o código traduzido para Java é apresentado na fig. 3.17.

```
 1  /*
 2   * This code was generated by ojc.
 3   */
 4  public class Sequencia
 5  {
 6      public static void main( String[] args )
 7      {
 8          System.out.println( "public static void "
 9              + "Sequencia.main(java.lang.String[]) foi chamado." );
10          Imp_Msg();
11      }
12      static void Imp_Msg()
13      {
14          System.out.println( "static void Sequencia.Imp_Msg() foi
15              chamado." );
16          System.out.println( "Mensagem impressa na tela." );
17      }
18  }
```

```
 1  public static void Sequencia.main(java.lang.String[]) foi chamado.
 2  static void Sequencia.Imp_Msg() foi chamado.
 3  Mensagem impressa na tela.
```

FIGURA 3.17 – Programa de nível-base traduzido – *Sequencia.java*

Para efetivar o processo de tradução, deve-se primeiramente compilar o programa de meta-nível, seguido do programa de nível-base. O compilador do OpenJava pode ser executado usando-se a JVM, como segue: *C:>java openjava.ojc.Main*

3.8.2 A API do Protocolo

A meta-classe *OJClass* constitui uma das partes mais importantes da API do OpenJava, provendo métodos para acessar as informações das classes. As classes *OJField*, *OJMethod* e *OJConsructor* representam os campos, métodos e construtores dos meta-objetos, estando disponíveis através dos métodos da *OJClass* [TAT 99b].

Na tabela 3.4 são listados os principais métodos da classe *OJClass*.

TABELA 3.4 – Principais métodos da classe *OJClass*

Métodos
`public OJClass getSuperclass()` → retorna o objeto OJClass que representa a superclasse da classe
`public OJClass[] getInterfaces()` → retorna um vetor de objetos OJClass representando a interface da classe
`public OJConstructor[] getDeclaredConstructors()` → retorna um vetor com todos os construtores da classe, excluindo os herdados
`public OJClass[] getDeclaredClasses()` → retorna um vetor com as classes declaradas nesta.
`public OJField[] getDeclaredFields()` → retorna um vetor com os campos declarados na classe, excluindo os herdados

`public OJMethod[] getDeclaredMethods()` → retorna um vetor com todos os métodos declarados na classe, excluindo os herdados
`protected OJMethod addMethod( OJMethod method )` → gera um objeto OJMethod com a mesma assinatura do objeto OJMethod dado, adicionando-o aos métodos da classe.
`protected OJMethod removeMethod( Signature signature )` → remove um método com a assinatura dada da classe.
`protected OJMethod replaceMethod( Signature signature, OJMethod replacement )` → troca um método com a assinatura correspondente pelo método fornecido
`public void translateDeclaration( Environment env )` → sobrescreve no processo de tradução a declaração da classe
`public Expression expandAllocation( AllocationExpression expr, Environment env )` → retorna o objeto OJMethod gerado
`public Expression expandArrayAllocation(AllocationExpression expr, Environment env )` → retorna o objeto OJMethod gerado
`public Expression expandVariableDeclaration( VariableDeclaration stmt, Environment env )` → retorna o objeto OJMethod gerado
`public Expression expandFieldRead( FieldAccess expr, Environment env )` → retorna o objeto OJMethod gerado
`public Expression expandFieldWrite( FieldAccess expr, Environment env )` → gera um objeto OJMethod com a mesma assinatura do objeto OJMethod fornecido, adicionando-o a classe
`public Expression expandMethodCall( MethodCall expr, Environment env )` → gera um objeto OJMethod com a mesma assinatura do objeto OJMethod fornecido, adicionando-o a classe
`public Expression expandExpression( Expression expr, Environment env )` → gera um objeto OJMethod com a mesma assinatura do objeto OJMethod fornecido, adicionando-o a classe

Maiores detalhes sobre a utilização do protocolo, bem como uma completa relação de seus métodos e classes, podem ser encontradas em Tatsubori [TAT 99b] e Chiba [CHI 99].

3.9 Protocolo Reflexivo Guaraná

O protocolo Guaraná consiste numa arquitetura reflexiva, na qual seu protocolo de meta-objetos permite a reutilização do código do meta-nível através da composição de meta-objetos [OLI 98b]. O Guaraná utiliza um protocolo de meta-objetos em tempo de execução, permitindo a composição dinâmica de meta-objetos, a qual possibilita uma maneira mais simples de composição destes objetos, tornando possível sua reconfiguração dinâmica.

O protocolo Guaraná foi implementado através da modificação da *Kaffe OpenVM*, uma implementação de domínio público da especificação da JVM padrão [OLI 98c]. Segundo Oliva [OLI 98d], a maior parte do protocolo é codificada em Java, mas sua máquina virtual sofreu localizadas modificações, a fim de prover operações de interceptação, materialização e criação de operações. Entretanto, a linguagem de programação Java não foi modificada: qualquer programa em Java, compilado por qualquer compilador Java, poderá ser executado nesta implementação, sendo ainda possível, estender suas capacidades através do uso de reflexão. As alterações no interpretador permitem [OLI 98e]:

- interceptação de operações, como invocação de métodos;
- leitura e escrita em variáveis, bem como em elementos de *arrays*
- criação de objetos e *arrays*;
- entrada/saída de monitores

Na versão 2, a API de reflexão de Java permite a um objeto realizar somente as operações concedidas a ele, sendo estas diretamente no código fonte, isto é, o controle de acesso é baseado em permissões da classe. O protocolo Guaraná incrementa esta característica [OLI 98c], introduzindo mecanismos de interceptação que não estão presentes em Java, além de mecanismo de segurança por objeto (contrário ao de classes), resultando que meta-objetos podem obter privilégio de acesso para objetos que eles controlam.

Sob forma de padronizar os termos que referenciam os elementos presentes no protocolo Guaraná e, consequentemente na Ferramenta desenvolvida, utilizar-se-á o prefixo *para* a fim de distinguir tais elementos. Assim, em uma discussão sobre dois objetos, aquele que sofre reflexão será rotulado de **para-objeto**, em oposição ao **meta-objeto** que exerce reflexão. Da mesma forma, **para-nível** torna-se o nível que sofre reflexão e está subordinado a um **meta-nível**. Deste modo, o domínio da aplicação é constituído por um único para-nível: o nível base. Já o meta-domínio é constituído de uma pilha de níveis reflexivos que podem assumir ambos papéis de meta-nível e para-nível, ou seja, exercer e sofrer reflexão. Estende-se, igualmente esta nomenclatura, às classes que sofrem reflexões, denominadas então **para-classes**.

3.9.1 Meta-Objetos do Protocolo

De acordo com Oliva [OLI 98], cada objeto pode estar diretamente associado com zero ou um meta-objeto, chamado de meta-objeto primário. Seu papel é observar todas as operações endereçadas ao seu objeto associado, bem como seus resultados. Tais

funções são garantidas pelos mecanismos de interceptação e reificação implementados no *kernel* do protocolo. É possível também, que a classe seja associada a um meta-objeto primário, no qual observará todas as operações referentes à classe associada e não com relação às suas instâncias, resultando disto que, os meta-objetos das classes e suas instâncias são independentes. Desta maneira, não serão interceptadas operações atribuídas a instâncias que não estejam ligadas a este meta-objeto [OLI 98b].

Diferentemente de outros protocolos, o Guaraná não permite que objetos do nível-base se refiram a seus meta-objetos, a finalidade de obter referências destes. O acoplamento entre objetos do nível-base (para-objetos) e meta-objetos é realizado pelas operações de interceptação e reificação e pelo mecanismo de ligação dinâmica. O método *reconfigure* (detalhado a seguir) do *kernel* é responsável pelo acoplamento entre objetos e seus meta-objetos.

Três possíveis possibilidades são retornadas pelo meta-objeto primário ao *kernel*, após suas operações de inspeção e reflexão sobre seus conteúdos [OLI 98b]:

a) um resultado: considerado pelo *kernel* como se fosse produzido pela execução da operação atual;
b) uma operação de substituição: o *kernel* irá repassar ao para-objeto, desconsiderando a original;
c) nenhuma das anteriores: o *kernel* devolverá a operação original para o para-objeto da aplicação.

Nas alternativas "b" e "c", onde o meta-objeto não fornece resultado, este poderá sinalizar ao *kernel* que pretende inspecionar ou alterar o resultado da operação. Diante desta situação, após a realização da operação, o *kernel* reificará o resultado e o apresentará ao meta-objeto primário, sendo possível neste ponto, para o meta-objeto primário, realizar qualquer operação apropriada. Cabe ressaltar que, o *kernel* somente aceitará este resultado modificado se o meta-objeto tiver indicado que ele poderia modificá-lo.

3.9.2 Composers

A associação de múltiplos meta-objetos a objetos da aplicação é perfeitamente possível neste protocolo [OLI 98b]. Devido a isto, ocorre o problema da organização do fluxo da interceptação de operações através dos meta-objetos. Para contornar tal situação, uma forma especializada de meta-objeto, denominado *composer*, é responsável pela aplicação de políticas que dão estrutura e ordem no fluxo das operações delegadas aos meta-objetos [OLI 98].

O agrupamento de objetos que geralmente trabalham juntos, constitui uma das aplicações dos *composers*, sendo possível a formação de uma estrutura recursiva, potencialmente infinita e hierárquica de meta-objetos, também conhecida como torre de meta-objetos [MAE 87]. Oliva [OLI 98] descreve que o *composer* implementado no protocolo é o seqüencial, no qual organiza os meta-objetos seqüencialmente, como uma pilha.

O *kernel* do protocolo guaraná constitui a base de sua arquitetura, tendo como funções básicas a realização das seguintes tarefas [OLI 98]:

- operações de interceptações e reificações;
- ligação dinâmica e invocação para objetos do meta-nível;
- manutenção da meta-informação estrutural;

3.9.3 Estrutura do MOP

Na fig. 3.18 está representada a estrutura do MOP do Guaraná. Na descrição que segue, será enfatizado o ponto de vista do usuário, ao invés do ponto de vista do projetista do protocolo, por uma razão de simplicidade. A entidade central do MOP é a classe *MetaObject*. Esta classe constitui-se no ponto de partida para a construção e definição de entidades de meta-nível. De acordo com Senra [SEN 2001], pode-se dividir esta figura em quatro quadrantes, a fim de simplificar o entendimento e análise dos relacionamentos entre a classe *MetaObject* e as demais meta-entidades. No primeiro quadrante, esta classe possui um relacionamento com *Operation* e *Result*, modelando desta forma, o aspecto de reificação do protocolo. As instâncias de *MetaObject* processam operações e seus resultados, as quais são convertidas em meta-informação e encapsuladas em entidades do meta-nível, sendo respectivamente instâncias de *Operation* e *Result*.

Os relacionamentos com *OperationFactory*, *OperationFactoryFilter* e *HashWrapper*, apresentados no segundo quadrante, representam, respectivamente:

i) a classe responsável pelo controle e espectro de operações que um meta-objeto pode aplicar sobre o seu para-objeto;

ii) um filtro restritivo para uma fábrica de operações mais permissiva;

iii) a classe responsável pela blindagem no processo de reificação sobre estruturas de dados presentes em uma dada meta-configuração, que referenciam implicitamente o para-objeto vinculado a uma mesma meta-configuração [SEN 2001], evitando que seja gerado um processo infinito de reificação.

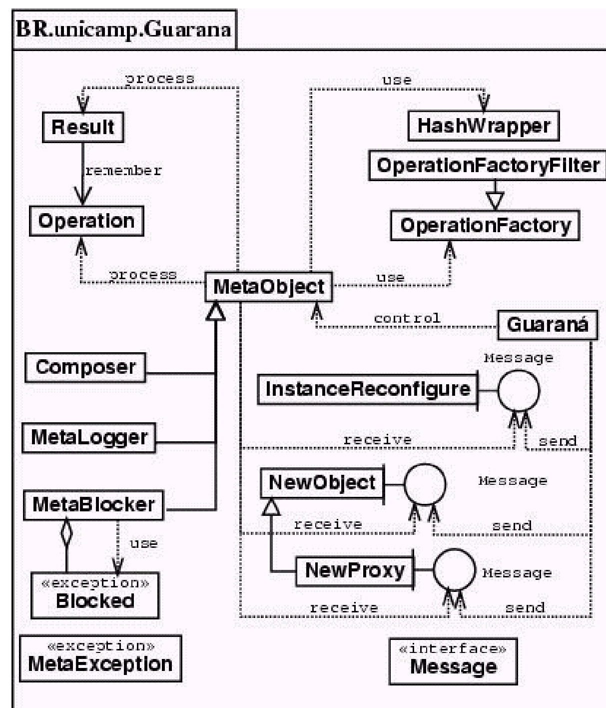


FIGURA 3.18 – Diagrama de Classe do MOP [SEN 2001]

O primeiro nível da hierarquia de especialização de *MetaObject* é visualizado no terceiro quadrante, utilizada para reificar o comportamento do meta-nível: *MetaBlocker, Composer* e *MetaLogger.* A primeira classe compreende meta-entidades que são vinculadas por *default* a qualquer para-objeto, desde o instante da criação deste até sua vinculação a uma meta-entidade válida, oferecendo aos para-objetos uma espécie de blindagem contra vinculações não autorizadas. Estas autorizações ficam sob responsabilidade do protocolo de validação [SEN 2001]. Instâncias da segunda classe (*Composer*) atuam como meta-entidades desempenhando o papel de redistribuidoras de meta-informação para outros meta-objetos. Partindo-se do princípio que tais entidades são também meta-objetos, torna-se possível a construção de hierarquias de meta-objetos [OLI 98], através do encadeamento de C*omposers* com diferentes semânticas de interconexões. Já a classe *MetaLogger* atua como exemplo de meta-objeto simples e desempenha papel de depuração sobre vinculações entre meta-nível e para-nível.

No último quadrante encontra-se a classe *Guarana*, que representa a *interface* para o núcleo do MOP, não permitindo desta maneira a criação de instâncias. Através desta *interface* manifestam-se os aspectos de vinculação e execução do MOP. Ressalta-se que, esta classe atua como juiz de requisições de vinculação, assegurando o que é chamado de privilégio de anterioridade, ou seja, meta-objetos já vinculados tem preferência em detrimento a meta-objetos suplicantes [SEN 2001]. Neste quadrante está também a *interface Message*. Através da troca de mensagens, torna-se possível o processo de comunicação entre meta-objetos, cujas mensagens são objetos instanciados de classes que implementam esta *interface* [SEN 2001].

3.9.4 Dinâmica do MOP

A dinâmica do MOP do Guaraná é dividida em quatros processos [OLI 98]:

- Vinculação: corresponde à primeira associação de uma meta-configuração a um para-objeto, tornando-o reflexivo;
- Atuação: compreende qualquer atividade de uma meta-configuração sobre o para-objeto ao qual esta meta-configuração está vinculada;
- Propagação: diz respeito a toda vinculação originada automaticamente por uma meta-configuração "x", objetivando instalar uma meta-configuração própria sobre um para-objeto recém criado por outro para-objeto vinculado a esta meta-configuração "x" existente.
- Reconfiguração: constitui uma modificação numa meta-configuração existente vinculada a um para-objeto.

Enfatiza-se que, a seqüência em que tais processos ocorrem, somente é determinada em tempo de execução, com exceção do processo de vinculação, o qual é sempre o primeiro a ocorrer. Tais processos compreendem interações entre meta-nível e para-nível. Um quinto processo, denominado *comunicação*, difere dos demais devido a sua ocorrência se fazer presente somente no meta-nível. De acordo com Senra [SEN 2001], este processo caracteriza-se por ser uma troca síncrona de objetos de meta-nível (instâncias de sub-classes de *Message*) entre meta-objetos.

3.9.4.1 Vinculação

Conforme explanado anteriormente, o processo de *vinculação* é o único com ordem de ocorrência definida, pois é sempre o primeiro a ocorrer, definindo o meta-objeto primário do para-objeto alvo. Somente existe vinculação direta entre o meta-objeto primário e seu para-objeto alvo. Por ocorrência de uma meta-configuração, onde podem ser encontrados diversos meta-objetos, diz-se que o para-objeto está vinculado a esta meta-configuração por intermédio de seu meta-objeto primário. Neste caso, este meta-objeto primário será instância de uma sub-classe de *Composer* [OLI 98].

O método responsável pela vinculação é *reconfigure (paraObj, oldMetaObj, newMetaObj)* da classe *Guarana*. Aqui, *paraObj* representa uma referência para o para-objeto alvo, *oldMetaObj* será nulo e *newMetaObj* corresponde uma referência para o meta-objeto primário da meta-configuração a ser vinculada. Ressalta-se que este pode ser o único meta-objeto de uma meta-configuração, no caso de uma meta-configuração unitária.

O núcleo do MOP verifica, antes que a vinculação seja feita, se a classe da qual o para-objeto foi instanciado corresponde a uma classe reflexiva. Em caso afirmativo, uma mensagem *InstanceReconfigure* é enviada ao meta-objeto primário da classe em questão, notificando-o da intenção de associação. Tal mensagem constitui-se de um objeto de meta-nível, o qual carrega os parâmetros *paraObj* e *newMetaObj* da requisição de vinculação.

Este mecanismo possui um objetivo principal: consultar a meta-configuração vinculada a esta classe antes da efetivar a vinculação, podendo a meta-configuração aceitar, recusar ou substituir a meta-configuração a ser vinculada. Toda vez que uma vinculação é estabelecida, o núcleo do MOP notifica o meta-objeto envolvido utilizando a primitiva *initialize(OperationFactory, paraObj)*. Desta forma, é repassado ao meta-objeto uma referência para as operações válidas (fábrica de operações) sobre o para-objeto que foi vinculado. Quando da desvinculação entre para e meta-objeto, o último recebe a notificação através da primitiva *release(paraObj)*.

