Um Sistema de Apoio ao Teste de Programas Orientados a Objetos com uma Abordagem Reflexiva

<Ivete Martins Pinto>[1] *<Ana Maria de Alencar Price>*[2]

[1] <Fundação Universidade do Rio Grande>
<Centro de Processamento de Dados>
<cpdipm>@cpd.furg.br
<Rio Grande>, <RS>, <Brasil>

[2] <Universidade Federal do Rio Grande do Sul>
< CPGCC - Instituto de Informática >
<anaprice>@inf.ufrgs.br
<Porto Alegre>, <RS>, <Brasil>

Resumo

O artigo descreve um protótipo de ferramenta de apoio ao teste de programas orientados a objetos, chamado *ATeste*, fundamentado no conceito de teste baseado em estado, com uma abordagem reflexiva, para auxiliar o teste de aplicações Smalltalk. O teste baseado em estado verifica as interações entre os objetos pela monitoração das mudanças que ocorrem em valores dos atributos. A abordagem reflexiva utilizada possibilita que mecanismos de análise de objetos possam ser associados à aplicação, através do uso de técnicas de reflexão computacional baseadas no conceito de gerenciadores de meta-objetos. Desta forma, a análise da aplicação não interfere nos aspectos comportamentais e estruturais dos objetos do domínio, permitindo a monitoração de objetos específicos selecionados pelo usuário, e apresentando como principal contribuição uma infra-estrutura para capturar informações, realizar verificações, e monitorar a execução das aplicações, de forma dinâmica, sem necessidade de alteração do código fonte, graças à composição de um conjunto de meta-objetos coordenados por um gerenciador.

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

Abstract

This paper presents a prototype of a tool, named ***ATeste***, which aims to support object oriented testing of Smalltalk applications by implementing the concept of state-based testing through a reflexive approach. In state-based testing objects interactions are verified through monitoring changes that occur to the values of object attributes. The reflexive approach allows the association of analysis mechanisms to the software system through the application of computational reflection concepts. Computational reflection provides dynamic analysis of the software, without interfering in both behavior and structural aspects of the object's domain. It allows monitoring of specific objects, which can be selected by the user. In this way, one of the main characteristics of the developed system is the analysis of the application under test in a dynamic way, without necessity of instrumenting its source code. This is possible due to computational reflection that allows the implementation of particular analysis mechanisms that can be dynamically associated with the application in execution.

Key-words: Object Orientation, Computational Reflection, Software Test, Object Oriented Software Test, Smalltalk Applications Test.

1. Introdução

Um dos problemas, constantemente citado quando se discute teste de software, é o alto custo. Segundo [PRE95], a atividade de teste é um elemento crítico da garantia de qualidade de software, e pode consumir até 40 % do esforço despendido no desenvolvimento de software. Por este motivo, o teste de software tornou-se, pouco a pouco, um tema de grande importância com a necessidade da aplicação de métodos práticos que assegurem a qualidade dos produtos finais, a fim de torná-los confiáveis e de fácil manutenção.

As técnicas e métodos baseados em Orientação a Objetos surgiram trazendo um enfoque substancialmente diferente dos métodos tradicionais. Uma grande vantagem da abordagem Orientada a Objetos é que esta adota formas mais próximas dos mecanismos humanos para gerenciar a complexidade inerente ao software, como a abstração, classificando elementos (objetos) em grupos (classes), através de uma estrutura hierárquica [COA93]. Neste paradigma, o mundo real é visto como consistindo de objetos autônomos, concorrentes, que interagem entre si, e cada objeto tem seu próprio estado e comportamento, semelhantes a seu correspondente no mundo real [KUN91]. É discutível se o teste orientado a objetos é essencialmente diferente do teste convencional, entretanto, não é questionável que a realização de testes de códigos OO é mais complexa, principalmente, em função da hierarquia de classes, do polimorfismo, da ligação dinâmica de mensagens, e da herança.

Este artigo apresenta um protótipo de ferramenta, ***ATeste***, que tem como objetivo auxiliar o desenvolvedor no teste de códigos OO. ***ATeste*** foi implementado em Smalltalk-80, e fornece auxílio para o teste de aplicações Smalltalk através da análise dinâmica da aplicação. Uma das principais características do sistema desenvolvido é a análise da aplicação de forma dinâmica, sem necessidade de instrumentação de seu código fonte. Isto é possível através da utilização de reflexão computacional, que possibilita a intervenção na computação da aplicação para a monitoração de objetos específicos, os quais podem ser selecionados pelo usuário em tempo de execução. Para facilitar a utilização deste conceito, ***ATeste*** foi construído usando como base o framework *Luthier MOPs* [CAM97], projetado para suportar a construção de ferramentas de análise dinâmica de programas através de técnicas de reflexão computacional baseadas em meta-objetos. Os conceitos de teste de programas OO baseado em estados [BIN95a] [TUR93] [TUR93a] [TUR93b] foram de grande valia para o desenvolvimento do protótipo proposto. O teste baseado em estados é uma técnica onde os objetos são caracterizados por estados, que são determinados pelos valores armazenados em cada uma das variáveis de instância destes objetos.

ATeste trabalha com um meta-nível, que monitora os objetos solicitados pelo usuário da aplicação do nível-base. Foi implementado um conjunto de meta-objetos que monitoram a execução das aplicações, capturando as informações requeridas para posterior visualização e análise. As classes selecionadas pelo usuário são refletidas em meta-objetos. e, sempre que uma mensagem é enviada a uma classe refletida, esta mensagem é interceptada e o estado do objeto antes e após a execução do método ativado é apresentado ao usuário, em tempo de execução, juntamente com outras informações relevantes, tais como o nome do método ativado, da classe que recebeu a mensagem, da classe que enviou a mensagem, e da classe que implementa o método. Além disto, ***ATeste*** também apresenta uma visão do caminho de execução da aplicação, até o momento de uma dada interceptação, com todas as informações já descritas acima. Isto faz com que o usuário tenha um conhecimento detalhado dos eventos executados pela aplicação.

