Uma ferramenta baseada em aspectos para o teste funcional de programas Java

André Dantas Rocha,* Adenilso da Silva Simão,
José Carlos Maldonado, Paulo Cesar Masiero

{rocha,adenilso,jcmandon,masiero}@icmc.usp.br

[1]Laboratório de Engenharia de Software
Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo
Av. do Trabalhador São-Carlense, 400 – Centro – Cx. Postal 668
São Carlos – São Paulo – CEP 13560-970

Abstract. *Availability of testing tools provides better quality and more productivity for the testing activities. In this paper, we present a tool for functional test of programs Java. The tool, denominated J-FuT, supports various functional testing criteria and offers coverage analysis based in those criteria. Aspect-oriented Programming is used in the phases of instrumentation and execution of the testing criteria. The use of this technique allows a clear separation between the testing code and the program code and is an effective and elegant solution for this type of tool. Furthermore, it makes easy adding and removing the aspects that implement the tests.*
Keywords. *Aspect Oriented Programming, Functional Testing, Test Coverage.*

Resumo. *A disponibilidade de ferramentas de teste de software propicia maior qualidade e produtividade para as atividades de teste. Neste artigo é apresentada uma ferramenta para teste funcional de programas Java. A ferramenta, denominada J-FuT, apóia os principais critérios da técnica funcional e oferece análise de cobertura baseada nesses critérios. A Programação Orientada a Aspectos é utilizada pela ferramenta nas etapas de instrumentação e execução dos critérios de teste. O uso dessa técnica permite a separação clara entre o código de teste e o código do programa, assim como torna mais fácil a adição e remoção dos aspectos que implementam o teste.*
Palavras-chave. *Programação Orientada a Aspectos, Teste Funcional, Análise de Cobertura.*

1. Introdução

Teste de software é uma etapa fundamental do ciclo de desenvolvimento e representa uma importante premissa para alcançar padrões de qualidade no produto criado. O teste diz respeito à análise dinâmica do programa e consiste na execução do produto com o intuito de revelar a presença de erros.

Essa atividade envolve quatro etapas que geralmente devem ser executadas durante o processo de desenvolvimento de software: planejamento de testes, projeto de casos de teste, execução de testes (e coleta dos resultados) e avaliação dos resultados coletados. A aplicação de testes segue diversas técnicas, que oferecem perspectivas diferentes e abordam diferentes classes de erros, e que portanto devem ser utilizadas de forma complementar. As várias técnicas propostas diferenciam-se pela origem das informações utilizadas para estabelecer os

*Financiado pelo CNPq – Processo 132110/03-2

requisitos e critérios de teste, podendo-se citar as técnicas funcional, estrutural e baseada em erros. De forma geral, as técnicas estrutural e baseada em erros necessitam da estrutura do software para derivar os requisitos de teste, enquanto a técnica funcional utiliza apenas sua especificação.

Com a crescente complexidade dos softwares, torna-se fundamental a utilização de ferramentas de teste visando a automatização dessa atividade. O teste de software é, em geral, custoso e propenso a erros e ferramentas que apóiem essa atividade são essenciais. Atualmente encontram-se disponíveis diversas ferramentas que buscam atender a esse propósito, no entanto a maioria oferece suporte apenas ao teste estrutural. Além disso, as poucas ferramentas que implementam a técnica funcional desconsideram seus principais critérios e não permitem análise de cobertura apropriada.

Neste artigo é analisada a técnica funcional e apresentada uma ferramenta para teste funcional de programas Java. Ao contrário das abordagens atuais, a ferramenta oferece suporte aos principais critérios dessa técnica e permite análise de cobertura funcional. Na implementação da ferramenta utilizou-se a Programação Orientada a Aspectos – POA [13] nas etapas de instrumentação e aplicação dos critérios funcionais, pois essa técnica de programação apresenta características que facilitam o teste funcional.