3.9.4.2 Atuação

Segundo Oliva [OLI 98c], um para-objeto sofre e exerce operações na terminologia deste MOP. Operações de leitura ou escrita em atributos de um para-objetos constituem operações sobre estes, assim como invocações de métodos pertencentes a estes mesmos para-objetos. Desta forma, caracteriza-se como *atuação* o processo de manipulação de operações endereçadas a para-objetos reflexivos por parte da sua meta-configuração associada. Na fig. 3.19 é apresentado uma interação típica entre dois objetos não reflexivos.

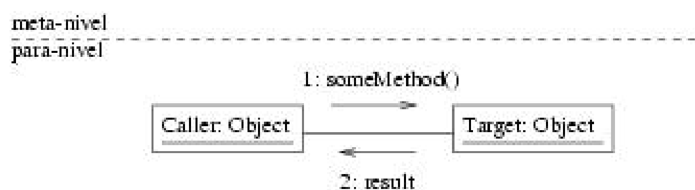


FIGURA 3.19 – Interação entre dois objetos do para-nível [OLI 98d]

Observa-se na figura acima que o objeto *Target* recebe a mensagem *someMethod()* presente em sua *interface* e, após o processamento, devolve o resultado *result* para o objeto *Caller*. Já na fig. 3.20, observa-se a interação com o objeto reflexivo *Target*, o qual está vinculado ao meta-objeto *MO*.

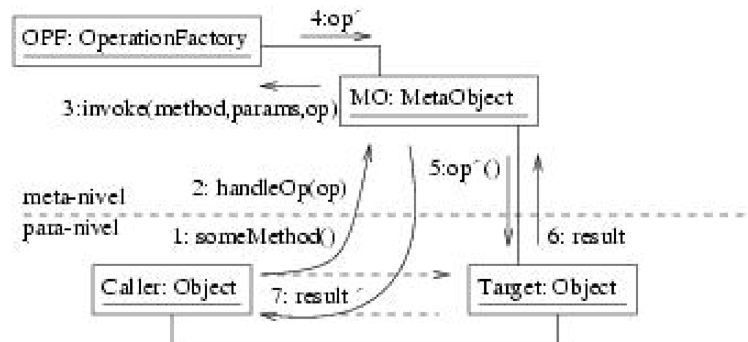


FIGURA 3.20 – Interação com um objeto reflexivo [OLI 98d]

Neste diagrama, não estão presentes algumas das entidades envolvidas presentes no núcleo do MOP, devido ao fato de estar modelando-se o nível em que atua o programador de meta-nível, e não o programador do próprio MOP. Na fig. 3.20 percebe-se que a mensagem *someMethod()* enviada para *Target* (passo 1) foi interceptada e reificada pelo núcleo do MOP no objeto de meta-nível *op*, o qual é uma instância de *Operation*. No segundo passo, o MOP invoca o método *handleOperation()* do meta-objeto *MO* vinculado à *Target*, notificando-o da intenção da execução da operação *op* sobre o para-objeto *Target*.

Uma entre três possíveis alternativas deve ser tomada por *MO* após a notificação:

1. A operação *op* é devolvida ao para-objeto *Target* (passo 5), pois *MO* decide não intervir;
2. Um resultado é devolvido diretamente ao objeto *Caller* por *MO*, sem a participação de *Target* (passo 7);
3. Uma operação de substituição para *op* é criada antes do repasse para *Target*;

Quando da criação de operações sobre para-objetos (alternativa 3), é necessário utilizar a fábrica de operações (OPF), a qual constitui as operações autorizadas pelo núcleo do MOP para o para-objeto em questão. Desta forma, no passo 3, *MO* requisita a nova operação *op'* a ser repassada para *Target* em substituição a *op* solicitada por *Caller*. Contudo, torna-se necessária antes do repasse da referida operação, uma sinalização por parte de *MO* ao núcleo do MOP, indicando sua intenção de ignorar, observar ou modificar o resultado produzido por *Target* após o repasse.

O processo de atuação de um meta-objeto é codificado no tratamento das primitivas *handle(Operation, paraObj)* e *handle(Result, paraObj)* [OLI 98], correspondendo aos passos 2 e 6 da figura acima.

3.9.4.3 Propagação

Trata das extensões de meta-configurações quando da criação de para-objetos. Existem dois contextos que podem desejar configurar reflexivamente um para-objeto quando este é criado: contexto *comportamental* e *estrutural* [SEN 2001]. O primeiro é representado pelo para-objeto criador, enquanto que o segundo é representado pela para-classe a partir da qual este novo para-objeto foi instanciado.

O primeiro contexto possui uma prioridade mais elevada quando da configuração do para-objeto criado. Quando um para-objeto é criado a partir de um para-objeto reflexivo, o meta-objeto deste é notificado sempre quando da criação de um novo para-objeto, através da primitiva *configure(newParaObj, currentParaObj)* presente em sua *interface*. O meta-objeto primário pode definir a composição inicial da meta-configuração do para-objeto recém criado. Suas alternativas de decisão são: propagar-se para a nova meta-configuração (ou seja, sua instância vincular-se-á ao novo para-objeto criado), delegar a decisão para outros eventuais meta-objetos presentes em sua meta-configuração ou então, não contribuir.

Já o contexto estrutural é detentor de uma prioridade mais baixa. Sempre que a para-classe do para-objeto criado é reflexiva, o meta-objeto primário desta para-classe reflexiva é notificado através do recebimento da mensagem *NewObject*. Isto ocorre quando dá criação de objetos de para-classes reflexivas. Quanto às alternativas de decisão sobre instalação de meta-configuração, este contexto apresenta as mesmas presentes no contexto comportamental.

3.9.4.4 Reconfiguração

O método *Reconfigure*, utilizado no processo de *vinculação*, é também utilizado no processo de *reconfiguração.* Para tanto, basta então invocar *Reconfigure(paraObj, oldMetaObj, newMetaObj)* da classe Guaraná.

O primeiro parâmetro, através de uma referência para o para-objeto alvo, especifica qual meta-configuração deve sofrer alteração. O segundo parâmetro identifica qual meta-configuração alvo sofrerá modificação. O terceiro e último parâmetro, identifica a natureza da alteração. De acordo com Senra [SEN 2001], a primitiva *Reconfigure* possui caráter informativo, cabendo ao meta-objeto alvo analisar seus parâmetros e decidir pela alteração solicitada.

A seguir são apresentados os cenários típicos de pedidos de reconfiguração. Assumindo-se que um meta-objeto *MO* vinculado ao para-objeto *PO* recebe:

- reconfigure(PO,MO,MO’): neste caso, o próprio meta-objeto primário MO é a porção de meta-configuração a ser alterada. Esta reconfiguração consiste em tornar MO’ o novo meta-objeto primário de PO. Ressalta-se que, se MO e MO’ constituem objetos distintos e pertencentes a meta-configurações unitárias, esta é uma típica requisição de substituição de meta-configuração. Caso contrário, se MO’ for a raiz de uma hiearquia de meta-objetos, no qual MO faça parte, configura-se um pedido de aumento de meta-configuração.
- reconfigure(PO,MO,null): aqui é identificada a meta-configuração alvo de alteração, ou seja, MO. Como o parâmetro *newMetaObj* é nulo,

verifica-se um pedido de remoção de meta-configuração. Se este pedido for aceito por MO, PO deixa de ser reflexivo.

- reconfigure(PO,null,MO'): como o parâmetro *oldMetaObj* é nulo, este pode ser interpretado como uma máscara para qualquer meta-configuração a ser instalada. Se PO não for reflexivo, este pedido é encarado como um processo de vinculação ao invés de reconfiguração. Diz-se então que, vinculação é um caso particular de reconfiguração.

3.9.4.5 Comunicação

A troca de mensagens entre meta-objetos é possível através do processo de *comunicação*. Estas mensagens constituem objetos de meta-nível, os quais são instanciados a partir de classes que implementem a *interface Message*. Entretanto, tais mensagens não podem ser enviadas diretamente a meta-objetos, mas sim através de referências de para-objetos, cujos efetivos receptores serão seus meta-objetos vinculados [SEN 2001]. De acordo com Oliva [OLI 98b], caso um para-objeto não seja reflexivo, tal mensagem não será entregue, não acarretando qualquer prejuízo ao sistema.

O método responsável pelo envio de mensagens é o *broadcast(paraObj, message)* da classe *Guarana*. Cabe citar que algumas mensagens são automaticamente enviadas pelo núcleo do Guaraná, como por exemplo *InstanceReconfigure* e *NewObject*, ambas explicadas anteriormente. A notificação da chegada de mensagens em cada meta-objeto dá-se pela primitiva *handle(Message, paraObj)*.

3.9.5 Gerenciamento de Meta-Configuração

Assim que uma aplicação é inicializada, todos os objetos possuem uma meta-configuração vazia, isto é, nenhum objeto está passível de reflexão. A aplicação é que pode criar objetos e meta-objetos e então associá-los [OLI 98]. Esta operação é realizada através do *kernel* do protocolo, utilizando o método *reconfigure( )*, o qual efetiva a meta-configuração de um objeto com um determinado meta-objeto, conforme explicado anteriormente. A fig. 3.21 exemplifica a meta-configuração de O_1: um *composer*, chamado de seu meta-objeto primário, delega para outros três meta-objetos, dos quais um é *composer* de si próprio, e delega para outros dois meta-objetos. O_2 não está associado com nenhum meta-objeto, resultando que suas operações não serão interceptadas, ou seja, ele não é reflexivo. O_3 compartilha MO_4 com O_1. MO_1 é um meta-objeto reflexivo, desde que ele possua sua própria (meta)meta-configuração.

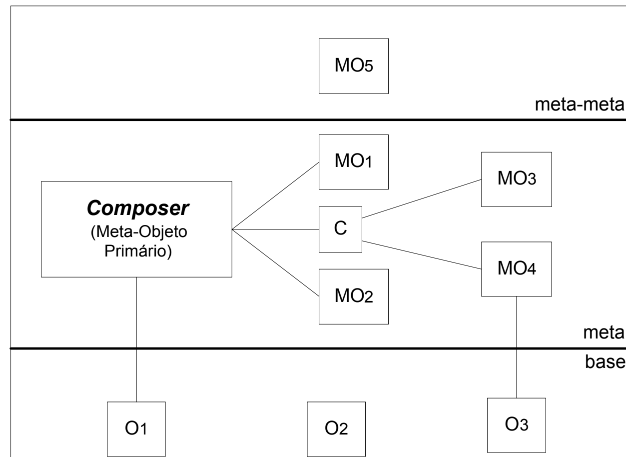


FIGURA 3.21 – Meta-configuração [OLI 98]

3.9.6 Bibliotecas de Meta-Objetos

O protocolo de meta-nível do Guaraná foi projetado e implementado sob forma de tornar possível o desenvolvimento de bibliotecas de meta-objetos, as quais podem implementar diferentes comportamentos no meta-nível, podendo compor com isto, uma complexa meta-configuração [OLI 98b]. Um exemplo desta utilização é encontrada em [OLI 98e], uma biblioteca de meta-objetos para sistemas distribuídos, a qual provê meta-objetos para realizarem persistência, distribuição, replicação, atomicidade e migração.

3.9.7 Principais Classes e Métodos do Protocolo Guaraná

O protocolo Guaraná foi desenvolvido através de um pacote denominado *BR.unicamp.guarana.* Os principais métodos da classe *Guarana*, a qual representa o *kernel* do protocolo, são apresentados na tabela 3.5. Uma completa referência sobre a hierarquia das classes pode ser encontrada em [OLI 98] [OLI 98b] [OLI 98c] [OLI 98d] [OLI 98e].

TABELA 3.5 – Métodos da Classe Guarana

Métodos
`public static void broadcast(Message msg, Object obj)` → solicita ao MetaObject associado para manipular a mensagem. Caso não possua associação, nada será feito.
`public static Result perform(Operation operation)` → quando uma operação é interceptada, esta é reificada para o MetaObject através deste método
`public static void reconfigure(Object obj, MetaObject oldMO, MetaObject newMO)` → utilizado para reconfiguração, ou seja, substitui o *oldMO* pelo *newMO* na meta-configuração de dado objeto.
`public static native Object makeProxy(Class cls, MetaObject MO)` → cria um objeto não inicializado, realizando um *broadcast* da mensagem *NewObject* para o meta-objeto da classe.
`public static native hashCode(Object obj)` → retorna o endereço de onde o objeto está armazenado.

Conforme descrito anteriormente, os meta-objetos são associados aos para-objetos durante a execução da aplicação, ou seja, de forma dinâmica. As capacidades reflexivas deste protocolo estão baseadas nas interceptações pelos meta-objetos sobre interações entre os para-objetos. Sob forma de exemplificar os conceitos anteriores, bem como de alguns dos mecanismos implementados pelo Guaraná, utilizar-se-á um meta-objeto simples, chamado *MetaLogger* [OLI 98b], o qual escreve no console um *log* descritivo das operações realizadas. Este é ativado quando da interceptação ou de outros tipos de interação do meta-nível.

A fig. 3.22, mostra um exemplo de como realizar a associação de um meta-objeto com um objeto. A primeira parte apresenta a codificação no meta-nível e, a segunda parte, apresenta a saída gerada pela classe *MetaLogger*.

```
1    public class ConfBasic {
2        int i =3;
3        synchronized int meth(int j) {
4            return i + j;
5        }
6        public static void main(String[ ] argv) {
7           ConfBasic o = new ConfBasic();
8           BR.unicamp.Guarana.MetaObject mo = new MetaLogger();
9           BR.unicamp.Guarana.Guarana.reconfigure(o, null, mo);
10          o.i = o.meth(1);
11       }
12   }
```

```
1    Initialize: ConfBasic@10de80
2    Operation: ConfBasic@10de80.ConfBasic.meth(int 1)
3    Operation: ConfBasic@10de80.<monitor enter>
4    Result: return null
5    Operation: ConfBasic@10de80.ConfBasic.i
6    Result: return 3
7    Operation: ConfBasic@10de80.<monitor exit>
8    Result: return null
9    Result: return 4
10   Operation: ConfBasic@10de80.ConfBasic.i=4
11   Result: return null
```

FIGURA 3.22 – Exemplo de Reconfiguração [OLI 98]

São criadas instâncias das classes *ConfBasic* na linha {7} e *MetaObject* na linha {8}, sendo requisitado ao *kernel* do referido protocolo, substituir o meta-objeto nulo pelo então criado, na meta-configuração do objeto {9}. A reconfiguração do mesmo é mostrada na linha <1>. A seqüência de caracteres que é produzida na linha <1>, representa a associação do objeto de nível-base com seu meta-objeto, na qual o nome do objeto é apresentado antes do sinal "@", seguido do seu endereço de memória em notação hexadecimal. Este nome é referenciado como *object-id*.

Após o processo de reconfiguração, o método *meth* é invocado {10}, sendo esta operação interceptada pelo *kernel* do protocolo e repassada à classe *MetaLogger*, a qual imprimirá o nome do objeto, o nome do método e a lista de argumentos <2>. De acordo com o método *meth*, ele adiciona o valor do argumento *j* {4} com o valor do campo *i*, sendo que o objeto *MetaLogger* recebe tal operação também, imprimindo o nome do objeto, bem como o nome do campo <5>, além do resultado da operação <6>. Por

último, o resultado do método é associado com o campo *i* do objeto *o* na linha {10}. Segundo Oliva [OLI 98], observa-se que devido ao método *meth* ser sincronizado, operações com monitores são executadas, como percebe-se nas linhas <3>, <4> e <7>.

3.10 Comparação entre os protocolos de reflexão

A aplicação da reflexão computacional no teste de sistemas consiste numa técnica bastante flexível, pois permite que seja realizada a análise da aplicação de forma dinâmica, sem a necessidade de instrumentação do seu código fonte. Isto é feito através da interceptação na computação da aplicação, com o objetivo de monitorar objetos determinados, os quais podem ser selecionados pelo usuário através de uma ferramenta de teste, em tempo de execução.

De acordo com [HER 99], estão sendo desenvolvidas implementações de técnicas de teste de software OO com o uso de reflexão computacional para a avaliação de invariantes associadas a classes, bem como pré e pós condições associadas aos métodos. Através da utilização de protocolos de reflexão que suportem o modelo de reflexão comportamental, ou seja, que permitam a interceptação de mensagens, estas técnicas podem ser utilizadas para verificar a integridade dos objetos, consultar valores de atributos de objetos, analisar se os estados dos mesmos são válidos, de acordo com pré e pós-condições especificadas pelo usuário em tempo de execução.

3.10.1 Análise dos Protocolos de Reflexão

Os protocolos de reflexão computacionais estudados apresentam significativas diferenças entre si. Estas são referentes a aspectos como *interface*, modelo e estilo de reflexão, além do modo pelo qual o meta-nível é ativado pelo protocolo. Durante a análise realizada, buscou-se descobrir como e quais informações poderiam ser obtidas do programa de nível-base, bem como do próprio programa de meta-nível, em tempo de execução. O enfoque maior foi dado em relação a possível utilização do protocolo como parte integrante de uma ferramenta de teste para programas escritos em linguagem Java. Para tal seria necessário que o protocolo oferecesse suporte à reflexão comportamental, pois esta permite que se possa modificar o comportamento dos objetos e a chamada de seus métodos, possibilitando uma análise aprofundada dos objetos em teste.