2. Teste de Software OO

Embora a abordagem Orientada a Objetos apresente várias vantagens em relação ao paradigma procedimental, a realização de testes é um dos maiores problemas encontrados no desenvolvimento de códigos OO, pois, ironicamente, ao mesmo tempo em que algumas características próprias das linguagens orientadas a objetos tentam reduzir alguns tipos de erros, podem favorecer a adição de novos riscos de defeitos [BIN95]. Entre as facilidades introduzidas por este paradigma, pode-se citar o tamanho dos métodos, que tendem a ser pequenos, com algoritmos menos complexos, facilitando a localização de defeitos, e o encapsulamento, que previne muitos dos problemas causados por um escopo de dados sem controle.

Uma diferença importante do teste de programas procedimentais em relação aos modelos orientados a objetos, encontra-se no fato que as aplicações OO não são executadas seqüencialmente. Como nenhuma ordem de invocação dos métodos é especificada explicitamente, a análise do fluxo de controle ou do fluxo de dados para o teste de classe é dificultada, fazendo com que técnicas de teste estrutural não sejam diretamente aplicáveis. Segundo [KUN95] os problemas adicionais para o teste e manutenção de programas introduzidos pelo paradigma podem ser resumidos *no entendimento de aplicações OO*, devido às características de encapsulamento, polimorfismo e ligação dinâmica, *na interdependência complexa*, causada pelos relacionamentos complexos que existem em um programa OO tais como herança, agregação, associação, criação dinâmica de objetos, polimorfismo, e outros, *no teste de comportamento dependente de estado*, pois os objetos possuem comportamentos dependentes de seu estado, isto é, o efeito de uma operação em um objeto depende de seu estado e pode alterar este estado, e *na carência de ferramentas de suporte ao teste* visto que o desenvolvimento de ferramentas para suporte ao teste OO ainda é imaturo e o uso de técnicas convencionais não atendem plenamente aos problemas específicos de teste OO.

Segundo Turner e Robson [TUR93], a menor unidade de teste para sistemas Orientados a Objetos é a classe. Estes autores recomendam que seja realizado inicialmente o teste das classes que não possuem dependentes, e então, recursivamente, o teste das classes/objetos que usam seus serviços, juntando desta forma os testes de unidade/integração. A estratégia de teste proposta por estes autores é denominada *Teste Baseado em Estados*. A ênfase do teste baseado em estados está nos valores armazenados na representação dos objetos, consistindo no teste das interações em uma classe pela monitoração das alterações dos valores dos atributos em um objeto, que conseqüentemente alteram seu estado.

Entre as propostas encontradas na área de teste OO, podemos também citar as propostas apresentadas por Fiedler, Binder e Harrold/McGregor, entre outras. Fiedler [FIE89] apresenta uma proposta onde foram aplicadas técnicas de teste tradicionais para softwares escritos em C++. As classes são validadas na ordem determinada por um grafo de chamada, sendo que as classes base do grafo são validadas primeiro, da mesma forma que na proposta de Turner, sem a necessidade de classes *driver* que emulam classes parametrizadas não validadas. Binder [BIN96] apresenta FREE (Flattened Regular Expressions), um framework para o desenvolvimento de casos de teste para softwares OO, com uma técnica baseada em estados, dirigindo-se a vários escopos de teste, tais como: teste de classes baseado na implementação e baseado na representação, teste de *clusters*, de subsistemas e de sistemas. Harrold e McGregor [HAR92] apresentam uma técnica de teste incremental, explorando a natureza hierárquica da relação de herança para testar grupos de classes relacionadas, reutilizando as informações de teste das superclasses para guiar o teste das subclasses.

3. Reflexão Computacional

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

A reflexão computacional define uma arquitetura em níveis, denominada arquitetura reflexiva. Em uma arquitetura reflexiva, um sistema computacional é visto como incorporando dois componentes: um representando o objeto, e outro a parte reflexiva. As computações do objeto, localizado no nível base, tem por objetivo resolver problemas e retornar informações sobre o domínio externo (domínio da aplicação), enquanto o nível reflexivo, localizado no meta-nível, resolve os problemas e retorna informações sobre as computações do objeto, podendo adicionar funcionalidades extras a este objeto. A Figura 3.1 permite observar uma arquitetura reflexiva, composta por um meta-nível, que contém a reflexão do sistema objeto e um nível-base, onde ficam os objetos do domínio da aplicação. Os dados do nível base são usados no meta-nível para a realização de computações reflexivas, que podem interferir nas computações de nível-base.

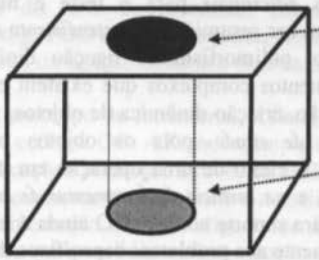

Figura 3.1 - Arquitetura Reflexiva

A computação reflexiva pode atuar tanto sobre a estrutura do sistema objeto, ocorrendo a nível de classes, quanto sobre o comportamento, quando ocorre a nível de objetos. Quando a computação reflexiva ocorre a nível de classes, o meta-nível é composto por meta-classes, que contém informações sobre os aspectos estruturais do nível-base. Se a computação reflexiva ocorre a nível de objetos, o meta-nível é composto por meta-objetos, os quais contém informações sobre os aspectos do comportamento dos objetos (instâncias das classes) do nível-base. A principal diferença entre a reflexão estrutural e comportamental é que a primeira realiza a associação de classes a classes de objetos (meta-classes), enquanto a segunda realiza a associação de objetos a objetos do meta-nível (meta-objetos). A Figura 3.2 mostra como se estruturam as classes na reflexão comportamental, utilizada em ***ATeste***.