O artigo encontra-se organizado da seguinte forma: na Seção 2. descreve-se a técnica funcional e seus principais critérios. Na Seção 3. são apresentados alguns conceitos pertinentes às ferramentas de teste, enfatizando-se a técnica funcional. A Seção 4. é dedicada ao AspectJ – uma linguagem orientada a aspectos para Java – e a algumas propostas existentes para aplicação da POA para o teste de software. Na Seção 5. aborda-se o projeto da ferramenta J-FuT e na Seção 6. um exemplo de seu uso. Por fim, na Seção 7., são apresentadas as considerações finais.

2. Teste funcional

Na técnica funcional (caixa-preta) os requisitos de teste são estabelecidos a partir da especificação do software, e sua estrutura não é necessariamente considerada. De acordo com Ostrand e Balcer [19], o objetivo do teste funcional é encontrar discrepâncias entre o comportamento atual do sistema e o descrito em sua especificação. Assim, consideram-se apenas as entradas e saídas e o testador não tem necessariamente acesso ao código fonte do software.

Nesse tipo de teste procura-se descobrir erros relacionados a cinco categorias [22]: (i) funções incorretas ou omitidas; (ii) erros de interface; (iii) erros de estrutura de dados ou de acesso a dados externos; (iv) erros de comportamento ou desempenho e, por fim, (v) erros de iniciação e término. No teste caixa-preta destacam-se três critérios: Particionamento de Equivalência, Análise de Valor-Limite e Teste Funcional Sistemático, descritos a seguir.

2.1. Particionamento de Equivalência

Neste critério divide-se o domínio de entrada do programa em classes de dados, dos quais os casos de teste são derivados [22]. A divisão é feita em *classes de equivalência* válidas e inválidas e busca-se produzir casos de teste que cubram diversas classes de erro e que ao mesmo tempo reduzam o total de casos necessários.

O particionamento do domínio de entrada deve ser feito de forma que para cada classe de equivalência, qualquer um dos seus elementos seja representativo de toda a classe e dessa forma o programa em teste se comporte da mesma maneira, independentemente do elemento escolhido na classe. Idealmente se um caso de teste de uma determinada classe de equivalência revela um erro, qualquer outro caso de teste nessa classe deve revelar o mesmo erro.

Segundo Myers [18], a definição das classes de equivalência é um processo heurístico e o critério exige que seja derivado um conjunto mínimo de casos de teste para exercitar as classes de equivalência válidas e um caso de teste individual para cada classe inválida. A exigência de um caso de teste específico por classe de equivalência inválida se faz necessário pois a verificação de algumas condições de entrada pode mascarar ou anular a verificação de outras.

2.2. Análise de Valor-Limite

A partir da aplicação do Particionamento de Equivalência, nota-se que grande parte dos erros tende a ocorrer nas fronteiras do domínio de entrada [18, 22]. O critério de Análise de Valor-limite complementa o Particionamento de Equivalência por meio de casos de teste que exercitam os valores limítrofes. Valores-limite referem-se a situações que ocorrem dentro de uma classe de equivalência, diretamente acima ou abaixo dela. Nesse critério, o domínio de saída do programa também é particionado e auxilia na derivação dos casos de teste, que por sua vez devem ser escolhidos em cada um dos limites das classes.

2.3. Teste Funcional Sistemático

O Teste Funcional Sistemático [14] é uma variante mais forte do Particionamento de Equivalência que leva em consideração os domínios de entrada e saída, segundo um conjunto bem definido de regras. Uma vez que o particionamento da entrada e saída é efetuado, esse critério exige que sejam exercitados ao menos dois casos de teste em cada partição, uma vez que a utilização de apenas um valor pode mascarar erros coincidentes.