O primeiro protocolo analisado, CoreJava, apresenta novas classes introduzidas no ambiente JDK versão 1.1, provendo uma API segura, dando suporte principalmente ao processo de introspecção na máquina virtual, na qual é possível obter informações sobre classes e objetos da aplicação em execução, instanciar novos objetos, acessar e modificar campos de objetos e classes, entre outros. Caracteriza-se por apresentar uma reflexão estrutural, não possuindo mecanismos de interceptação de chamada a métodos, como por exemplo, os requeridos por uma ferramenta de teste de aplicações em tempo de execução. Neste protocolo, as solicitações para o meta-nível são feitas pelo nível-base.

O protocolo MetaJava, também conhecido como MetaXa [KLE 97], é implementado através de uma extensão da máquina virtual de Java, permitindo desta forma, a implementação de reflexão estrutural e comportamental. As computações do nível-base geram eventos que são entregues ao meta-nível, sendo estes gerados de forma síncrona.

Este protocolo, permite o encadeamento de múltiplos meta-objetos, no qual o último destes objetos é o para-objeto. Entretanto, segundo Kleinöder [KLE 97], existe carência de um mecanismo que permita que estes objetos possam cooperar entre si, fato

este suprido pelo protocolo Guaraná [OLI 98]. Uma desvantagem do protocolo MetaJava é a carência de um mecanismo para suporte à programação reflexiva. Segundo Kleinöder [KLE 97], durante a configuração e comunicação dos meta-objetos com a MLI, o programador deve referir-se a certas entidades do modelo de programação do protocolo, algumas referidas de forma indireta, através de *strings*. Acontece que o compilador não pode verificar se estas estão sendo aplicadas corretamente, não sendo possível detectar tais problemas num estágio anterior do processo de desenvolvimento. Isto constitui num erro freqüente de programação com este protocolo, referencias a métodos inválidos.

O protocolo Guaraná [OLI 98] é implementado através da modificação da *Kaffe OpenVM*, uma implementação da especificação da máquina virtual Java. É composto por dois objetos de meta-nível: os meta-objetos e os *composers*. Este protocolo apresenta o estilo de reflexão estrutural e comportamental, nas quais as computações no meta-nível são ativadas pelo processo de interceptação de mensagens.

O protocolo Guaraná apresenta características propícias ao teste e depuração de programas em tempo de execução. Facilita o teste de “caixa-branca”, visto que este protocolo, ao implementar o modelo de reflexão por meta-objetos, permite que seja aberta a implementação dos para-objetos, sendo possível a realização de injeção de falhas para posterior análise.

Outra vantagem do protocolo Guaraná é a capacidade de poder projetar diversos meta-objetos de testes distintos, combinando-os de forma a compô-los numa única aplicação, fazendo estes operarem sobre um mesmo para-objeto ou um mesmo grupo de para-objetos. Desta forma, vários tipos de teste podem ser aplicados sobre um ou mais objetos selecionados.

No protocolo Guaraná, a introdução de novos meta-objetos é totalmente realizada no meta-nível, decorrendo disto que a aplicação a ser testada não necessita ser modificada, garantindo com isto que o teste verificará o comportamento real do programa. A desvantagem deste protocolo, é que, como é implementado sobre a modificação de uma máquina virtual, esta torna-se necessária para a execução da aplicação reflexiva, a qual utiliza seus recursos, contradizendo em parte, com a filosofia da linguagem Java.

O protocolo OpenJava difere um pouco dos demais protocolos analisados. Implementa um outro tipo reflexão computacional, chamado intercessão. Segundo Chiba [CHI 99], neste tipo de reflexão não existe degradação de desempenho, pois é empregada uma técnica chamada de reflexão em tempo de compilação. De qualquer forma, esta técnica implica em algumas limitações no processo de reflexão. Segundo Chiba [CHI 99], não é possível alterar a estrutura de uma classe dinamicamente em tempo de execução, sendo necessária então, a modificação da JVM. Esta limitação pode constituir um fator decisivo na sua utilização em uma ferramenta de teste.

As principais características analisadas nos protocolos de reflexão para a linguagem Java estão resumidas na tabela 4.1.

TABELA 4.1 – Resumo das principais características dos protocolos de reflexão

Características / Protocolo	CoreJava	MetaJava	Guaraná	OpenJava
Estilo de Reflexão	Estrutural	Comportamental	Comportamental	Comportamental
Modelo de Reflexão	Meta-Classe	Meta-Objetos	Meta-Objetos	Meta-Objetos
Ativação do meta-nível	Solicitação do nível-base	Interceptação	Interceptação	Interceptação
Chamada dinâmica de métodos	Presente	Presente	Presente	Presente
Modificações na JVM	-	Presente	Presente	-

4 A Ferramenta *KTest*

A ferramenta *KTest* objetiva fornecer apoio às atividades de teste de aplicações escritas na linguagem Java, dando suporte ao teste de *software* baseado em estados. Utiliza para tanto, a tecnologia da Reflexão Computacional para fazer a análise dos estados dos objetos de forma dinâmica, ou seja, durante a execução da aplicação em teste.

Através da especificação de asserções, feitas pelo usuário (testador) da ferramenta, na forma de invariantes de classe, pré e pós-condições, é possível verificar os estados dos objetos da aplicação em teste.

KTest foi desenvolvida na linguagem Java, e utiliza o Protocolo Reflexivo Guaraná versão 1.6 [OLI 98]. Esta versão do protocolo está implementada na JVM *Kaffe OpenVM* 1.0.5, sendo um híbrido entre o versão 1.1 e 1.2 do JDK.

A escolha desse protocolo deu-se por ele apresentar características propícias ao teste e depuração de programas em tempo de execução. É possível, conforme apresentado anteriormente, realizar testes de "caixa-branca", visto que este protocolo, com modelo de reflexão por meta-objetos, permite que seja aberta a implementação das classes, sendo possível a realização de injeção de falhas para posterior análise. Outra vantagem do protocolo Guaraná é a capacidade de poder projetar diversos meta-objetos de testes distintos, combinando-os de forma a compô-los numa única aplicação, fazendo estes operarem sobre um mesmo objeto ou um mesmo grupo de objetos. Desta forma, vários tipos de teste podem ser aplicados sobre um ou mais objetos selecionados. Como neste protocolo, a introdução de novos meta-objetos é totalmente realizada no meta-nível, decorre que a aplicação a ser testada não necessita ser modificada, garantindo com isto que o teste verificará o comportamento real do programa. A desvantagem deste protocolo, é que, como é implementado sobre a modificação de uma máquina virtual, esta torna-se necessária para a execução da ferramenta de teste.

4.1 Características

Utilizando um meta-nível para monitorar objetos de classes selecionadas pelo testador, a ferramenta *KTest* intercepta toda a interação realizada entre para-objetos. Através da utilização de reflexão comportamental sobre as classes escolhidas para teste, mensagens enviadas a objetos instanciados destas classes são interceptadas pelo gerenciador do protocolo, o qual verifica qual meta-objeto está associado ao para-objeto receptor da mensagem, entregando-lhe o controle da aplicação. Este meta-objeto realiza então as computações necessárias à verificação de asserções associadas a métodos e classes.

A ferramenta *KTest* possui as seguintes características e/ou funcionalidades:

- Apoia o teste baseado em estados. Dessa maneira, podem ser avaliadas as várias mudanças de estados pelas quais passam os objetos de determinada classe, tendo-se como base o modelo dinâmico da classe (diagrama/máquina de estados);
- Verifica invariantes de classe, pré e pós-condições de métodos tornando, assim, possível a verificação da integridade dos estados de um objeto durante

a execução do programa. Com a interceptação de mensagens entre objetos, possibilitada através da reflexão destes, tal verificação pode ser realizada através de consultas aos valores dos atributos dos objetos;

- Armazena a seqüência de métodos chamados por para-objetos, tornando possível ao testador visualizar o histórico de interações no nível-base;
- Apresenta uma *interface* gráfica para interação com usuário, o que lhe possibilita escolher as classes, métodos, especificar asserções e visualizar resultados sobre a aplicação em teste.

Todas as funcionalidades descritas são realizadas sem a necessidade de instrumentação do código-fonte da aplicação em teste, constituindo esta, uma das mais importantes características da ferramenta *KTest*. Com a utilização da Reflexão Computacional é possível monitorar os objetos, implementando diferentes mecanismos de análise na aplicação, sem que isso venha a interferir no código do *software* em teste.

A classe *KTest* e suas especializações definem a *interface* de interação com o usuário, gerenciando as informações por ele providas, sendo responsáveis também pela apresentação dos resultados da aplicação monitorada. Através da *interface* para interação com o usuário é possível escolher a aplicação para teste. Logo após, é apresentada a hierarquia de classes da aplicação com seus métodos e atributos de cada classe. Em seguida, o usuário especifica quais classes e/ou métodos ele deseja selecionar para serem monitorados. Para cada classe escolhida pode ser especificada uma invariante associada a esta classe e, para cada método podem ser especificadas a pré e a pós-condição para avaliação, não sendo, entretanto, obrigatória uma especificação dessas asserções.

A fig. 4.1 apresenta o fluxograma de funcionamento da *KTest*.

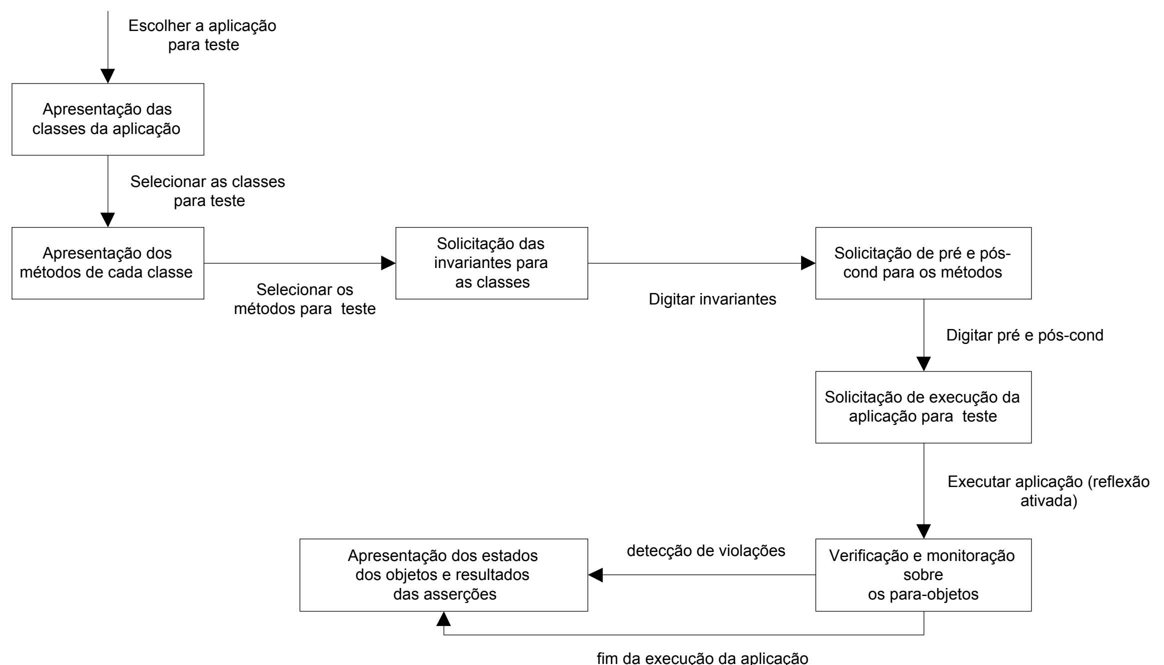


FIGURA 4.1 – Diagrama de Funcionamento da *KTest*

A identificação da hierarquia de classes de uma aplicação é obtida através de reflexão computacional estrutural, através da qual são coletadas todas as informações das classes. Tais informações, adicionadas das seleções feitas pelo usuário (classe/método), bem como as especificações das asserções, são armazenadas numa estrutura de classes, definida no meta-nível.

A próxima etapa consiste na reconfiguração do meta-nível, com o objetivo de instanciar meta-objetos e associá-los às instâncias das classes escolhidas para teste. A meta-classe *KMeta*, que estende a classe *MetaObject*, provê os recursos responsáveis para estas instanciações e operações do meta-nível. A estrutura da classe *MetaObject* foi detalhada na Seção 3.9.3.

A execução da aplicação é interrompida quando mensagens forem encaminhadas a objetos de classes selecionadas para teste, sendo então transferido o controle da aplicação para o meta-nível, onde os métodos dos meta-objetos associados fazem as computações necessárias para a verificação das asserções e, consequentemente, validação dos estados dos para-objetos.

4.1.1 Verificação de Asserções

Através da verificação das asserções é possível determinar se o estado de um objeto está ou não de acordo com a especificação feita pelo usuário. Conforme explanado anteriormente, são três os tipos de asserções especificadas pelo usuário: i) invariantes de classe ii) pré-condições de métodos e iii) pós-condições de métodos.

Sempre que existe interação entre objetos do nível-base, ou seja, quando mensagens são enviadas aos para-objetos, a chamada ao método é reificada como uma operação e entregue ao meta-objeto correspondente ao para-objeto receptor da mensagem. Nesse momento, o meta-nível encarrega-se, primeiramente, de verificar se tal operação refere-se a um objeto cuja classe foi previamente escolhida para teste, através de uma pesquisa na estrutura (de classes) presente no meta-nível. Esta estrutura contém a relação de classes e métodos escolhidas para teste, além de outras informações. Se a classe é encontrada nesta estrutura, ocorre então uma busca referente ao método que foi invocado, de forma a verificar se o mesmo também foi selecionado para teste. Em caso afirmativo, o meta-nível procura a pré-condição especificada para esse método e, se houver, a mesma é avaliada. As informações sobre o estado do objeto (valores dos seus atributos) são, então, captadas para verificar se o estado do para-objeto atende à pré-condição para a ativação do método solicitado. Caso essa pré-condição seja falsa, o meta-nível se encarregará de apresentar ao usuário a violação da asserção.

Sendo a pré-condição verdadeira, o método original é chamado e, logo após o final de sua execução, o controle da aplicação é repassado novamente ao meta-nível, que captará na mesma estrutura, a pós-condição relativa a esse método, se esta existir. Em caso afirmativo, são recuperados os valores dos atributos do objeto e a asserção é verificada, com o objetivo de saber se o estado do objeto atende à pós-condição, que define seu estado final após a ativação do método em questão. Caso a pós-condição seja violada, o meta-nível apresentará ao usuário a situação.

Torna-se mister ressaltar que, sempre que mensagens são interceptadas, independentemente de ativarem ou não métodos escolhidos para monitoração, o meta-nível fará a recuperação da invariante de classe, caso seja uma classe escolhida para

teste. O propósito desta recuperação é verificar, após o término da execução do método, a invariante da classe. As informações do estado do objeto são, então, captadas e a respectiva invariante é validada para constatar se o objeto atende a essa condição.

Quando do término da execução da aplicação em teste ou, quando da ocorrência de uma violação sobre uma asserção, seja ela uma invariante, uma pré ou pós-condição, o usuário pode verificar a seqüência de ativação dos métodos da aplicação em teste e os estados dos objetos (valores dos seus atributos) antes e após a execução dos métodos escolhidos para teste, além do resultado da avaliação de cada uma das asserções.

4.2 Implementação da *KTest*

Nesta seção são apresentadas as classes desenvolvidas para implementar a ferramenta *KTest*. São detalhadas aqui as principais funcionalidades de cada classe, juntamente com uma explicação de cada um dos seus respectivos atributos e métodos, que juntos dão suporte ao comportamento da ferramenta.

4.2.1 Classes desenvolvidas

Quando da escolha de uma aplicação a ser testada pela *KTest*, uma estrutura de classes torna-se necessária, para: i) dar o suporte ao gerenciamento do leque de informações acerca da aplicação em teste e ii) prover as funcionalidades necessárias à execução dos referidos testes.

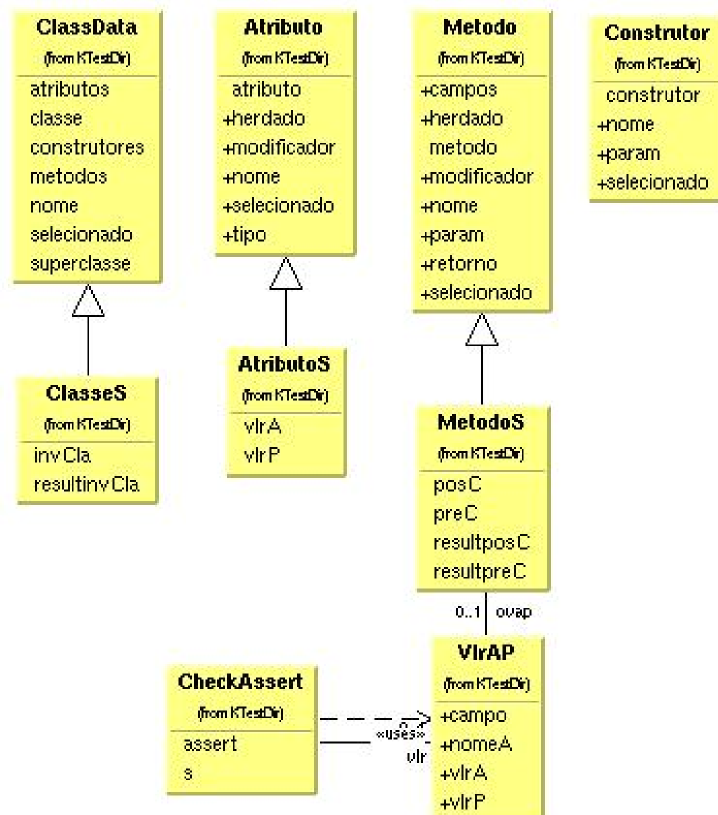


FIGURA 4.2 –Classes que armazenam informações sobre a aplicação em teste

Com relação ao primeiro objetivo, um conjunto de classes foi desenvolvido para armazenar as informações estruturais da(s) classe(s) escolhida(s) para teste, durante a

execução da reflexão estrutural, realizada no início da análise da aplicação. Entre estas informações cita-se: nome da classe, seus atributos (nomes e tipos), seu(s) construtor(es), nome e assinatura de métodos (modificador, tipo de retorno e parâmetros). Tal instrospecção analisa sempre se estas definições ocorreram na classe em análise ou foram herdadas de sua super-classe. São ainda armazenadas nesta estrutura, as asserções especificadas pelo testador após a escolha de classes e/ou métodos para teste.