O conceito de meta-objetos foi introduzido como um mecanismo para suportar comportamento reflexivo em linguagens orientadas a objetos [MAE87]. O comportamento computacional de cada objeto que faz parte de uma aplicação é determinado por um meta-objeto que controla e define a operação deste objeto. Novos comportamentos podem ser adicionados, ou a forma como os métodos são executados pode ser modificada. A reflexão computacional permite o controle do comportamento do objeto ao receber uma mensagem, possibilitando a verificação de informações sobre o processo de execução, com o objetivo de monitorá-la, podendo modificar o curso desta, se necessário. A conexão é realizada ligando-se um objeto a um meta-objeto. Sempre que o objeto receber uma mensagem, o meta-objeto

associado a intercepta e passa a realizar o tratamento da mensagem, dependendo de informações dinâmicas, obtidas durante a computação. O meta-objeto realiza a transformação de atributos do programa em dados disponíveis ao próprio programa (reificação[1]), e realiza as computações no meta-nível. Os objetos reificados constituem as meta-informações sobre as quais são realizadas as computações reflexivas.

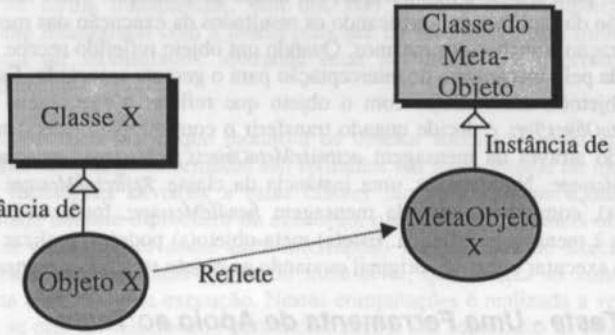

Figura 3.2 - Reflexão Comportamental

3.1 Framework Luthier MOPs

O framework *Luthier MOPs* (Meta-Object Protocol) [CAM96] [CAM97], desenvolvido em Smalltalk-80, foi utilizado como base para a construção de ***ATeste*** por suas características de apoio à construção de ferramentas de análise dinâmica de programas. Este framework foi projetado para suportar a construção deste tipo de ferramentas através de técnicas de reflexão computacional baseadas em meta-objetos. Neste modelo de reflexão, os meta-objetos definem, implementam, dão suporte e participam da execução da aplicação do nível base, permitindo implementar diferentes mecanismos de análise que podem ser dinamicamente associados com a aplicação analisada.

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

Entre as principais contribuições de *Luthier MOPs* pode-se mencionar o conceito de gerenciadores de meta-objetos, que provê facilidades para a associação de meta-objetos com classes, instâncias ou métodos específicos, e suporte para múltiplos meta-objetos com prioridade de ativação. Um gerenciador de meta-objetos age como um mediador entre o nível

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

base e os meta-objetos, contendo a informação necessária para determinar qual meta-objeto, se houver algum, deve ser ativado quando uma mensagem é refletida.

A utilização deste framework por ***ATeste*** acontece através da implementação de um conjunto de meta-objetos específicos, que são, de forma dinâmica, ligados às classes selecionadas pelo testador, para análise e monitoração, em tempo de execução. Estes meta-objetos monitoram a execução das aplicações, verificando os resultados da execução das mensagens, para posterior visualização e análise dos mesmos. Quando um objeto refletido recebe uma mensagem, esta é desviada pelo mecanismo de interceptação para o gerente associado. Este gerente verifica o(s) meta-objeto(s) associado(s) com o objeto que refletiu a mensagem através da mensagem *findMetaObjectFor:* e decide quando transferir o controle para este(s) meta-objeto(s), o que é realizado através da mensagem *activateMetaObjects:forMessage:*, enviando-lhe(s) a mensagem *handleMessage:*. Na ativação, uma instância da classe *ReflectedMessage* é enviada ao(s) meta-objeto(s), como parâmetro da mensagem *handleMessage:*, fornecendo-lhe(s) as informações relativa à mensagem refletida. Este(s) meta-objeto(s) pode(m) realizar computações próprias, e então executar o método original enviando ao objeto refletido a mensagem *send*.

4. ATeste - Uma Ferramenta de Apoio ao Teste

ATeste é um protótipo de ferramenta para auxílio ao teste de programas orientados a objetos, desenvolvido para uso em aplicações Smalltalk, com a finalidade de prover facilidades ao desenvolvedor de software nas atividades relacionadas à análise de códigos OO. Para o desenvolvimento de ***ATeste*** foi utilizado o ambiente VisualWorks, um ambiente totalmente orientado a objetos para construção de aplicações na linguagem de programação Smalltalk-80. ***ATeste*** implementa uma estratégia de teste, denominada ***teste dinâmico de caminho***, a qual aplica conceitos de teste baseado em estados [BIN96] [BIN95a] [TUR93] [TUR93a], aliados às vantagens da reflexão computacional. A reflexão computacional possibilita a análise dinâmica da aplicação através de mecanismos que podem ser associados à aplicação em tempo de execução, sem necessidade de alterar o código fonte da mesma. O comportamento de uma classe é definido por um determinado conjunto de valores encapsulados que a classe possui em determinado momento, e controlado por valores encapsulados, seqüências de mensagens, ou ambos. Os métodos de uma classe fornecem o comportamento desejado pela interação com a representação de dados, e o teste baseado em estados testa as interações monitorando os efeitos que os métodos tem no estado de um objeto.