Além disso, o Teste Funcional Sistemático exige a avaliação dos valores limites e de suas proximidades, e oferece regras para facilitar a identificação dos casos de teste. Nessas regras são descritos que tipos de dados devem ser selecionados para variados tipos de função, domínios de entrada e de saída, o que pode conduzir à seleção de um ou mais casos de teste, de acordo com o programa que está sendo testado.

3. Ferramentas de teste funcional

A utilização de ferramentas de teste desempenha papel fundamental na aplicação de critérios de teste. Sem o apoio de uma ferramenta, a atividade de teste torna-se trabalhosa, propensa a erros e limitada a programas simples. Além disso, como descreve Binder [3], a automatização permite a verificação rápida e eficiente das correções de defeitos, agiliza o processo de depuração, permite a captura e análise dos resultados de teste de forma consistente e facilita a execução de testes de regressão, nos quais os casos de teste podem ser reutilizados para revalidação do software após modificação.

Em geral as ferramentas utilizam o conceito de *cobertura* para medir o quanto um critério foi satisfeito em relação aos seus requisitos. Coberturas de nós, arcos e definições-uso constituem alguns exemplos utilizados pelas ferramentas [5, 26]. A análise de cobertura permite identificar áreas do programa não exercitadas por um conjunto de casos de teste e dessa forma avaliar a qualidade desse conjunto. Além disso, por meio da análise de cobertura é possível medir o progresso do teste e decidir quando finalizar essa atividade.

Embora a maioria das ferramentas existentes ofereça apoio apenas ao teste estrutural, algumas ferramentas específicas para teste funcional encontram-se disponíveis, podendo-se citar TestComplete [1], SPACES [2], Jtest [20], XDE Tester [24] e o framework JUnit [10] que, apesar de não apoiar nenhuma técnica de teste específica, pode ser utilizado para especificação e execução dos casos de teste.

Tabela 1. Comparativo entre ferramentas de teste funcional

Característica	JUnit	Jtest	TestComplete	XDE Tester	SPACES
Geração de casos de teste		✓			✓
Execução de casos de teste	✓	✓	✓	✓	✓
Especificação de classes de equivalência					
Especificação de valores-limite					
Especificação de pré e pós-condições		✓			✓
Análise de cobertura		✓			✓
Teste de interface			✓	✓	

Na Tabela 1 é exibida uma comparação entre essas ferramentas de teste, na qual enfatiza-se características importantes do teste funcional. Nota-se que nenhuma das ferramentas mostradas contempla todas as características desejáveis, e as características *Especificação de classes de equivalência* e *Especificação de valores-limite* não têm suporte. A incapacidade de especificar esses dois tipos de requisitos funcionais (classes de equivalência e valores-limite) acaba por limitar o poder das ferramentas analisadas.

A análise da tabela evidencia também que as ferramentas Jtest e SPACES são as mais completas sob o ponto de vista enfatizado. No entanto, ainda que possuam características semelhantes, é importante salientar que a ferramenta SPACES implementa as características segundo a ótica do teste funcional, enquanto a Jtest apresenta uma abordagem semelhante à técnica estrutural.

De forma geral, nenhuma das ferramentas analisadas utiliza as técnicas ou critérios difundidos pela comunidade de teste de software e muitas vezes o teste funcional é tratado apenas como teste de interface. Além disso, nessas ferramentas a quantificação da atividade de teste é feita apenas em termos de cobertura de comandos ou funções. Cabe ao testador analisar, a partir da cobertura dos comandos executados e da especificação do software, a cobertura funcional alcançada. A maioria das ferramentas analisadas também exige a disponibilidade do código fonte, característica muitas vezes não desejável no teste funcional. Assim, a técnica funcional muitas vezes é relegada em razão da falta de ferramentas que forneçam apoio direto à aplicação dos seus critérios.

4. AspectJ e teste de software

A Programação Orientada a Aspectos [13] baseia-se no conceito de separação de interesses e busca identificar interesses que estão espalhados pelo código da aplicação e implementá-los como módulos separados (denominados aspectos). Essa abordagem permite que o projeto e a codificação sejam executados de maneira estruturada, refletindo a forma como o sistema é imaginado pelos seus projetistas [9].