Esta estrutura, formada pelas classes *ClassData*, *Metodo*, *Atributo*, *Construtor*, *VlrAP* e suas sub-classes, constituem o principal alvo de consultas realizadas pelos métodos dos meta-objetos, os quais a utilizam como fonte de informações para o embasamento das decisões computacionais realizadas no meta-nível. A fig.. 4.2 apresenta os relacionamentos existentes entre essas classes.

As demais classes, como por exemplo, *KTest*, *KUtil* e *KMeta*, são responsáveis pela *interface* da ferramenta, funções de instrospecção e gerenciamento de meta-configuração aplicada sobre a para-aplicação em teste.

4.2.1.1 Classe ClassData

A classe *ClassData* é responsável por armazenar uma coleção de informações de natureza estrutural sobre todas as classes presentes na para-aplicação em teste. Para cada classe presente na para-aplicação, é instanciado um objeto desta classe, o qual armazena em seus atributos:

- o nome da própria classe;
- o nome de sua super-classe;
- o próprio nome da classe;
- um vetor de objetos da classe C*onstrutor*, os quais armazenam informações sobre os construtores definidos na para-classe;
- um vetor de objetos da classe *Atributo*, que armazena informações sobre os atributos da classe;
- um vetor contendo objetos da classe *Método*, que armazena informações sobre os métodos declarados na referida para-classe;
- um valor lógico indicando se esta classe foi ou não escolhida para monitoramento.

A fig. 4.3 apresenta os atributos da classe *ClassData.*

Classe *ClassData*

`Class classe;`
`String superclasse;`
`String nome;`
`Vector construtores;`
`Vector atributos;`
`Vector metodos;`
`boolean selecionado;`

FIGURA 4.3 – Classe *ClassData*

4.2.1.2 Classe *ClasseS*

A classe *ClasseS* é uma especialização da classe *ClassData*, a qual armazena somente as classes que efetivamente foram escolhidas para monitoramento pelo testador. Esta decisão de projeto fundamenta-se na constante busca de minimização do tamanho das estruturas presentes no meta-nível, as quais serão alvos de sucessivas buscas por informações do para-nível. Desta forma, um considerável ganho de desempenho foi obtido nas interações de meta-nível. Na fig. 4.4 são apresentados os atributos da classe *ClasseS*.

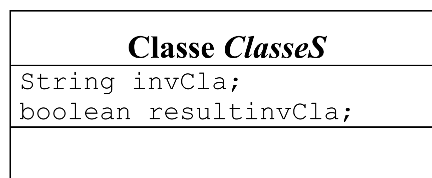


FIGURA 4.4 – Classe *ClasseS*

O atributo *invCla* armazena a invariante de classe especificada pelo testador, enquanto que o atributo *resultinvCla* detém o valor do resultado da avaliação desta asserção.

4.2.1.3 Classe Construtor

Na classe *Construtor* são armazenados os dados referentes aos construtores de cada classe presente na para-aplicação, conforme mostra a fig. 4.5. No atributo *construtor* é armazenado um objeto do tipo *Constructor*, no atributo *nome* a descrição do construtor e, no vetor *param*, todos os parâmetros declarados em cada implementação dos possíveis construtores existentes para a classe em investigação.

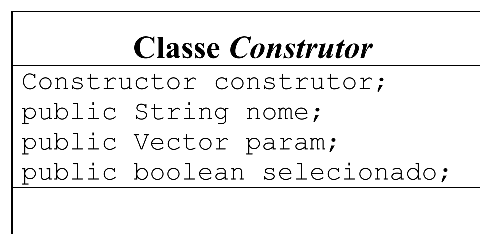


FIGURA 4.5 – Classe *Construtor*

Objetos desta classe são instanciados para cada construtor de cada classe definida na para-aplicação, sendo armazenados no atributo *construtores* dos objetos instanciados a partir da classe *ClassData*.

4.2.1.4 Classe Atributo

Os objetos instanciados a partir da classe *Atributo* são responsáveis pelo armazenamento das informações referentes a cada atributo de cada classe da para-

aplicação, conforme mostra a fig. 4.6. A variável de instância *atributo* contém um objeto do tipo *Field*, o qual representa o atributo em questão. *nome, tipo e modificador* são strings que representam o nome, o tipo e o modificador do atributo respectivamente. A variável *selecionado* indica se esta variável de instância pertence a uma classe escolhida ou não para monitoramento. Já a variável de instância *herdado*, indica se este atributo, que foi identificado durante a reflexão estrutural, foi originalmente declarado na classe que está sofrendo a instrospecção ou foi herdado de sua super-classe.

Classe *Atributo*

`Field atributo;`
`public String nome;`
`public String tipo;`
`public String modificador;`
`public boolean selecionado;`
`public boolean herdado;`

FIGURA 4.6 – Classe *Atributo*

Esta distinção faz-se necessária devido ao fato do testador precisar vislumbrar quais campos são originalmente especificados por uma classe, e quais são herdados. Isto facilita a especificação de asserções para determinados atributos, face sua perfeita compreensão da real estrutura de classes da para-aplicação em teste. Outra justificativa é a não recuperação direta de atributos herdados utilizando a reflexão estrutural CoreJava, necessitando mecanismos adicionais para determinar a real localização das declarações dos atributos.

Objetos desta classe são armazenados também num vetor (atributo) da classe *ClassData*, o qual representa a coleção de objetos declarados como variáveis de instância para uma determinada classe.

4.2.1.5 Classe AtributoS

A classe *AtributoS* representa os atributos das classes que foram selecionadas para monitoramento pelo testador após a fase da reflexão estrutural. É uma sub-classe de *Atributo*, onde foram adicionados mais duas variáveis de instância: *vlrA* e *vlrP*, sendo ambas do tipo *Object*, representando o valor do atributo antes e após a execução de um método. A fig. 4.7 apresenta a estrutura desta classe.

Classe *AtributoS*
`Object vlrA;` `Object vlrP;`

FIGURA 4.7 – Classe *AtributoS*

A justificativa para a criação dessa classe, assemelha-se à da criação da classe *ClasseS*. A redução do tamanho de estruturas que sofrerão constantes acessos no meta-nível leva a consideráveis ganhos de desempenho. O tipo dos atributos *vlrA* e *vlrP* foram definidos como *Object*, pois o protocolo Guaraná devolve os valores dos atributos que são lidos durante a interceptação como objetos desta classe para evitar

incompatibilidade, visto que todos os objetos em Java são instâncias de classes que descendem de *Object*. Somente no momento da validação das asserções os objetos serão convertidos para os tipos que foram originalmente definidos.

4.2.1.6 Classe *VlrAP*

A classe *VlrAP* possui a finalidade de instanciar objetos que armazenam informações sobre os tipos e valores dos atributos dos objetos de classes escolhidas para monitoramento. Possui quatro atributos, conforme mostra a fig. 4.8. O atributo *campo* contitui-se num vetor de objetos do tipo *Field*, representando os atributos de uma determinada classe escolhida para teste. O vetor *nomeA* armazenam *strings* representando os nomes dos atributos da classe. Já os vetores *VlrA* e *VlrP* armazenam os valores dos atributos antes e após a execução dos métodos escolhidos para teste. Desta forma, um objeto *VlrAP* é instanciado para cada objeto *MetodoS* criado, possibilitando assim um completo conjunto de informações sobre os estados dos objetos.

Classe *VlrAP*

`Field campo[];`
`Vector nomeA;`
`Vector vlrA;`
`Vector vlrP;`

FIGURA 4.8 – Classe *VlrAP*

4.2.1.7 Classe *Metodo*

A classe *Metodo* é responsável pelo armazenamento de informações pertinentes a todos os métodos de cada classe da para-aplicação. A fig. 4.9 apresenta suas variáveis de instância.

Classe *Metodo*

`Method metodo;`
`public String nome;`
`public String retorno;`
`public Vector param;`
`public String modificador;`
`public boolean selecionado;`
`public boolean herdado;`

FIGURA 4.9 – Classe *Metodo*

A recuperação das informações de cada método em cada classe da para-aplicação são armazenadas nas seguintes variáveis de instância: *metodo* armazena um objeto do tipo *Method*, o qual representa o método em investigação; *nome* armazena o nome do método; *retorno* o tipo de valor retornado por este método; e *modificador* armazena um possível modificador para este método. Já as variáveis de instância *param*, constitui-se num vetor contendo a lista de parâmetros que identifica a assinatura do método, *selecionado* indica se este método foi ou não escolhido para ser interceptado

durante a execução da para-aplicação. A justificativa para esse campo dá-se pelo mesmo motivo de ganho de desempenho no meta-nível. Desta maneira, somente os métodos escolhidos pelo testador são colocados nas estruturas de meta-nível que fornecem informações aos métodos dos meta-objetos responsáveis pelo monitoramento nos para-objetos vinculados. O atributo *herdado* indica se este método esta sendo herdado de uma super-classe ou é definido na classe que está sofrendo introspecção.

4.2.1.8 Classe *MetodoS*

A classe *MetodoS,* sub-classe de *Metodo*, foi acrescida de cinco variáveis de instância, conforme apresenta a fig. 4.10: *preC* e *posC* armazenam respectivamente a pré e pós-condição para o método selecionado para teste, *resultpreC* e *resultposC*, representando respectivamente os resultados das avaliações das asserções e *ovap*, a qual constitui-se num objeto da classe *VlrAP*. Conforme explanado anteriormente, é possível manter os valores de todos os atributos de cada objeto, antes e após a execução de cada método escolhido para teste. Mais uma vez, a criação desta sub-classe deu-se pelos motivos de desempenho já citados anteriormente.

Classe *MetodoS*

`String preC;`
`String posC;`
`boolean resultpreC;`
`boolean resultposC;`
`VlrAP ovap;`

FIGURA 4.10 – Classe *MetodoS*

4.2.1.9 Classe *KTest*

A classe *KTest* compreende uma das principais classes da ferramenta desenvolvida. Provê a principal *interface* de comunicação com o usuário, através da qual é escolhida a aplicação a ser testada, bem como onde são selecionados as classes e métodos a serem monitorados. Na fig. 4.11 são apresentados alguns dos atributos e métodos presentes nesta classe.

Classe *KTest*

`Button bEspecAss`
`List listClasses`
`List listMet`
`List listMetH`
`. . .`
`Vector classes`
`Vector classesS`

`private void AbriAplicActionPerformed`
`private void bEspecAssActionPerformed`
`private void bExecAplicActionPerformed`
`public void showInfo`
`public void itemStateChanged`
`. . .`

FIGURA 4.11 – Classe *KTest*

O vetor *classes* é uma variável de instância que contém a estrutura de classes da para-aplicação. Compreende uma coleção de objetos do tipo *ClassData* explicados anteriormente. Já a variável de instância *classesS*, contém somente as informações estruturais das classes e métodos efetivamente selecionados para monitoramento.

O método *AbriAplicActionPerformed* tem como função permitir a escolha da aplicação a ser monitorada pela ferramenta *KTest*, enviando após isso, as mensagens necessárias aos métodos da classe *KUtil*, *cuja* atribuição é realizar a instrospecção nas classes da para-aplicação.

O método *bEspecAssActionPerformed* é o responsável por criar uma segunda estrutura de armazenamento das informações estruturais, a qual contém somente as classes e métodos escolhidos para teste. Desta maneira, o acesso a tais estruturas torna-se mais rápido nas interações de meta-nível. Constitui ainda sua responsabilidade a invocação de métodos presentes em outras classes, descritas a seguir, responsáveis pela especificação das asserções.

Já o método *bExecAplicActionPerformed* é chamado para executar a aplicação escolhida, disparando os métodos necessários para a ativação da reflexão computacional e dos mecanismos de interceptação, presentes na classe *KMeta*, descrita a seguir.

Por último, o método *showInfo* encarrega-se de mostrar na *interface* de *KTest* todas as informações relacionadas às introspecções realizadas. O método *itemStateChanged* é um *listener* cuja atribuição é gerenciar a *interface* com o usuário na escolha das classes e métodos que serão selecionados.

4.2.1.10 Classe *KMeta*

A classe *KMeta* é a meta-classe da ferramenta. Implementa os métodos necessários ao gerenciamento das meta-informações, provendo as estruturas necessárias ao controle das interceptações. A fig. 4.12 mostra os relacionamentos que *KMeta* possui com as classes *Operation*, *Result, OperationFactory* e *Message*, definidas no protocolo Guaraná, conforme explicado na Seção 3.9.3, bem como seus principais atributos. Seus principais métodos são listados na fig. 4.13.

KMeta apresenta primeiramente três *hashtables*: *hClasseS*, *hClasseSAtr* e *hClasseSMet*. A primeira contém objetos *ClasseS*, ou seja, as classes escolhidas para monitoração. Possui como chave de acesso objetos *Class*, caracterizando-se desta forma como um mecanismo de acesso com extrema velocidade na recuperação de informações, uma vez que é possível obter objetos do mesmo tipo da chave no meta-nível durante as interceptações. A utilização de *hashtables* no meta-nível é explicada na Seção 4.2.3. A segunda *hashtable*, *hClasseSAtr*, contém como chave objetos do tipo *Class* e como conteúdo objetos do tipo *VlrAP*. A terceira *hashtable*, *hClasseSMet*, contém também como chave objetos do tipo *Class* e como conteúdo outra *hashtable* que possui como chave objetos *Method* e como conteúdo objetos *MetodoS* da classe selecionada para teste.

Uma quarta *hashtable*, denominada *pending*, é utilizada para evitar o processo de recursão infinita no meta-nível, que será abordado na Seção 4.2.2.

O vetor *vCla* é usado para associação ao vetor preenchido com os elementos de teste (classes e/ou métodos) na classe *KTest*, escolhidos pelo testador.

Os vetores *vMetHist* e *vObjHist* são utilizados para armazenar o histórico da invocação de métodos, bem como dos objetos que receberam tais mensagens, tornando possível ao testador visualizar o histórico das interações que ocorreram no para-nível.

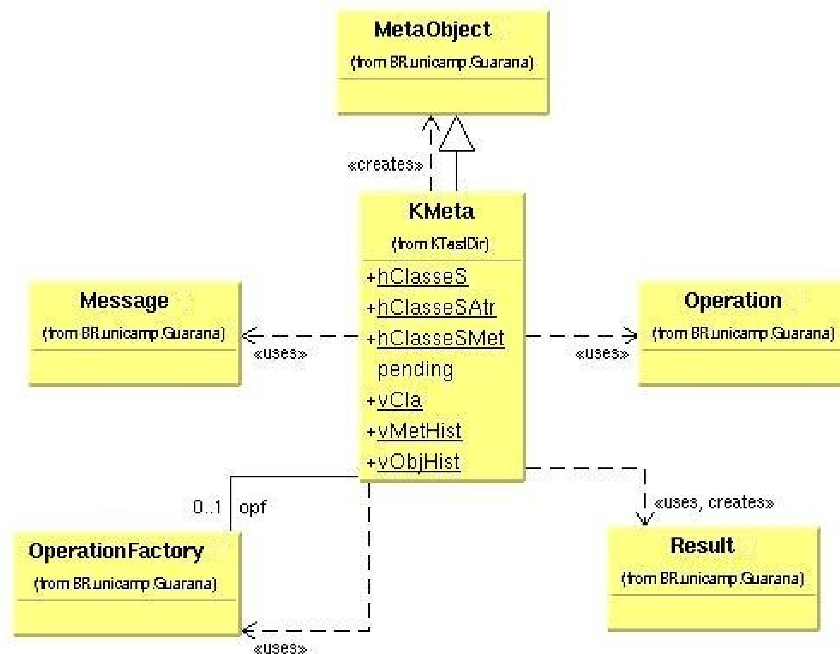
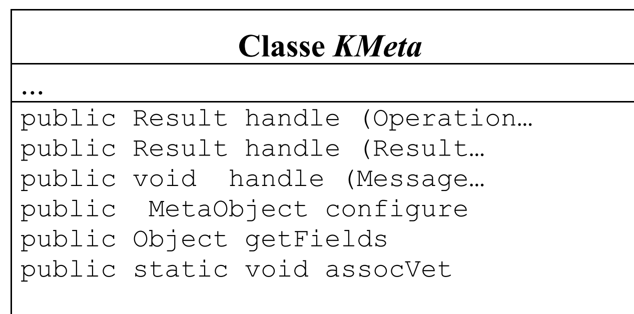


FIGURA 4.12 – *KMeta* - Classe de Meta-Nível da *KTest*



Classe *KMeta*

...

```
public Result handle (Operation…
public Result handle (Result…
public void  handle (Message…
public  MetaObject configure
public Object getFields
public static void assocVet
```

FIGURA 4.13 – Classe *KMeta*

Os métodos definidos nesta meta-classe constituem os métodos declarados na *interface* do protocolo Guaraná, sendo os quatro primeiros explicados na Seção 3.9.4. O método *getFields*, possui como atribuição realizar a leitura nos valores dos atributos dos objetos que estão sendo interceptados e cujas classes tenham sido previamente escolhidas para teste. Este método deve tratar também o problema da recursividade infinita no meta-nível, discutida na Seção 4.2.2. O último método, *assocVet*, é um método estático cuja finalidade é associar o atributo *vCla* à estrutura de teste preenchida na classe *KTest*, explicado anteriormente.