4.1 Características de ATeste

ATeste tem como objetivo auxiliar o teste de aplicações Smalltalk, através da monitoração dinâmica de métodos executados para análise do estado dos objetos. ***ATeste*** intercepta o envio de mensagens quando o objeto receptor pertence a uma classe cuja reflexão foi solicitada. Suas principais funcionalidades são: captura e apresentação do estado dos objetos receptores das mensagens interceptadas (valores de suas variáveis de instância) "antes e após a execução dos métodos selecionados para monitoração", juntamente com outras informações tais como nome do método que implementa a mensagem interceptada, objeto que enviou a mensagem e objeto que recebeu a mensagem; apresentação de uma lista de todas as mensagens enviadas até um determinado momento aos objetos monitorados, contendo adicionalmente, o estado destes objetos no momento da execução, o nome do método que implementa a mensagem interceptada, objeto que enviou a mensagem e o objeto que recebeu a mensagem.; e permissão ao testador de informar os possíveis estados do objetos antes e após a execução dos métodos selecionados, para que a verificação dos resultados seja feita automaticamente.

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

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

A estratégia aqui proposta é realizada de forma iterativa, realizando a monitoração da execução da aplicação para as classes e métodos selecionados pelo usuário. Em uma primeira etapa, o usuário de ***ATeste*** interage com a ferramenta, devendo informar a(s) categoria(s)[2] que deseja monitorar. A partir desta informação o sistema recupera a lista de classes que compõem cada categoria para que o usuário selecione somente aquelas que deseja monitorar. Os nomes dos métodos encapsulados nestas classes são mostrados para que o testador tenha ainda a possibilidade de selecionar apenas aqueles métodos mais críticos que deseja refletir. A reflexão das classes selecionadas é ativada e a informação sobre os métodos que devem ser monitorados armazenada.

Na segunda etapa, a monitoração é realizada executando-se a aplicação em análise através da ativação de um determinado método e inicializando-se os objetos com um estado inicial, sendo que próximos estados são determinados pelo próximo método a ser chamado a partir do estado atual. A ativação deste primeiro método faz com que uma seqüência de mensagens seja ativada. Quando uma classe cuja reflexão foi solicitada, recebe uma mensagem, o estado do objeto é verificado, antes e após a execução do método ativado, para que o testador compare o estado atual (gerado durante a execução da aplicação) com o estado esperado (estados definidos na especificação). A partir destas informações são apresentados, na seqüência de execução, o estado dos objetos antes e após a execução dos métodos selecionados para monitoração. Adicionalmente, é fornecida uma visão de todo o caminho de execução da aplicação, com o estado dos objetos antes e após a execução de todos os métodos ativados pelas mensagens recebidas por objetos que pertencem às classes refletidas.

4.2 Estrutura Geral

A arquitetura do protótipo compõem-se de três grupos de classes: classes próprias, que implementam as funcionalidades do protótipo; classes especializadas a partir do framework *Luthier MOPs*, necessárias para implementar a reflexão computacional desejada, e classes do

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

próprio ambiente VisualWorks/Smalltalk. Na Figura 4.1 são mostrados os três grupos de classes através de um diagrama de objetos que os destaca.

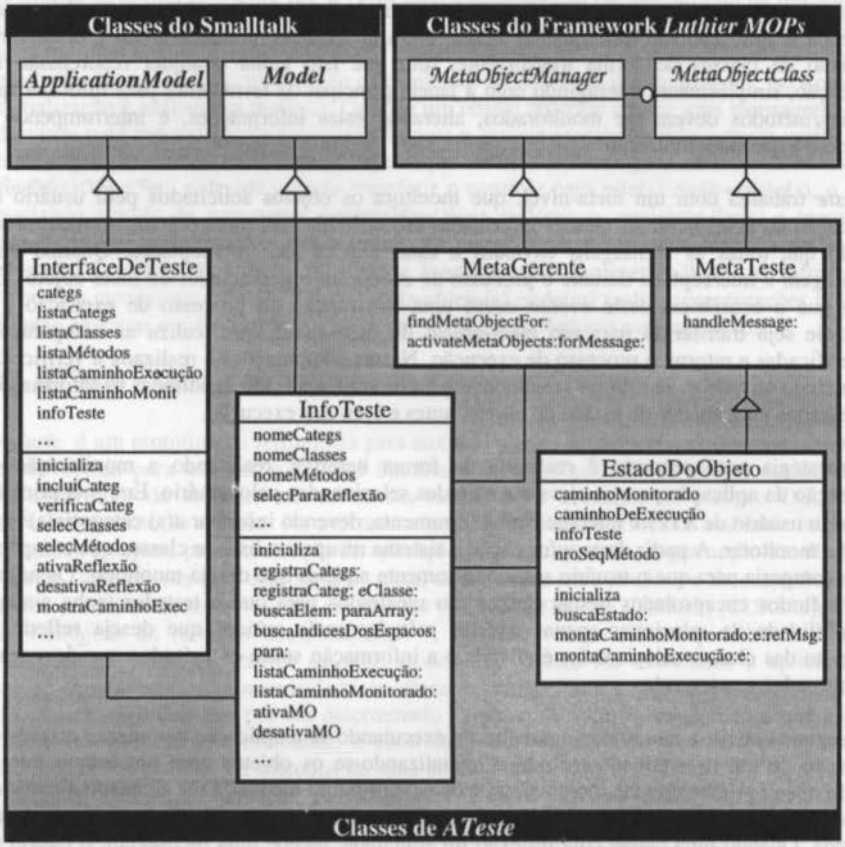

Figura 4.1 - Diagrama de Objetos de *ATeste*

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

Para utilização da estrutura do framework *Luthier MOPs,* as classes MetaObjectClass e MetaObjectManager foram especializadas com o objetivo de atender as necessidades