4.2.1.11 Classe *KUtil*

KUtil foi desenvolvida com o objetivo de fornecer métodos que gerenciem o processo de instrospecção, ou seja, na atividade de reflexão estrutural realizada na primeira fase de teste, quando o testador seleciona uma aplicação a ser monitorada. Os principais métodos definidos nesta classe são apresentados na fig. 4.14.

Classe *Kutil*
...

`public static void Disc_All`
`public static Vector getInhFields`
`public static Vector getInhMethods`

FIGURA 4.14 – Classe *KUtil*

O método *Disc_All*, realiza todo o procedimento de reflexão estrutural, preenchendo as estruturas de meta-nível com as informações das classes da para-aplicação a ser monitorada. *getInhFields* possui como atribuição realizar instrospecção nos atributos que são herdados pela classe que está sendo analisada, permitindo com isso que sejam separados os atributos herdados dos definidos na própria classe. Já o terceiro método, *getInhMethods*, possui a mesma finalidade do anterior, direcionando seus propósitos em separar os métodos herdados dos implementados na classe em análise. Ressalta-se que tais métodos são estáticos, não necessitando com isso que sejam instanciados objetos da classe *KUtil*.

4.2.1.12 Classe diaEspInvClass

Esta classe permite ao testador especificar as invariantes de classe para as classes previamente selecionadas para teste. Possui também como atribuição apresentar a *interface* necessária, bem como seu controle, para a interação com o usuário. É uma sub-classe da classe `java.awt.Dialog`. Seus principais atributos e métodos são mostrados a fig. 4.15.

Classe *disEspInvClass*

`private java.awt.Label label1;`
`private java.awt.Panel panel2;`
`private java.awt.List listClasseS;`
…
`Vector vclasseS;`
`ClasseS cs;`
`AtributoS as;`

`private void bVerifInvActionPerformed` `private void listClasseSItemStateChanged`

FIGURA 4.15 – Classe *disEspInvClass*

Os primeiros atributos referem-se a objetos do pacote *AWT* da linguagem Java. O atributo *vclasseS* é uma referência para a estrutura de armazenamento das informações sobre as classes no meta-nível. Deste modo, podem ser incluídos as

informações sobre as invariantes aqui especificadas nas respectivas classes. Os dois últimos atributos, *cs* e *as*, são objetos das classes *ClasseS* e *AtributoS* respectivamente, já definidas anteriormente e, servem de suporte às operações necessárias a inclusão das asserções.

O método *bVerifInvActionPerformed* possui duas atribuições: i) repassar a asserção (invariante de classe) especificada para a classe *CheckAssert*, a qual realizará uma análise léxica e sintática, bem como se certificará de que as variáveis especificadas constituem atributos da classe, a qual a invariante se refere e ii) armazenar a invariante na estrutura de dados. O método *listClasseSItemStateChanged* é um *listener* necessário ao gerenciamento das interações da *interface* com o testador.

4.2.1.13 Classe diaEspPrePos

Esta classe possui quase as mesmas atribuições da classe anterior, referindo-se, entretanto, à especificação de pré e pós-condições dos métodos selecionados para monitoramento. É também definida como sub-classe da classe `java.awt.Dialog`. A fig. 4.16 apresenta seus principais atributos e métodos.

Classe *disEspPrePos*

```
private java.awt.Label label3;
private java.awt.List listMetS;
private java.awt.List listAtribS;
...
Vector vclasseS;
Vector vaux;
ClasseS cs;
AtributoS as;
MetodoS ms;
PosXY p;
```

```
private void bVerifPrePosActionPerformed
private void listMetSItemStateChanged
```

FIGURA 4.16 – Classe *disEspPrePos*

Os atributos *vaux*, *cs*, *as*, *ms* e *p*, constituem atributos auxiliares necessários ao armazenamento das asserções no vetor *vclasseS*, o qual constitui uma referência para a supra-citada estrutura de armazenamento no meta-nível. O método *bVerifPrePosActionPerformed* possui também duas funções principais: repassar as asserções especificadas (pré e pós-condições) para a classe *CheckAssert* para análise, bem como seu armazenamento. O último método, *listMetSItemStateChanged* compreende o *listener* responsável pelo gerenciamento da interação com o testador.

4.2.1.14 Classe CheckAssert

A classe *CheckAssert* foi desenvolvida com o objetivo de realizar a verificação léxica e sintática das asserções, além de realizar suas validações.

Conforme mostra a fig. 4.17, esta classe apresenta três atributos: *assert*, *desc* e *vlr*. O primeiro compreende um objeto do tipo *JEP* (*Java Expression Parser*), utilizado para realizar as validações sobre as asserções, descrito na Seção 4.2.4. O segundo

atributo, armazena um *string* correspondente à asserção especificada. Já o terceiro, é um objeto do tipo *VlrAP*, explicado anteriormente.

Classe *CheckAssert*

`JEP assert;`
`String desc;`
`VlrAP vlr;`

`public boolean checkValue ( )` `public boolean checkSintax ( )`

FIGURA 4.17 – Classe *CheckAssert*

Os métodos *checkValue* e *checkSintax* realizam a avaliação da expressão ou a verificação de sua sintaxe, respectivamente. Ambos retornam valores *booleanos*.

4.2.1.15 Classe diaResultAssert

Responsável pela apresentação dos resultados dos testes ao usuário, *diaResultAssert* é sub-classe de *java.awt.Dialog*. Possui atributos do pacote *AWT* e implementa os métodos necessários para a interação com o usuário. A fig. 4.18 apresenta seus principais atributos e métodos.

Classe *diaResultAssert*

`private java.awt.List lCla;`
`private java.awt.List lMet;`
`private java.awt.Label label3;`
`...`
`Hashtable hhClassS;`
`Hashtable hhClasseSMet;`
`Vector vvObjHist;`
`Vector vvMetHist;`

`private void bPreActionPerformed`
`private void bPosActionPerformed`
`private void bInvClaActionPerformed`
`private void bHistActionPerformed`
`private void listClaItemStateChanged`
`private void listMetItemStateChanged`

FIGURA 4.18 – Classe *disResultAssert*

As *hashtables hhClassS* e *hhClasseSMet* armazenam a coleção de classes e métodos selecionados para teste, juntamente como os estados dos objetos (valores de seus atributos) antes e após a execução dos métodos, além da avaliação das asserções. Nos vetores *vvObjHist* e *vvMetHist* está armazenada a seqüência de ativação dos métodos da aplicação submetida ao teste. Os métodos implementados nesta classe são responsáveis pela apresentação dos resultados ao usuário, os quais podem ser: avaliação das invariantes, avaliação das pré e/ou pós-condições dos métodos e a relação dos métodos invocados pelo para-nível. Estão aqui implementadas também, os *listeners* necessários ao controle da seleção das classes e/ou métodos realizada pelo usuário.

4.2.2 Problema Crítico: Recursão Infinita no Meta-nível

Durante o desenvolvimento da ferramenta, surgiu um problema crítico que requereu um trabalho intenso para tornar possível o desenvolvimento da ferramenta *KTest*. Devido a características de implementação do protocolo Guaraná, bem como a forma de sua arquitetura, operações de interceptação de para-objetos podem ocasionar a ocorrência de recursividade infinita de interceptações. Estas características são explicadas pelo autor do protocolo como fatores de flexibilidade.

Conforme explicado na Seção 3.9, segundo a terminologia do MOP Guaraná, um para-objeto sofre e executa operações. Operações de leitura ou escrita em atributos de para-objetos constituem operações sobre estes, assim como invocações a métodos pertencentes a estes mesmos para-objetos [OLI 98c]. Desta forma, quando existe interação entre para-objetos na aplicação em teste, estas são interceptadas e reificadas pelo núcleo do MOP.

Devido aos propósitos da ferramenta desenvolvida, operações de leitura sobre atributos de para-objetos que acabaram de ter suas interações interceptadas e conseqüentemente reificadas (com o objetivo de verificar asserções e validar o estado do referido para-objeto) tornam-se necessárias. Ressalta-se que somente são executadas tais operações quando as interceptações no nível-base constituem interações sobre para-objetos de classes selecionadas para teste ou invocação de métodos previamente selecionados para monitoramento. Disto decorre que novas operações sobre esses para-objetos devem ser executadas. Assim, como ocorre um acesso ao para-objeto, partindo esse acesso do seu meta-objeto vinculado, novas interceptações e novas reificações ocorrem, levando a uma recursividade infinita no meta-nível.

O tratamento para o processo de atuação de um meta-objeto é codificado na primitiva *handle(Operation, paraObj)*. Desta forma, sempre que ocorre uma interceptação, chegam nessa primitiva dois parâmetros: i) a operação (materializada em um objeto *Operation*), contendo informações sobre o objeto alvo desta operação, o tipo da operação e seus argumentos; ii) o para-objeto alvo da operação. Sendo identificada a operação como uma invocação de um método, é então criada uma operação de leitura para todos os atributos do referido para-objeto, através da fábrica de operações. Esta criação dá-se nesta própria primitiva *handle(Operation, paraObj)* ou chamando-se outro método a partir dela.

A solução adotada para este problema consiste então, em armazenar uma referência para essa operação de leitura criada sobre o para-objeto em uma *HashTable*. Este procedimento é feito após a criação da operação de leitura e antes de sua execução. Desta maneira, quando esta operação é executada sobre um atributo do para-objeto (o que gera nova interceptação e conseqüentemente uma nova chamada a *handle(Operation, paraObj)*), é testada neste método a existência de uma referência para esta operação na *hashtable*. Em caso afirmativo, o meta-objeto retorna como resultado *null*, desprezando a reinterceptação gerada. Uma vez concluída a operação de leitura sobre cada atributo, a referência para esta operação é retirada da *HashTable*, pois caso contrário, somente o primeiro atributo seria efetivamente lido e novamente recair-se-ia na situação de recursão infinita. Ressalta-se que, para a leitura de cada atributo definido no para-objeto, uma nova operação de leitura precisa ser criada.

Assim, torna-se clara a existência da *HashTable pending* como atributo da classe *KMeta*, comentada na Seção 4.2.1.10. Na variável *pending* são armazenadas as referências a todas as operações de leituras criadas sobre os para-objetos, quando as

interceptações ocorridas no nível-base constituem interações relevantes para o teste. Tais operações também são criadas quando da finalização da execução de um método, sendo, entretanto, criadas a partir da primitiva *handle(Result, paraObj)*, a qual é notificada quando do término de uma operação, caso esta tenho sido previamente solicitada pelo meta-objeto.

4.2.3 *HashTables*: ganho de desempenho em meta-interações

Inicialmente, a estrutura de armazenamento de informações sobre as para-classes presentes no meta-nível foi organizada sobre a forma de vetores de objetos. Decorreu disto que um baixo nível de desempenho foi constatado, quando do monitoramento de aplicações onde se escolhia um grande número de classes e/ou métodos para monitoramento. O mesmo fato foi notado sobre aplicações que apresentavam uma grande interação entre seus objetos. O elevado número de consultas a esta estrutura, cujas decisões computacionais estão embasadas, degradava de certa forma o desempenho do teste.

Analisando-se alguns trechos de códigos-exemplo sobre a utilização do protocolo Guaraná, foi constatado o constante uso de estruturas que possibilitavam um aumento de desempenho quando da utilização de reflexão computacional, visto que esta tecnologia apresenta grande *overhead*. Foi então que decidiu-se alterar em parte o modo como as informações no meta-nível passariam a ser tratadas.

As *HashTables* constituem um mecanismo extremamente útil para o armazenamento organizado de dados, aumentando o desempenho quando da recuperação destes. *Hashtables* funcionam através da associação de uma chave a um valor (função de mapeamento), ambos armazenado em memória. Ao invés de pesquisar-se todos os elementos armazenados em busca de um certo elemento, o uso das chaves permite o uso de um esquema de indexação rápida extremamente eficiente e que pode ser utilizado com vantagens em inúmeras situações.

Java oferece a classe *java.util.Hashtable* onde tanto as chaves como os valores armazenados são do tipo *Object*, ou seja, podem ser objetos de qualquer classe. Contudo, dados de tipos primitivos não podem ser usados como chave nem armazenados diretamente, sendo necessário o uso de classes *Wrapper* oferecidas pela linguagem Java.

À medida que objetos são inseridos na *HashTable* esta é automaticamente expandida, eliminando a necessidade do programador de controlar seu tamanho.

Utilizou-se como chave para a estrutura do meta-nível, objetos do tipo *Class*, que representam as classes do para-nível. Assim, através de uma chamada ao método *BR.unicamp.Guarana.Guarana.getClass( )* pode-se descobrir a classe do objeto interceptado e utilizar o retorno desse próprio método como chave de pesquisa na *HashTable* que armazena as informações das classes. A fig. 4.19 mostra um trecho do método *handle(Operation, paraObj)* da meta-classe *KMeta*. Neste trecho observa-se que é obtido primeiramente o objeto *Class* que representa a classe do objeto ao qual a chamada de um determinado método foi interceptada. Logo em seguida é realizado um teste para verificar se foi encontrado um objeto *ClasseS* na *HashTable hClasseS*.

```
...
if ( op.isMethodInvocation() ) {

  //verifica se o objeto e' de uma classe escolhida para teste...
  Class c = Guarana.getClass(ob);
  ClasseS cd = (ClasseS) hClasseS.get(c);

  if ( cd  == null )
  return Result.noResult;

...
```

FIGURA 4.19 – Utilizando *HashTables*

Similarmente, no controle da criação de operações de leitura dos valores dos atributos dos para-objetos, as próprias operações foram usadas como chave e conseqüentemente como conteúdo, uma vez que operações constituem objetos do tipo *Operation*, conforme mostra o trecho de código apresentado na fig. 4.20.

```
...
Operation op = opf.read(ff[i]);

pending.put(op, op);        //coloca esta operacao na HashTable
                            //pending para evitar reinterceptacao
Object value = op.perform().getObjectValue();
pending.remove(op);         //retira operacao da Hash pending
...
```

FIGURA 4.20 – Utilizando *HashTables*

4.2.4 Validação e teste das asserções

Para realizar a verificação léxica e sintática das asserções, além de realizar suas avaliações, foi desenvolvida a classe *CheckAssert*, apresentada na Seção 4.2.1.14.

Utiliza para tal, o pacote JEP – *Java Expression Parser*. Criado por Nathan Funk [FUN 2001], este pacote permite realizar análise léxica, sintática e avaliar expressões matemáticas e lógicas. O JEP foi gerado através da submissão de uma gramática de expressões (de acordo com a BNF de Java) ao *software* JavaCC – *Java Compiler Compiler* [SUN 2000]. JavaCC é uma ferramenta de geração de código baseado na especificação de gramáticas, as quais são convertidas em código Java, com o intuito de fazer análises e avaliações sobre diversos tipos de expressões. Referências sobre o JEP e/ou JavaCC podem ser encontrados em [SUN 2000] [FUN 2001].

```
...
org.nfunk.jep.JEP parser = new org.nfunk.jep.JEP();

parser.addVariable("x", 0);
parser.addVariable("y", 5);
parser.addVariable("z", 10);
parser.parseExpression(“y >= x && z > (x+y)”);
result = parser.getValue();

...
```

FIGURA 4.21 – Exemplo da utilização do pacote *JEP*

A fig. 4.21 apresenta um trecho de código onde uma pequena expressão é submetida à avaliação.

O primeiro passo consiste em criar um objeto do tipo JEP. Logo em seguida, utilizando o método *addVariable*, adiciona-se ao objeto as variáveis desejadas, juntamente com seus respectivos valores. Caso o valor a ser passado compreenda um *wrapper*, deve-se utilizar o método *addVariableAsObject*. O método *parseExpression* recebe como parâmetro a expressão a ser avaliada. Finalmente, invocando-se o método *getValue* é possível obter-se o resultado da expressão. Entretanto, caso houverem expressões booleanas, por exemplo, *vlr == false*, deve-se converter os valores lógicos para *0* (zero) ou *1* (um), respectivamente falso e verdadeiro, uma vez que o JEP assim os interpreta. O método *hasError*, pode ser utilizado antes do método *getValue*, sob forma de assegurar que na expressão submetida à avaliação, constem somente variáveis que foram previamente definidas. Da mesma maneira, erros na expressão podem ser detectados.

A versão atual (2.10) do pacote JEP suporta variáveis definidas pelo usuário, constantes, expressões com *strings* e possui implementação da maioria das funções matemáticas. Desta forma, expressões como *nome == "John"*, por exemplo, são perfeitamente possíveis.

Conforme apresentado na fig. 4.17, a classe *CheckAssert* possui o atributo *assert*, o qual é um objeto do pacote JEP. Objetos do tipo JEP são instanciados a cada nova submissão de uma asserção para avaliação sintática e/ou avaliação. O atributo *desc* corresponde à asserção em si, a qual será repassada como argumento ao objeto instanciado *assert*. O atributo *vlr*, da classe *VlrAP*, compreende a relação de atributos e seus respectivos valores, presentes na classe do objeto em que uma invariante de classe ou uma pré/pós-condição de um método esteja sendo avaliada.

4.3 Funcionamento da Ferramenta

Sob forma de explicar o funcionamento da ferramenta *KTest*, além de avaliar suas reais contribuições, submeteu-se à *KTest*, uma aplicação para monitoramento.

A aplicação escolhida foi um programa de Contas Bancárias, desenvolvido para a disciplina de Engenharia de *Software* durante o curso de mestrado.

A fig. 4.22 apresenta o diagrama de classes da aplicação de Contas Bancárias.

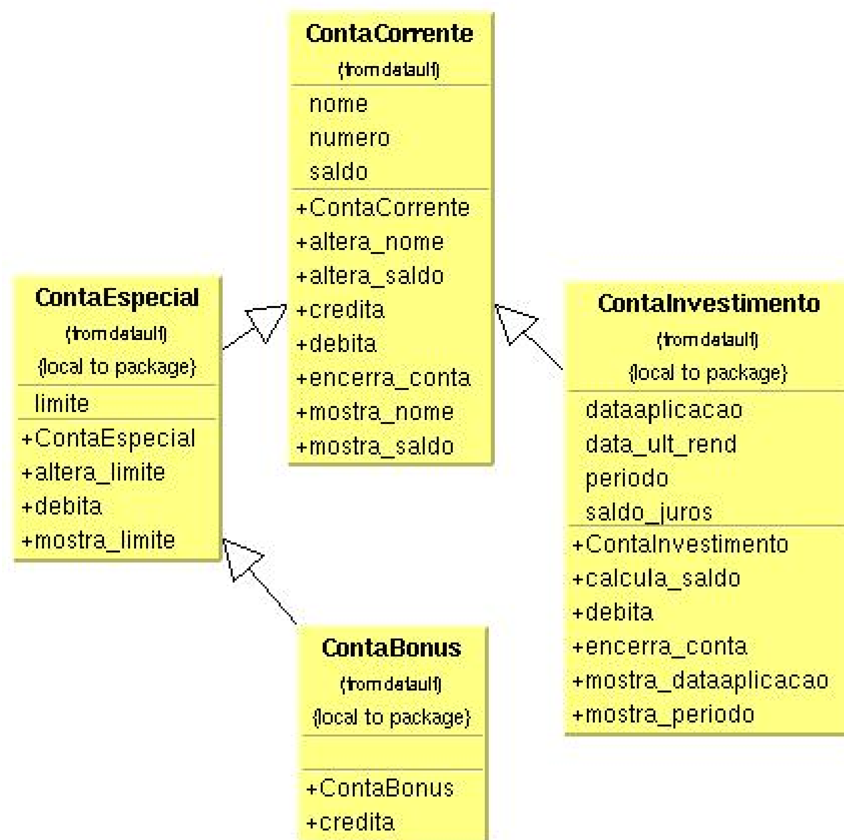


FIGURA 4.22 – Diagrama de Classes – *Contas Bancárias*

A classe principal, *ContaCorrente*, apresenta duas subclasses: *ContaEspecial* e *ContaInvestimento*. A classe *ContaEspecial* possui uma subclasse denominada *ContaBonus*.

O aplicativo Contas Bancárias permite o cadastro de quatro tipos de contas:

a) Contas corrente: são contas comuns, com número, nome do correntista e saldo atual. Não é permitido que o correntista tenha saldo negativo;

b) Contas especiais: são contas que, além das características das contas correntes comuns, possuem um limite que permite ao correntista ter um saldo negativo até esse valor;

c) Contas investimento: são aplicações feitas por uma pessoa por determinado tempo. Os juros são creditados a cada mês. Esses juros podem ser alterados no próprio programa, através do menu manutenção.

d) Contas especiais com bônus: são contas especiais com o diferencial que, quando um valor é creditado à conta, caso o novo saldo seja o triplo do limite da conta, o novo limite será igual à metade do saldo.

O aplicativo permite que contas-corrente ou investimentos com saldo zero sejam fechadas, através de uma opção no menu principal. Este aplicativo possui uma *interface* de interação com o usuário, implementada através de classes específicas que, como não serão escolhidas para monitoramento, não são descritas neste trabalho.

Através da *interface* do aplicativo, o usuário pode realizar todas as operações sobre os quatro tipos de contas citadas acima. O método *credita* recebe como parâmetro

um valor, o qual é acumulado ao saldo existente e armazenado na variável de instância *saldo*. Já o método *debita*, também recebe como parâmetro um valor, o qual será subtraído da variável de instância *saldo*. O método credita foi redefinido na classe *ClassBonus* pelo fato de realizar cálculos extras a fim de estabelecer o novo limite da conta. Os demais métodos, tais como *mostra_nome*, *mostra_saldo*, *mostra_limite* são usados em operações de consultas sobre as contas.

Assim que o usuário ativa a *KTest*, deverá ser feita a escolha das classes sobre as quais serão realizadas reflexões computacionais estruturais, com o objetivo de apresentar ao usuário a hierarquia de classes presente na aplicação, bem como as informações sobre cada uma, como seus atributos, métodos, etc. A fig. 4.23 apresenta a classe escolhida para monitoramento, *ContaBonus*.

Uma vez realizada a reflexão estrutural, *KTest* apresenta todas as super-classes de *ContaBonus*, respectivamente *ContaEspecial* e *ContaCorrente*, seus construtores, métodos e atributos nelas definidos e/ou herdados, conforme mencionado anteriormente. Escolhendo-se outras classes, as respectivas informações sobre as mesmas serão apresentadas nos locais indicados. Ressalta-se que, neste momento, devem também ser selecionados os métodos que serão monitorados durante a execução da aplicação, caso houverem.

Uma vez escolhidas as classes e, possivelmente os métodos para monitoramento, neste caso o método *credita* e *debita* da classe *ContaBonus*, o próximo passo consiste em especificar as asserções, ou seja, as invariantes de classe e as pré e pós-condições dos métodos selecionados. Entretanto, a especificação das asserções não é, em momento algum, obrigatória, recebendo o valor *null* caso não determinada. A fig. 4.24 apresenta a tela para a digitação das invariantes das classes escolhidas, neste caso, a classe *ContaBonus*.

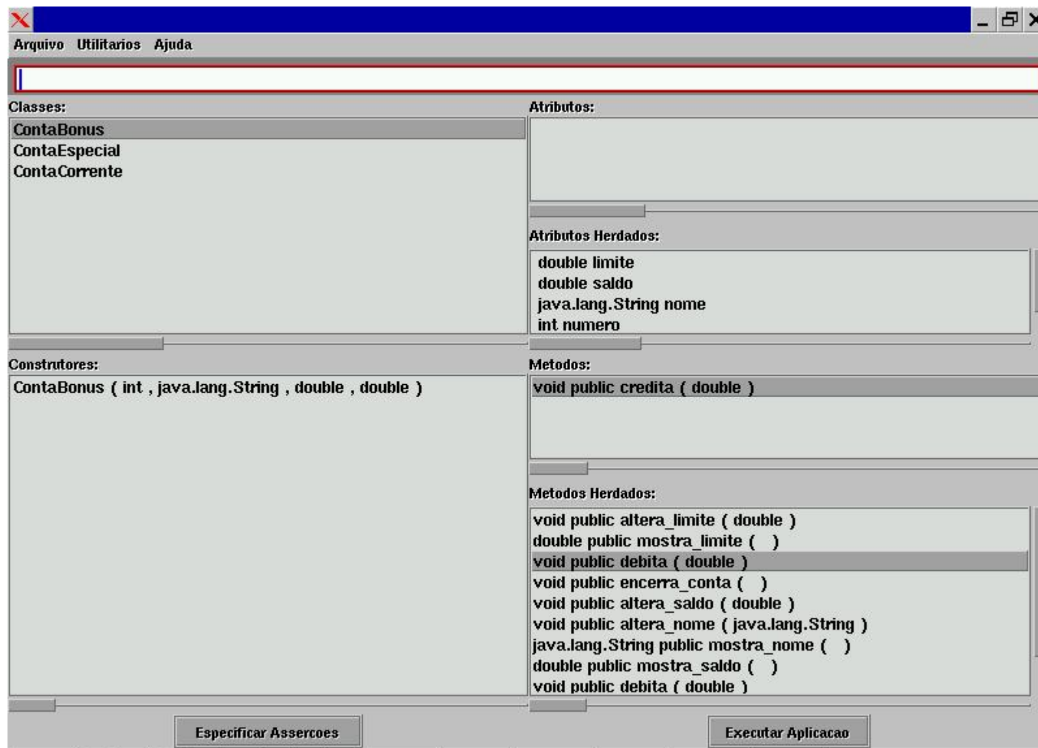


FIGURA 4.23 – *KTest* – classes e métodos escolhidos para monitoramento

A invariante especificada para a classe escolhida é:

(numero > 0 && nome != null && saldo > (limite * -1) && limite > 0)

De acordo com esta invariante, em nenhum momento um objeto da classe *ContaBonus* poderá possuir como valores de seus atributos um *numero* menor ou igual a zero, um *nome* nulo, um *saldo* menor que o limite estipulado para essa conta, nem um *limite* igual ou inferior a zero.

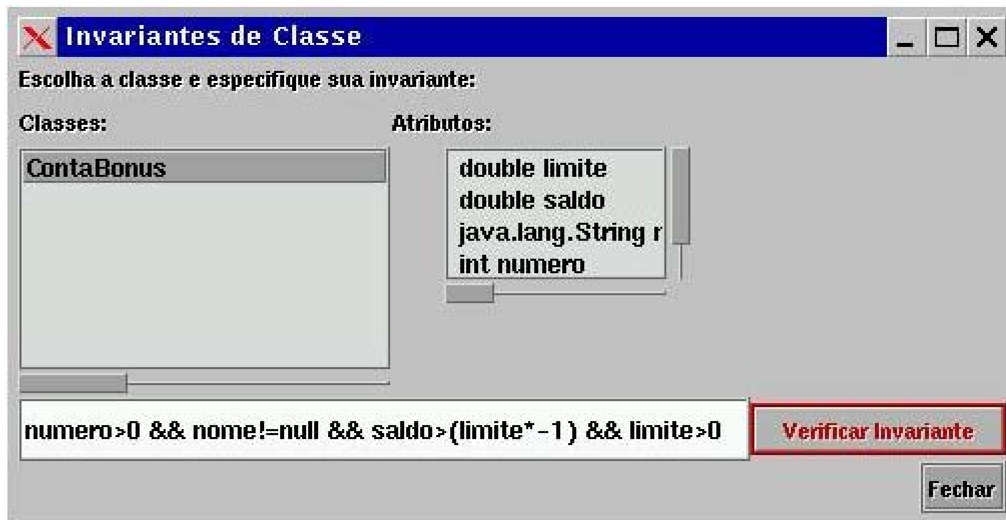


FIGURA 4.24 – *KTest* – janela para especificação de invariantes

Ao pressionar a opção "*Verificar Invariante*", três operações distintas são executadas: primeiro, assegurar que a asserção digitada corresponde a uma expressão lógica válida em Java. Segundo, verificar se os atributos referenciados na asserção correspondem aos atributos declarados na classe em questão e, por último, realizar o armazenamento da asserção nas estruturas presentes no meta-nível, para que possa ser recuperada e analisada no momento oportuno.

A próxima etapa consiste na especificação das pré e pós-condições dos métodos selecionados. Para o método *credita*, responsável por acumular o valor na variável *saldo* do objeto da *ContaBonus*, foi determinado a seguinte pré-condição:

(contaencerrada = = false)

Desta forma, nenhuma operação de crédito pode ser realizada em um conta que esteja na situação de encerrada. Já sua pós-condição não foi estabelecida, permanecendo como *null*, pois conforme mencionado, a especificação das asserções não é obrigatória. A opção "*Verifica Pré e Pós*" leva à execução das mesmas operações citadas anteriormente: verificação da sintaxe das pré e pós-condições, certificação com relação aos nomes dos atributos utilizados na asserção, e ao armazenamento nas estruturas do meta-nível.

É apresentado na fig. 4.25 a pré-condição escolhida para o método *credita*.

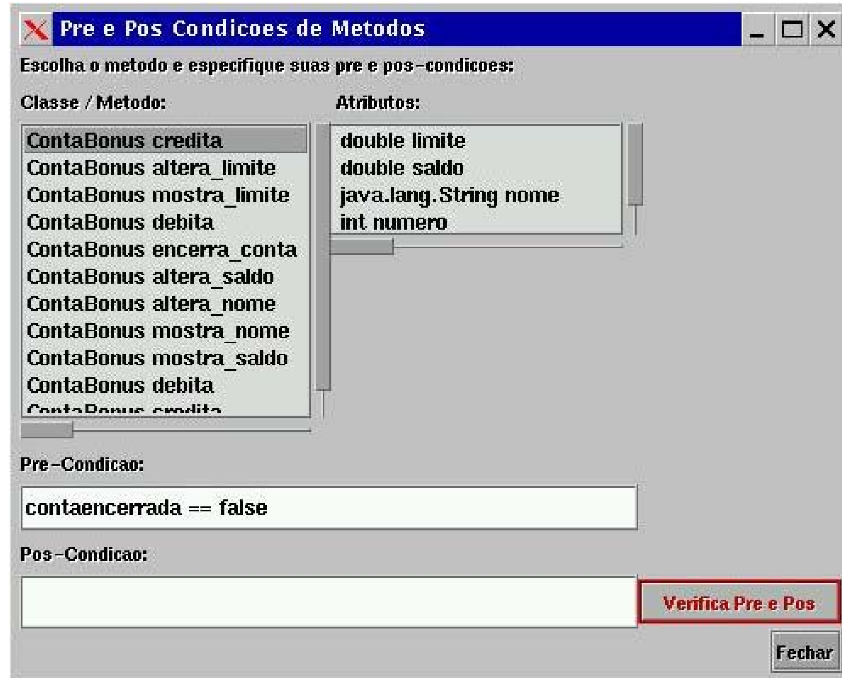


FIGURA 4.25 – *KTest* – Pré e pós-condições do método *credita*

A fig. 4.26 apresenta a especificação da pré e pós-condição do método *debita*, sendo sua pré-condição igual a do método *credita* explicada anteriormente.

Assim, nenhuma operação de débito pode ser realizada sobre uma conta que esteja assinalada como encerrada. Já sua pós-condição é definida como:

*(saldo > (limite * -1))*

Com esta expressão, após a execução do método *debita*, o saldo deve permanecer maior que o valor do limite.

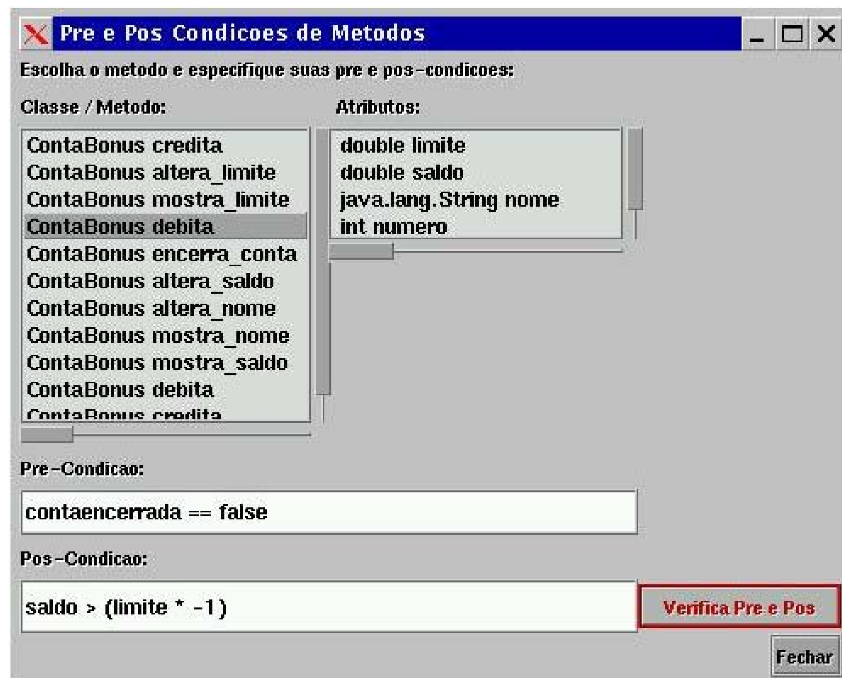


FIGURA 4.26 – *KTest* – Pré e pós-condições do método *debita*

4.3.1 Execução da Aplicação: ativação da reflexão

Após a escolha das classe e métodos, bem como da especificação das asserções, é necessário indicar qual das classes que compõem a aplicação será executada primeiramente, a fim de dar início ao processo de ativação da reflexão. De qualquer forma, não é essencial que a classe que será inicializada primeiramente tenha sido previamente selecionada para teste.

Associado à primeira classe a ser executada, será instalado um meta-objeto primário que receberá a notificação de qualquer objeto por ela instanciado, ficando sobre sua responsabilidade a decisão da propagação da meta-configuração para as demais classes. Esta propagação, entretanto, deve ser realizada somente para objetos de classes previamente selecionadas para teste, ou seja, somente devem ser instalados meta-objetos sobre objetos de classes escolhidas para monitoramento. Com esta medida, logra-se um considerável ganho de desempenho, devido ao menor número de meta-objetos instanciados, diminuindo assim o número de interceptações e, conseqüentemente, obtendo-se uma atenuação no *overhead* entre meta-nível e para-nível.