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

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

- InterfaceDeTeste, que é uma subclasse de ApplicationModel. Define a interface de comunicação com o usuário a fim de verificar os métodos que devem ser monitorados, os requisitos de teste e a apresentação dos resultados. A coordenação da comunicação entre os campos da interface e o modelo de dados é facilitada pela herança de atributos e do comportamento da classe ApplicationModel. As informações recebidas do usuário são tratadas em interfaces (janelas) desenvolvidas no próprio VisualWorks, através da ferramenta Canvas, que auxilia o desenvolvimento de desenhos de telas para interação com o usuário.
- InfoTeste, que fornece suporte para administrar as informações de entrada e saída, definindo a interface com as classes que necessitam destas informações para a interceptação de mensagens, associação de meta-objetos e execução da monitoração. InfoTeste é uma subclasse de Model.
- MetaGerente, é uma classe especializada da classe abstrata MetaObjectManager (framework *Luthier MOPs*), que implementa os mecanismos específicos de associação e ativação de meta objetos através dos métodos *findMetaObjectFor:* e *activateMetaObjects:forMessage:*.
- MetaTeste, que provê a implementação do método *handleMessage:* para tratar as mensagens refletidas, sendo uma especialização da classe abstrata MetaObjectClass do framework *Luthier MOPs*. O método *handleMessage:* define as ações que devem ser tomadas quando uma mensagem é interceptada, ativando métodos da classe EstadoDoObjeto, que realizam as verificações necessárias. Este método é ativado pela classe MetaGerente quando uma mensagem é refletida, sendo enviado como argumento uma instância da classe ReflectedMessage, contendo as informações acerca da mensagem refletida.
- EstadoDoObjeto, a qual recebe da classe MetaTeste as informações acerca da mensagem interceptada e, a partir das informações sobre o objeto receptor, busca o estado deste objeto. A funcionalidade básica desta classe é a implementação das atividades de auxílio ao teste, através da monitoração do estado dos objetos e da transformação dos atributos do objeto em dados disponíveis para visualização. Esta classe implementa o *teste de caminho* do objeto selecionado para análise se o método ativado pela mensagem interceptada foi selecionado para monitoração, montando uma lista de mensagens refletidas contendo o estado do objeto neste momento. É responsável também pela geração de uma lista com o caminho completo da aplicação monitorada, ou seja, de todas mensagens recebidas pela(s) classe(s) sob teste, mesmo que estas não ativem métodos selecionados para monitoração, com os respectivos estados dos objetos receptores das mensagens.

4.3 Funcionamento de ATeste para um Estudo de Caso

Para demonstrar o funcionamento de ***ATeste***, é apresentado um estudo de caso. A monitoração de estados é realizada sobre objetos de classes da categoria Examples-VWTutorial, que agrupa as classes Check, Checkbook, e CheckbookInterface, as quais implementam uma aplicação bancária de controle de cheques. As principais transações desta aplicação são: depositar valores (método *deposit:*), eliminar cheques selecionados da lista de cheques (método *cancelCheck:*), e gravar novos cheques emitidos na lista de cheques (método *recordCheck:*).

Para monitorar uma aplicação, o usuário de ***ATeste*** deve inicialmente fornecer ao sistema as informações referentes à(s) categoria(s) que agrupam as classes que implementa(m) esta aplicação. O primeiro passo é ativar a ferramenta no ambiente VisualWorks. Este ambiente encontra-se disponível tanto para estações de trabalho Sun como para uso em microcomputadores PC. Após a ativação de ***ATeste***, é apresentada a janela principal de comunicação do protótipo, a qual é mostrada na Figura 4.2.

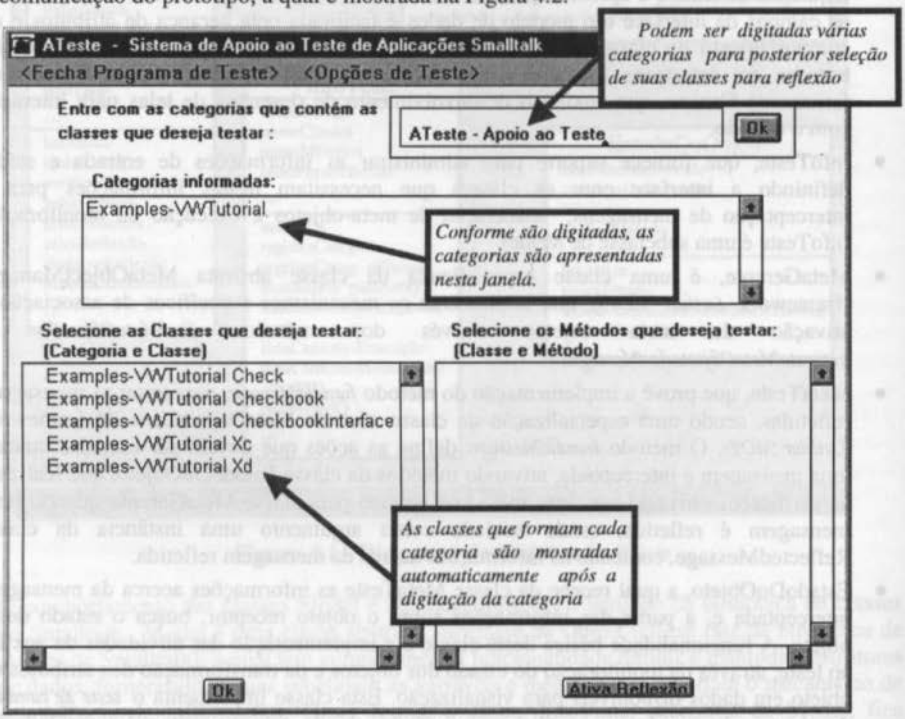

Figura 4.2 - Janela de Comunicação Principal de *ATeste*

Com a ativação da ferramenta inicia-se a primeira etapa do processo de teste, que refere-se à interação do usuário para fornecer informações para ativação da reflexão. Deve-se então digitar o nome da categoria no campo de entrada de categorias. Após a verificação da categoria, todas as classes agrupadas nesta categoria são recuperadas e adicionadas à lista de classes desta mesma janela, na forma de pares categoria/classe para identificar a que categoria pertence cada classe. A(s) classe(s) que se deseja monitorar são selecionadas, e é realizada a

busca de todos os métodos encapsulados na(s) classe(s) selecionada(s), sendo estes mostrados na janela de comunicação na forma de pares classe/método, identificando a classe a qual pertence cada método, para a seleção dos métodos que se deseja monitorar. Conforme a Figura 4.3, os métodos *recordCheck:* e *deposit:*, da classe Checkbook foram selecionados para serem refletidos. Após a seleção dos métodos, estes são armazenados e a reflexão das classes selecionadas é ativada com base nestas informações.

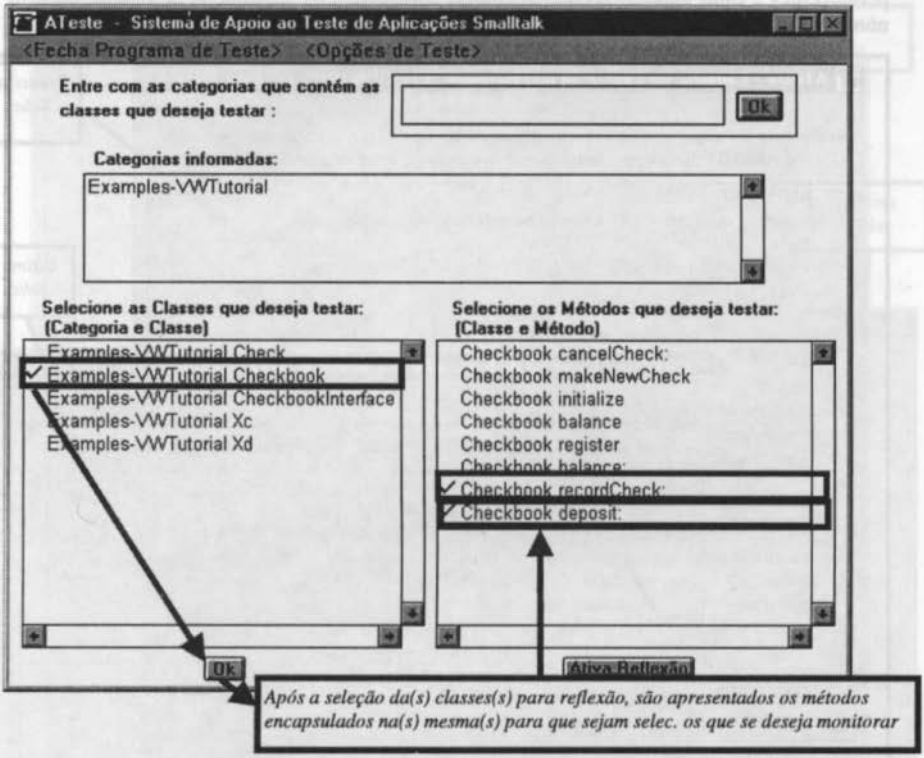

Figura 4.3 - Mostra Métodos das Classes Selecionadas

A segunda etapa independe de qualquer nova interação do usuário com a ferramenta e consiste na execução da aplicação sob teste, e na comparação do estado dos objetos monitorados com o estado esperado. Prosseguindo com a demonstração do estudo de caso, é ativada a aplicação de controle de cheques, onde as transações de depósito e de gravação de novos cheque são executadas, considerando que a classe Checkbook, e os métodos *deposit* (deposita um valor) e *recordCheck* (grava um novo cheque) estão selecionados para monitoração. A Figura 4.4 mostra o estado dos objetos antes e após a execução dos métodos selecionados, na janela "Caminho de Execução dos Métodos Selecionados para Teste". Esta janela possui o botão de ação "Caminho de Execução da Aplicação", cuja funcionalidade é apresentar a segunda janela de monitoração ao usuário. A janela "Caminho de Execução da Aplicação" (Figura 4.5) apresenta o estado dos objetos no momento de interceptação de qualquer mensagem enviada a uma classe selecionada para monitoração. Cada método monitorado gera um bloco de informações, identificando se as informações referem-se ao estado do objeto antes ou após a

execução do método ativado, e fornecendo as informações referentes à mensagem interceptada, tais como: número seqüencial, nome do método ativado pela mensagem interceptada; nome da classe receptora da mensagem; nome da classe que enviou a mensagem; e nome da classe que implementa o método ativado. O número seqüencial informa a ordem de ativação dos métodos das classes selecionadas, a qual é seqüencial e geral para todas as classes refletidas, sendo incrementada a cada mensagem recebida por objetos pertencentes a estas classes. As demais linhas apresentam os subestados do objeto, ou seja, o nome e o valor de cada variável de instância do objeto naquele momento.