A opção "*Executar Aplicação*", apresentada na fig. 4.23 solicita então, ao usuário, que seja escolhida a classe para inicialização. A fig. 4.27 mostra a tela inicial da aplicação de Contas Bancárias, a qual foi executada a partir da *KTest*.

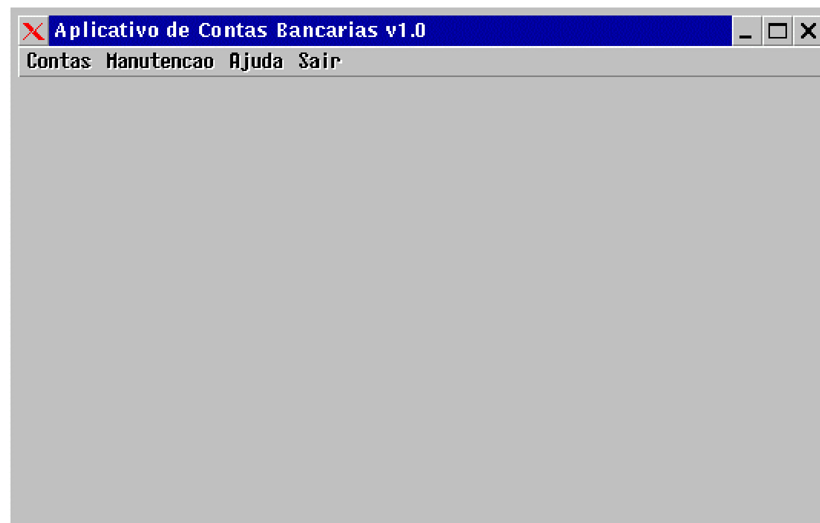


FIGURA 4.27 – Aplicação de Contas Bancárias

Com o objetivo de demonstrar o comportamento da *KTest* sobre a aplicação em questão, serão realizadas as seguintes operações na aplicação de Contas Bancárias:

a) criar um objeto do tipo *ContaBonus*, com os seguintes dados iniciais:

número → 1
nome → John Clark
saldo → $ 1500
limite → $ 800

b) creditar um valor de $ 500;

c) debitar um valor de $ 350;
d) consultar a conta;
e) creditar um valor de $ 3000;
f) debitar um valor de $ 680;
g) consultar a conta;
h) debitar um valor de $ 8000.

A figura 4.28 apresenta a criação de uma *ContaBonus* com os dados supra citados.

FIGURA 4.28 – Abertura de uma Conta Especial com Bônus

Após a execução das operações “b” e “c”, uma consulta à conta é realizada e é apresentada pela fig. 4.29, a qual é demonstra que o saldo passa a ter o valor de $1650.

FIGURA 4.29 – Consulta à Conta Especial com Bônus após operação “c”

As operações acima executadas não violaram nenhuma das asserções especificadas anteriormente. A invariante da classe *ContaBonus (numero > 0 && nome != null && saldo > (limite * -1) && limite > 0)* permanece verdadeira pois, o número da conta é maior que zero, o nome do cliente é diferente de nulo, seu saldo é maior que o limite e este limite é maior que zero. Igualmente, a pré-condição dos métodos *credita* (operação “b”) e *debita* (operação “c”) continuam verdadeiras, pois a conta não está encerrada. A pós-condição do método *debita (saldo > (limite * -1))* permanece, também, verdadeira, pois o saldo ainda é maior que o limite da conta. Estas mesmas situações podem também serem verificadas após as execuções das operações “e” e “f”,

ressaltando-se que o limite da conta agora foi alterado, pois terminada a operação "e", que consistia num crédito, o saldo da conta foi elevado para um valor superior a três vezes o valor do limite. Desta forma, de acordo com a especificação da funcionalidade desta classe, o novo limite passaria a ser a metade do saldo atual. Este novo estado é mostrado na fig. 4.30.

FIGURA 4.30 – Consulta à Conta Especial com Bônus após operação "e"

Diferentemente das operações anteriores, a operação "h" apresenta a violação de duas asserções: a pós-condição do método *debita* e a invariante da classe *ContaBonus*.

Neste ponto, o saldo da conta em questão fica negativo ($-4030), decorrente do débito de uma valor ($8000) que ultrapassa seu saldo ($3970) e limite ($2325). Quando ocorre a violação de uma asserção, conforme explanado anteriormente, a *KTest* interrompe a execução da aplicação e começa a apresentar os resultados ao usuário. Estes resultados são apresentados pelas fig. 4.31 e 4.32, respectivamente.

Neste momento, todas as vinculações existentes entre meta-nível e para-nível são desfeitas, pois as apresentações dos resultados envolvem classes previamente selecionadas para teste, as quais possuem meta-configuração. Desta maneira, evita-se a necessidade de implementação de novos mecanismos de bloqueio ao processo de recursão infinita no meta-nível, detalhado na Seção 4.2.2.

FIGURA 4.31 – Violação da Pós-Condição do método *debita* após a operação "h"

A tela "*Resultados das Asserções*" permite ao usuário verificar diretamente qual ou quais asserções foram avaliadas como falsas. Entretanto, esta janela apresenta primeiramente as opções "*Classes*" e "*Métodos*". Uma vez escolhida a classe, o usuário pode optar por verificar a avaliação da invariante desta classe após a execução do último método invocado (opção "*Invariantes*"). São apresentados o estado do objeto (seus atributos com seus respectivos valores) após a execução do último método invocado, a invariante previamente especificada, bem como o resultado de sua avaliação (verdadeiro ou falso). Caso o usuário selecione um método, poderá optar por visualizar:

a) a avaliação da pré-condição deste método (opção "*Pré-Cond*"): nesta modalidade é apresentado o estado do objeto antes da execução do método escolhido, a pré-condição previamente especificada, bem como o resultado de sua avaliação (verdadeiro ou falso)

b) a avaliação da pós-condição deste método (opção "*Pós-Cond*"): é apresentado o estado do objeto antes e após da execução do referido método, a pós-condição previamente estabelecida, bem como o resultado de sua avaliação (verdadeiro ou falso)

Já a opção "Histórico" relaciona a seqüência de ativação de todos os métodos, de todas as classes escolhidas para monitoramento, sendo esses métodos selecionados ou não para interceptação.

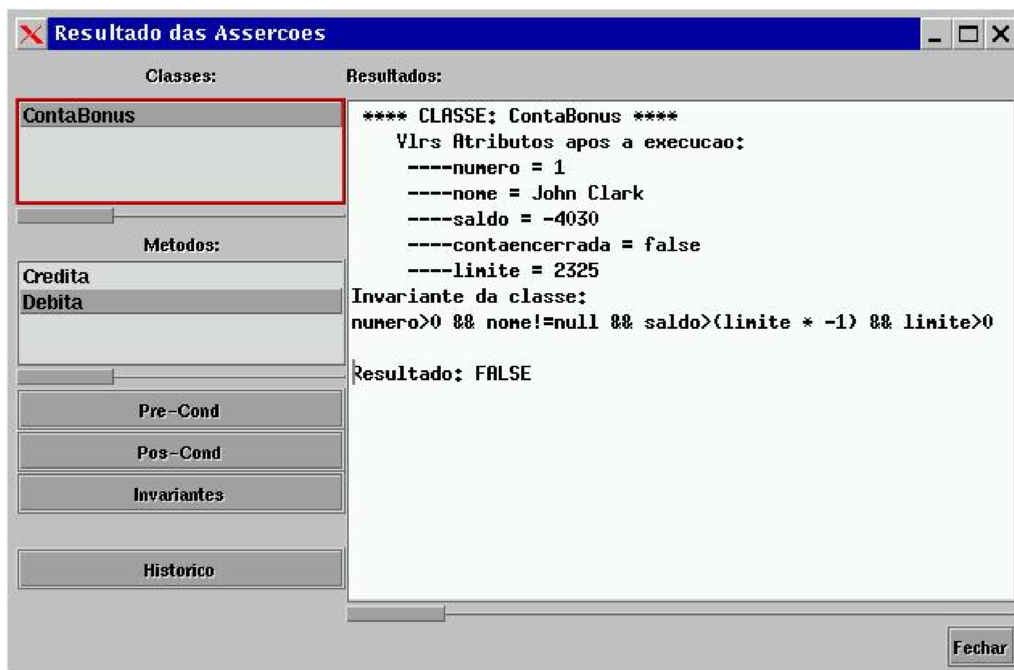


FIGURA 4.32 – Violação da invariante da classe *ContaBonus* após a operação "h"

A fig. 4.32 mostra a janela "*Resultado das Asserções*" reportando a violação da invariante de classe de *ContaBonus*. Nela pode ser constatado que o campo *saldo* tornou-se menor que o campo *limite* da conta após a execução de um de seus métodos.

Na figura 4.33 são apresentadas as informações sobre a avaliação da pré-condição do método *credita* antes da execução da operação "e". Aqui pode ser observado que a asserção é verdadeira, pois o crédito destina-se a uma conta que não possui o atributo *contaencerrada* com valor falso.

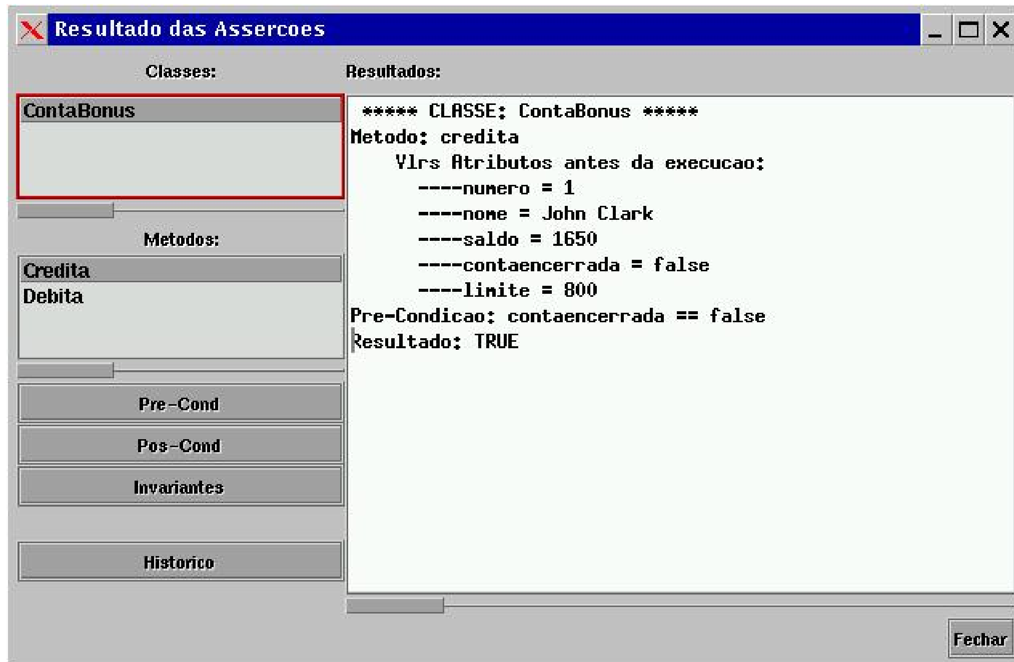


FIGURA 4.33 – Resultado da Pré-Condição do método *credita* – operação “e”

O resultado da avaliação da pré-condição do método *debita* na operação “h” é apresentado na fig. 4.34, tendo esta validação retornado verdadeira, pois o débito é feito sobre uma conta que não está encerrada.

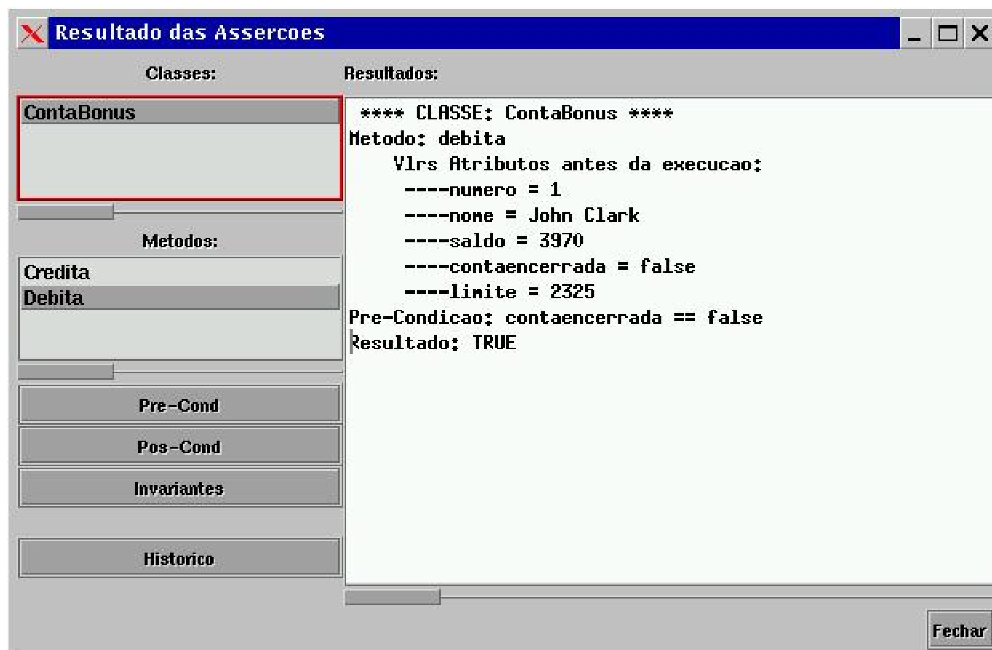


FIGURA 4.34 – Resultado da Pré-Condição do método *debita* – operação “h”

Finalmente, o usuário pode obter a relação dos métodos chamados durante a execução da aplicação em teste.

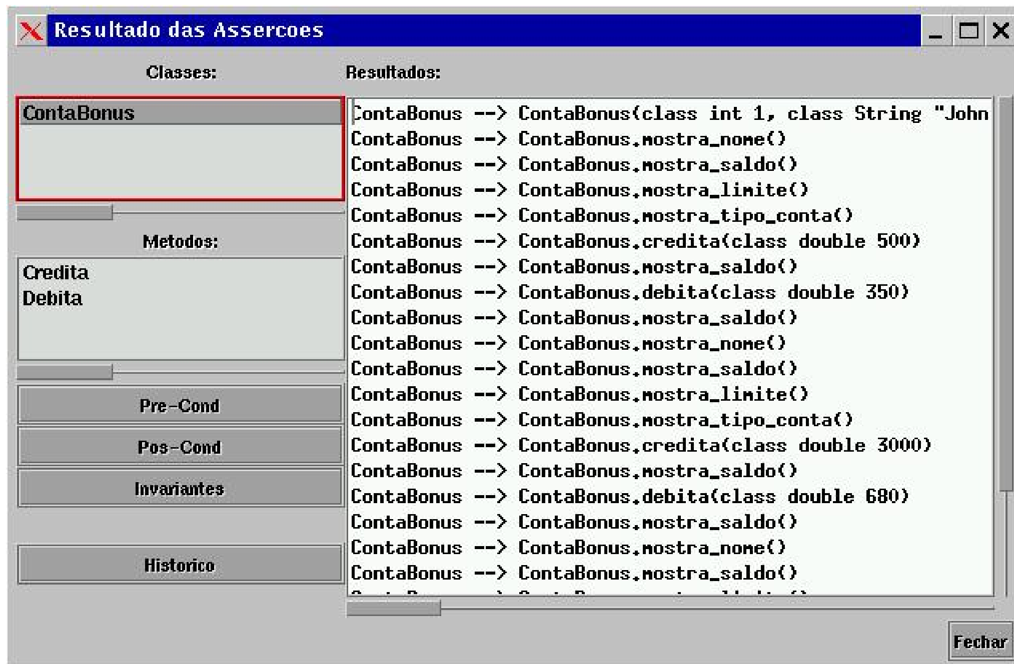


FIGURA 4.35 – Histórico dos métodos de *ContaBonus*

Esta relação contém todos os métodos das classes escolhidas para monitoramento. Para isto, basta escolher a opção “*Histórico*” na janela dos resultados das asserções, conforme mostra a fig. 4.35. Esta relação apresenta os métodos na exata ordem em que foram invocados, fornecendo informações adicionais sobre seus parâmetros, tais como tipo e valor. Como pode-se notar, a seqüência de invocação corresponde fielmente à seqüência de operações a que a aplicação Contas Bancárias foi submetida. A chamada aos métodos *mostra_saldo*, *mostra_nome*, *mostra_limite* e *mostra_tipo_conta* correspondem as operações “d” e “g”, pois ambas caracterizavam operações de consultas sobre a conta.

Todos os resultados acima apresentados podem ser armazenados em arquivos apontados pelo usuário, sob forma de ficar documentado os resultados obtidos quando da submissão de aplicações à *KTest*. Tal especificação pode ser realizada na opção “*Utilitários*”, no menu principal da ferramenta.

5 Conclusões

A Automação da fase de teste de *software* constitui-se numa relevante área de pesquisa, pois verifica-se um constante crescimento da complexidade das aplicações desenvolvidas. Neste sentido, o uso de ferramentas de teste pode contribuir de forma significativa à realização deste processo. Ferramentas podem ser construídas para apoiar diversas técnicas de teste de software, buscando-se atingir um grau de confiabilidade cada vez maior nas aplicações desenvolvidas.