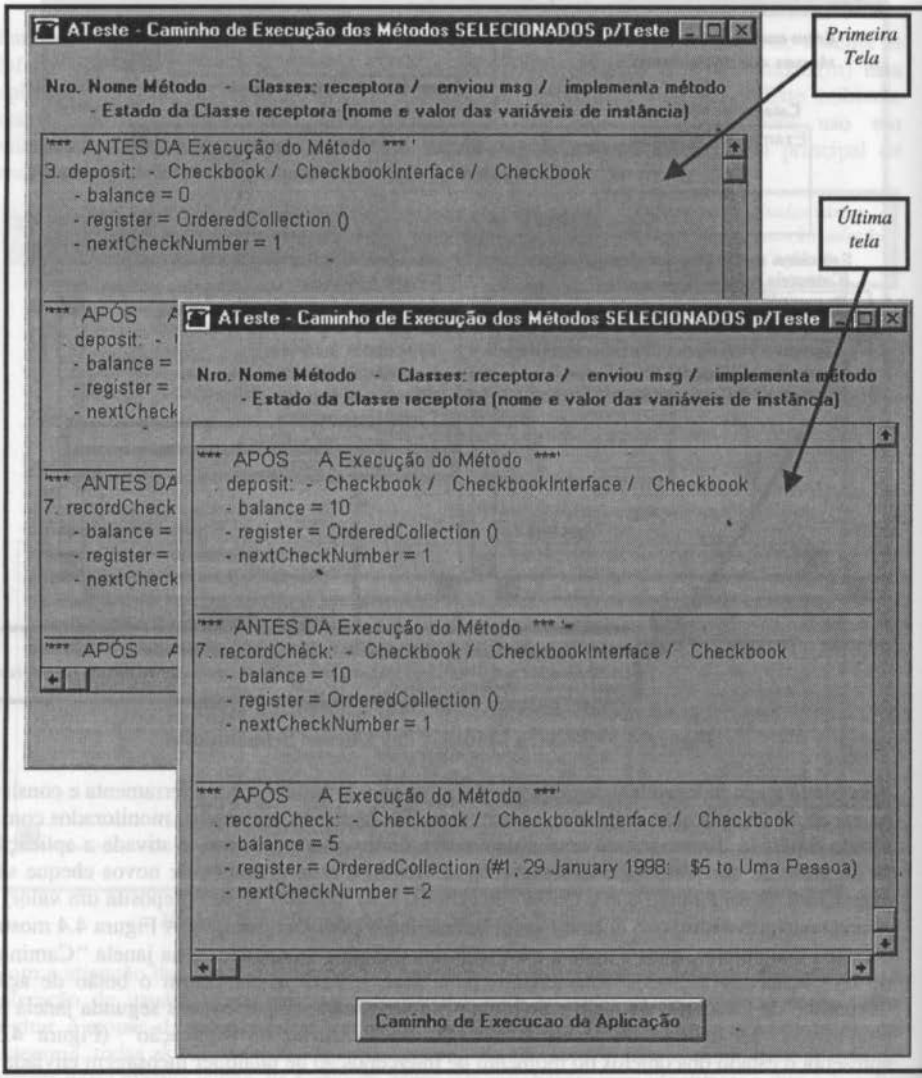

Figura 4.4 - Caminho de Execução dos Métodos Selecionados para Teste

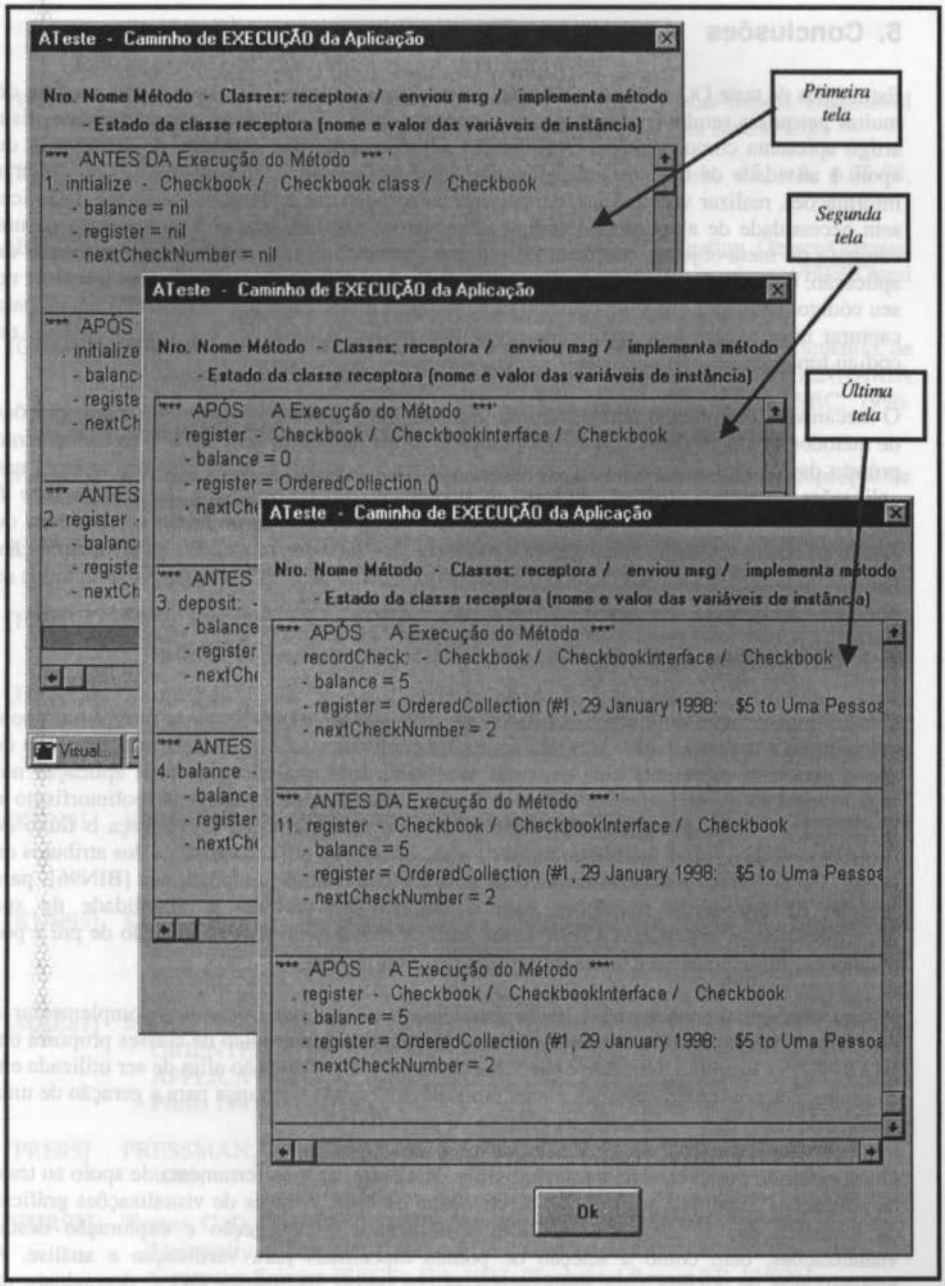

Figura 4.5 - Caminho de Execução da Aplicação

5. Conclusões

Estratégias de teste OO ainda são imaturas e a atividade de teste ainda se encontra em fase de muitas pesquisas sendo desenvolvidas, e propostas ainda não validadas completamente. Este artigo apresenta como principal contribuição a proposta de um protótipo de ferramenta de apoio à atividade de teste de aplicações OO, que fornece uma infra-estrutura para capturar informações, realizar verificações, e monitorar a execução das aplicações, de forma dinâmica, sem necessidade de alteração do código fonte. Isto é possível graças à composição de um conjunto de meta-objetos, coordenados por um gerenciador, que monitoram a execução da aplicação, dinamicamente, realizando a análise e visualização dos resultados sem interferir no seu código, enquanto que o mecanismo utilizado pela maior parte das ferramentas atuais para capturar informações para análise de programas de forma dinâmica é a instrumentação do código fonte.

O mecanismo de reflexão computacional utilizado por ***ATeste*** possibilita que interceptações de métodos sejam estabelecidas e suprimidas de forma dinâmica, sem afetar o funcionamento próprio das classes interceptadas, eliminando assim a possibilidade de imiscuir-se erros nas aplicações durante o processo de teste. A implementação da estratégia de *teste dinâmico de caminho*, aplicando conceitos de teste baseado em estados, fornece ao testador uma visão do estado da classe sob teste, antes e após a execução dos métodos requeridos para monitoração, bem como uma visão textual de todo o caminho executado pela aplicação, mostrando todos os métodos executados e os respectivos estados, de maneira ordenada.

5.1 Extensões Futuras

Esta ferramenta apresenta um vasto horizonte de possíveis extensões para prover um apoio mais amplo à automatização da tarefa de teste de softwares OO. A geração automatizada de dados para teste representa uma extensão necessária, mas que não é de fácil aplicação nas linguagens OO, principalmente devido à ligação dinâmica de mensagens, polimorfismo e herança. Para implementar-se