O paradigma OO tem contribuído para a reutilização de soluções, evidenciando que o teste deve ser realizado com a finalidade de garantir que defeitos presentes nestas soluções sejam descobertos impossibilitando, assim, sua propagação em reutilizações de futuras aplicações. Apesar do paradigma OO surgir com o objetivo de melhorar a qualidade e produtividade no desenvolvimento de software, algumas de suas características tornam a atividade de teste mais complexa do que em outras abordagens. Entre essas características cita-se a herança, o encapsulamento, o polimorfismo e a ligação dinâmica [UNG 97].

A utilização da reflexão computacional contribui de maneira significativa no processo de teste, permitindo que se monitore uma aplicação em tempo de execução, sem a necessidade da instrumentação do código-fonte.

A API de reflexão da linguagem Java permite somente a realização de reflexão estrutural, não possibilitando reflexão comportamental. A reflexão comportamental permite, através de interceptação de mensagens entre objetos, uma monitoração das interações destes, tornando possível a verificação da integridade de objetos cujas classes tenham sido escolhidas para teste, não sendo necessária a instrumentação do código-fonte da aplicação.

A ferramenta *Ktest,* apresentada nesta dissertação, aplica a reflexão comportamental para auxiliar o teste de programas escritos em Java. *KTest* utiliza o protocolo de reflexão *Guaraná* para a monitoração de interações entre os objetos, verificando a integridade dos objetos, consultando os valores de seus atributos, analisando se os estados dos objetos são válidos, de acordo com o emprego de asserções. Assim, alguns tipos específicos de erros podem ser detectados e consequentemente corrigidos, garantindo uma maior qualidade do produto de *software*.

5.1 Principais contribuições

Este trabalho apresentou, primeiramente, um estudo sobre Teste de *Software* Orientado a Objeto, especificamente o teste baseado em estados, apoiado pela ferramenta desenvolvida, pela utilização de asserções como mecanismo de auxílio a este tipo de teste.

Foram também apresentados os principais conceitos que norteiam a reflexão computacional, sua arquitetura, modelos e estilos de reflexão. Sob forma de viabilizar a implementação da ferramenta desenvolvida, quatro protocolos reflexivos disponíveis para a linguagem Java foram analisados: *Corejava*, *MetaJava*, *OpenJava* e *Guaraná*. Este estudo resultou na escolha do protocolo reflexivo Guaraná, por apresentar uma

série de características apropriadas ao tipo de teste escolhido para implementação. O protocolo reflexivo Guaraná foi, neste trabalho, apresentado com um maior nível de detalhamento quanto a sua arquitetura, estrutura e dinâmica.

O desenvolvimento da ferramenta *KTest* representa a principal contribuição do presente trabalho. Realizando a análise dos estados dos objetos de forma dinâmica, ou seja, durante a execução da aplicação em teste através da utilização de reflexão computacional, a ferramenta *KTest* objetiva fornecer apoio às atividades de teste e validação de aplicações escritas na linguagem Java

Através da interceptação de mensagens e da especificação de asserções, estas especificadas pelo usuário (testador) da ferramenta, na forma de invariantes de classe, pré e pós-condições, é possível verificar os valores dos atributos dos objetos da aplicação em teste. *Ktest* possibilita também, armazenar a seqüência de métodos chamados pelos objetos da aplicação em teste, tornando possível ao testador, visualizar o histórico de interações no para-nível.

Os resultados dos testes são informados ao usuário da *KTest* quando do final da execução da aplicação ou, quando da violação de qualquer uma das asserções especificadas pelo testador. Ocorrendo uma violação, são apresentados os valores dos atributos dos objetos antes e após a execução dos métodos, assim como uma completa relação da seqüência de ativação das mensagens (métodos). Esses resultados podem ser armazenados em arquivos, para posterior comparação a novas submissões.

5.2 Extensões Futuras

Ressalta-se, que é possível estender a ferramenta *KTest* adicionando-se novas funcionalidades, de forma a expandir seu campo de atuação. Entre essas extensões, cita-se:

- inclusão de novas técnicas de teste, permitindo a detecção de novas categorias de possíveis erros;
- implementação em outra linguagem de programação, dependendo das características reflexivas desta;
- geração de gráficos que demonstrem, por exemplo, métodos e/ou variáveis mais acessados;
- preparação para o teste de aplicações reflexivas: estender a ferramenta para a instalação de uma torre reflexiva, ou seja, um segundo meta-nível, fazendo com que este meta-nível passe a controlar o meta-nível da aplicação reflexiva.
- preparação para o monitoramento de aplicações distribuídas: o MOP Guaraná não possui ainda suporte explícito para distribuição. Os meta-níveis, neste caso, são distintos e precisam se comunicar explicitamente para atuar de forma coordenada. De acordo com Senra [SEN 2001], existem planos para criar meta-objetos que oferecam distribuição no nível-base. Entretanto, não existem planos ainda de distribuição do próprio meta-nível.

Bibliografia

[BER 99] BERTAGNOLLI, S. C. **Taxonomia de Protocolos de Reflexão Computacional e sua Aplicação em Java**. 1999. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BIN 94] BINDER, R. V**. The FREE Approach to Testing Object-Oriented Software**: An Overview. Chicago: RBSC Corporation, 1994. Disponível em: <http://www.rbsc.com/pages/FREE.html>. Acesso em: 12 dez. 1999.

[BIN 95] BINDER, R. V. **Testing Object-Oriented Systems**: A Status Report. Chicago: RBSC Corporation., 1994. Disponível em: <http://www.rbsc.com/pages/site_map.html>. Acesso em: 12 dez. 1999.

[BIN 99] BINDER, R. **Testing Object-oriented systems**: models, patterns, and tools. New York: Addison Wesley, 1999.

[BOO 94] BOOCH, G. **Object-Oriented Analysis and Design**. Califórnia: The Benjamin Cummings, 1994.

[CAM 97] CAMPO, M. R. **Compreensão Visual de Frameworks através da Introspeção de Exemplos**. 1997. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CAZ 98] CAZZOLA, W.; SOSIO, A.; TISATO, F**. Reflection and Object-Oriented Analysis**. Italy: DISCO – Department of Informatics, Systems and Communications, University of Milano, 1998. Disponível em: <ftp://ftp.disi.unige.it/pub/person/CazzolaW/OORaSE99/095-106 Cazzola.ps.gz>. Acesso em: 01 dez. 1999.

[CHI 99] CHIBA, S.; TATSUBORI, M. **A Yet Another java.lang.class**. Japan: Institute of Information Science and Electronics, University of Tsukuba, 1999. Disponível em: <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/>. Acesso em: 05 jan. 2000.

[CHO 78] CHOW, T. S. Testing Software Design Modeled by Finite-State Machines. **IEEE Transactions on Software Engineering**, New York, v.4, n.3, p.178-187, 1978.

[COR 98] CORNELL, G.; HORSTMANN, Cay S. **Core Java**. São Paulo: Markon Books, 1998.

[DEM 87] DEMILLO, R. A. **Software Testing and Evaluation**. New York: The Benjamin/Cummings Publishing Company, 1987.

[EIS 97] EISENSTADT, M. My Hairiest Bug War Stories. **Communications of the ACM**, New York, v.40, n.4, p.30-37, 1997.

[FER 89] FERBER, J. Computational Reflection in Class Based Object-Oriented Languages. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE, 1989, New York. **Proceedings...** New York: ACM Press, 1989. p.317-326.

[FOO 93] FOOTE, B. Architectural Balkanization in the Post-Linguistic Era. In: WORKSHOP ON OO REFLECTION AND META-LEVEL ARCHITECTURES – OOPSLA, 1993, Washington. **Proceedings...** Washington: ACM Press, 1993. p.1-9.

[FUN 2001] FUNK, Nathan. **JEP – Java Expression Parser**. 2001. Disponível em: < http://jep.sourceforge.net/ >. Acesso em: 01 set. 2001.

[GOL 97] GOLM, M. **Design and Implementation of a Meta Architecture for Java**. Erlangen-Nürnberg: Diplomarbeit im Fach Informatik, Friedrich-Alexander Universität, Jan. 1997.

[GRA 89] GRAUBE, N. Metaclass compability. **SIGPLAN Notices**, New York, v.24, n.10, p.305-315, Oct. 1989.

[GRE 97] GREEN, D. **The Java Tutorial**: A practical guide for Programmers. Palo Alto, Califórnia: Sun MicroSystems, 1997. Disponível em: <http://java.sun.com/docs/books/tutorial/index.html>. Acesso em: 05 dez. 1999.

[HER 97] HERBERT, J. S. **Teste e Depuração de Software Orientado a Objetos**. 1997. Exame de Qualificação (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HER 97b] HERBERT, J. S. **Teste de Software Orientado a Objetos Baseado em Estados**. 1997. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[HER 99] HERBERT, J. S.; PRICE, A. M. de A. Técnicas e Ferramentas de Teste para a Linguagem Java. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 3., 1999, Porto Alegre, RS. **Anais...** Porto Alegre: SBC, 1999.

[HER 99b] HERBERT, J. S.; PRICE, A. M. de A. Utilizando Ferramentas no Processo de Teste de Software OO: Gerenciamento e Distribuição de Tarefas. In: CONGRESSO NACIONAL DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 19., 1999, Rio de Janeiro. **Anais...** Rio de Janeiro: SBC, 1999.

[HET 91] HETZEL, B.; GELPERIN, D. Software Testing: Some Troubling Issues. **American Programmer**, New York, v.4, n.4, Apr. 1991.

[IPL 96] IPL INFORMATION PROCESSING. **An Introduction to Software Testing**. Bath, UK, 1996.

[KIC 92] KICZALES, G.; des RIVIERES, J; BOBROW, D. G. **The Design and Implementation of Meta-Object Protocols**. Cambridge: MIT Press, 1991.

[KLE 96] KLEINÖDER, J.; GOLM, M. **MetaJava: An Efficient Run-Time Meta Architecture for Java**. Erlangen-Nürnberg, Germany: Computer Science Department, Friedrich-Alexander-University, 1996.

[KLE 97] KLEINÖDER, J.; GOLM, M. **MetaXa and the Future os Reflection**. Germany: Computer Science Department 4, University of Erlangen-Nürnberg, 1997.

[KUN 91] KUNG, C. **The Object-Oriented Paradigm**. Arlington: Computer Sience Engineering, Department of University of Texas, Nov. 1991.

[KUN 95] KUNG, D. et al. Developing an Object-Oriented Software Testing and Maintenance Environment. **Communications of the ACM**, New York, v.38, n.10, Oct. 1995.

[LIS 96] LISBOA, M. L. B.; RUBIRA, C. M. F. Técnicas de Programação para Tolerância a Falhas. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 1., 1996, Belo Horizonte, MG. **Anais...** Belo Horizonte:UFMG, 1996.

[LIS 97] LISBÔA, M. L. B. Arquiteturas de meta-nível. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 11., 1997, Fortaleza. **Anais...** Fortaleza: SBC, 1997.

[LIS 98] LISBÔA, M. L. B. **Reflexão computacional no modelo de objetos**. Porto Alegre: CPGCC da UFRGS, 1998.

[MAE 87] MAES, P. Concepts and experiments in computational reflection. In: OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS CONFERENCE - OOPSLA, Orlando, 1987. **Proceedings...** Orlando: ACM Press, 1987.

[MAE 88] MAES, P. Issues in Computational Reflection. In: MAES,P.; NARDI, D. (Ed). **Meta-Level Architecture and Reflection**. Amsterdam: [s.n.], 1988.

[MAL 98] MALDONADO, J. C. et al. **Aspectos Teóricos e Empíricos de Teste de cobertura de Software**. 1998. Notas dos autores.

[MAR 95] MARTIN, J.; ODELL, J. **Análise e Projeto Orientado a Objetos**. São Paulo: Makron Books, 1995.

[MCG 96] McGREGOR, J. D. Testing Object-Oriented Components. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1996, Linz. **Tutorial Notes**. Berlin: Springer-Verlag, 1996.

[MYE 79] MYERS, G. **The Art of Software Testing**. New York: John Willey & Sons, 1979.

[NAK 92] NAKAJIMA, S. What makes a language reflective and how ? In: INTERNATIONAL WORKSHOP ON NEW MODELS FOR SOFTWARE ARCHITECTURE / REFLECTION AND META-LEVEL ARCHITECTURES, 1992, Tokyo, Japan. **Proceedings...** [S.l.:s.n.], 1992. p.125-136.

[NIE 97] NIEMEYER, P.; PECK, J. **Exploring Java**. 2 ed. Sebastopol: O'Reilly & Associates, 1997. 500 p.

[OLI 98] OLIVA, A.; BUZATO, L. E.; GARCIA, I. C. **The Reflexive Architecture of Guaraná**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 10 jan. 2000.

[OLI 98b] OLIVA, A.; BUZATO, L. E. **Guaraná: A Tutorial**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 10 jan. 2000.

[OLI 98c] OLIVA, A.; BUZATO, L. E. **Composition of Meta-Objects in Guaraná**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 10 jan. 2000.

[OLI 98d] OLIVA, A.; BUZATO, L. E. **The Desing and Implementation of Guaraná**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 10 jan. 2000.

[OLI 98e] OLIVA, A.; BUZATO, L. E. **An Overview of MOLDS – A Meta-Object Livrary for Distributed Systems**. Campinas: Instituto de Computação, Universidade Estadual de Campinas – UNICAMP, 1998. Disponível em: <http://www.dcc.unicamp.br/~oliva/guarana>. Acesso em: 10 jan. 2000.

[PAL 2000] PALAVRO, I. **Ferramenta de Teste de Aplicações Orientada a Objetos Baseada em Estados**. 2000. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[PER 2000] PERRY, William E. **Effective Methods for Software Testing**. New York: John Wiley & Sons, 2000.

[PIN 96] PINTO, I. M. Teste de Sistemas Orientados a Objetos. In: SEMANA ACADÊMICA DO CPGCC. Porto Alegre: CPGCC da UFRGS, 1996.

[PIN 98] PINTO, I. M.; PRICE, A. M. de A. Um Sistema de Apoio ao Teste de Programas Orientados a Objetos com uma Abordagem Reflexiva. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 12., 1998, Maringá. **Anais...** Maringá: Depto. de Informática, Universidade Estadual de Maringá, 1998.

[PIN 98b] PINTO, I. M. **Um Sistema de Apoio ao Teste de Aplicações Smalltalk**. 1998. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[PRE 95] PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

[ROS 95] ROSENBLUM, D. S. A practical approach to programming with assertions. **IEEE Transactions on Software Engineering,** New York, n.21, p.19-31, 1995.

[RUB 97] RUBIRA, C. M. F.; BUZATO, L. E. Estruturação de Sistemas Orientados a Objetos Tolerantes a Falhas usando Reflexão Computacional. In: ESCOLA REGIONAL DE INFORMÁTICA, 5., 1997. Santa Maria, RS. **Anais...** Santa Maria: SBC, 1997.

[RUB 98] RUBIRA, C. M. F.; SILVA, R. C.; CORREA, S. L.; BUZATO, L. E. Reflexão Computacional em Linguagens de Programação: Um Estudo Comparativo. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 2., 1998. Campinas. **Anais...** Campinas: UNICAMP, 1998.

[RUM 97] RUMBAUGH, J. et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Campus, 1997.

[SEN 2001] SENRA, R. D. A. **Programação Reflexiva sobre o MOP Guaraná**. 2001. Dissertação (Mestrado em Ciência da Computação) - Instituto de Computação, Universidade Estadual de Campinas, Campinas.

[SIE 92] SIEGAL, S. M. **Strategies for Testing Object-Oriented Software**. 1992. Compuserve CASE Forum Library.

[SIL 2000] SILVEIRA, F. F. **Utilização de Reflexão Computacional no Teste de Softwares Desenvolvidos em Java**. 2000. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[SIL 97] SILVA, R. C.; BUZATO, L. E; CORRÊA, S. L.; RUBIRA, C. M. F. Reflexão Computacional em Linguagens de Programação: um estudo comparativo. In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 2., 1997, Campinas, SP. **Anais...** Campinas: UNICAMP, 1997.

[SMI 92] 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.

[STE 94] STEEL, L. Beyond objects. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 8., 1994, Bologna, Italy, 1994. **Proceedings...** Berlin: Springer-Verlag, 1994. p.1-11.

[SUN 2000] SUN MICROSYSTEMS AND WEBGAIN. **Java Compiler Compiler™ (JavaCC) - The Java Parser Generator**. 2000. Disponível em: <http://www.webgain.com/products/java_cc/>. Acesso em: 01 maio 2001.

[TAT 99] TATSUBORI, M. **OpenJava Tutorial**. Japan: Institute of Information Science and Electronics, University of Tsukuba, 1999. Disponível em: <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/>. Acesso em: 21 dez. 1999.

[TAT 99b] TATSUBORI, M. **An Extension Mechanism for the Java Language**. Master of Engineering Dissertation, Graduate School of Engineering, University of Tsukuba, Japan, 1999. Disponível em: <http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/>. Acesso em: 21 dez. 1999.

[UNG 97] UNGAR, D. et al. Debugging and the Experience of Immediacy. **Communications of the ACM**, Mountain View, CA, v.40, n.4, p.38-43, 1997.

[VER 97] VERGÍLIO, S. R. et al. Aumentando a Eficácia dos Critérios Estruturais Através da Utilização de Critérios Restritos. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia, SP. **Anais...** Águas de Lindóia: ICMSC/USP, 1997.

[VOA 97] VOAS, J. How Assertions Can Increase Test Efectivenes. **IEEE Software**, Los Alamitos, v.14, n.2, p.118-120, Mar./Apr. 1997.

[WON 94] WONG, W. E. et al. Constrained Mutation in C Programs. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 8., 1994, Curitiba, PR. **Anais...** Curitiba: SBC, 1994.