esta funcionalidade, é necessário que se conheça o fluxo de controle de toda a classe (incluindo todos os seus métodos) e o fluxo de dados dos atributos da classe. Estão sendo analisadas as técnicas de análise estrutural, proposta por [BIN96], para geração da árvore de transições com o objetivo de verificar a viabilidade de sua implementação, a fim de que ***ATeste*** possa auxiliar o desenvolvedor na geração de pré e pós condições, automatizando a tarefa de geração de casos de teste.

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

Outra extensão possível, seria a transformação de ***ATeste***, de uma ferramenta de apoio ao teste de aplicações Smalltalk, para um ambiente visual de teste. Através de visualizações gráficas da dinâmica das aplicações, poderia disponibilizar-se a navegação e exploração destas visualizações, bem como a seleção de pontos específicos para verificação e análise. A portabilidade de ***ATeste*** para implementação em outras linguagens OO é dependente da existência de facilidades reflexivas nas linguagens hospedeiras, mas pode ser estudado o uso de mecanismos específicos próprios destas linguagens para sua viabilização.

Bibliografia

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

[BIN95a] BINDER R. V. State-Based Testing. Object, New York, v.5, n.6, p.75-78, July/Aug.1995.

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

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

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

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

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

[HAR92] Harrold, M. J.; McGregor, J. D.; Fitzpatrick, K. J. Incremental Testing of Object-Oriented Class Structures. In: International Conference on Software Engineering, 14., 1992, Melbourne, Austrália. Proceedings... Los Alamitos: IEEE, 1992. p.68-80.

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

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

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

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

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

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

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