O AspectJ [12] é a linguagem orientada a aspectos mais difundida atualmente e constitui-se de uma extensão orientada a aspectos para a linguagem Java. Esta extensão acrescenta novas construções à linguagem, permitindo a implementação de interesses entrecortantes de forma modular. Por meio do AspectJ, o usuário pode definir pontos específicos no fluxo de execução do programa (pontos de junção (*joinpoints*)), como chamadas a métodos e criação de objetos.

Os pontos de junção podem ser agrupados em conjuntos (conjuntos de junção (*pointcuts*)), facilitando a sua referência. Quando os pontos de junção são alcançados é possível executar procedimentos específicos (adendos (*advices*)), cuja estrutura é semelhante à de um método. Os adendos podem ser executados antes, depois ou em substituição ao ponto de

junção interceptado e permitem a alteração do comportamento do programa. Informações de *runtime* referentes ao ponto de junção interceptado podem ser acessadas por meio de elementos específicos da linguagem e utilizadas no interior dos adendos.

As novas construções do AspectJ são definidas em estruturas denominadas aspectos (*aspects*), que possuem formato semelhante ao das classes e são codificadas separadamente. Os aspectos e classes são compilados por um compilador especial, que une os elementos e gera *bytecode* compatível com a linguagem Java.

```
1 aspect Rastreamento {
2   pointcut todasChamadas() : call(* *.*(..));
3
4   before() : todasChamadas() {
5     System.out.println("Método " +
6           thisJoinPoint + "invocado");
7   }
8 }
```

Figura 1. Exemplo de aspecto

O código da Figura 1, por exemplo, implementa um aspecto de rastreamento. O conjunto de junção `todasChamadas` é responsável por interceptar chamadas a qualquer método de qualquer classe. Sempre que uma chamada ocorrer, e antes da execução do método correspondente, o nome do método interceptado será impresso, como descrito pelo adendo anterior (especificado pela palavra `before`).

4.1. Abordagens existentes para teste de software

Apesar da POA ser uma abordagem recente, já existem iniciativas do seu uso para o apoio ao teste de software, utilizando principalmente a linguagem AspectJ. A separação entre código funcional e código de teste auxilia o manuseio de ambos, e a facilidade de inserir e remover aspectos na aplicação pode levar à criação de cenários de teste de forma mais rápida. Além disso, os aspectos podem ser facilmente removidos quando a atividade de teste é finalizada, conservando intacto o programa original.

Uma aplicação desse conceito é o uso de imitações virtuais de objetos, como uma alternativa aos objetos imitadores tradicionais no teste de unidade (objetos *mock*). Nessa técnica, proposta por Monk e Hall [17], nenhum objeto imitador é criado e, ao invés disso, o AspectJ é utilizado para interceptar chamadas aos métodos que estão sendo testados e retornar o valor desejado. O uso de objetos imitadores virtuais permite que os dados de teste sejam definidos no próprio teste, e que testes diferentes atribuam valores imitadores distintos para o mesmo método. Além disso, com o uso dessa técnica apenas os componentes de interesse são testados, e não é necessário instanciar objetos imitadores.

Em outra abordagem, o AspectJ foi utilizado por Bruel et al. [4] para embutir testabilidade em componentes de software. Segundo os autores, testabilidade pode ser vista como um interesse não funcional, e ser implementada sob a forma de aspectos. Nessa proposta cada caso de teste é descrito por um aspecto e o processo de combinação insere código de teste no componente, tornado-o "testável". O uso de aspectos permite embutir testabilidade em componentes de software e separar código funcional do código de teste. Utilizando as construções fornecidas pelo AspectJ é possível introduzir métodos e atributos nas classes em teste, facilitando a captura e alteração do estado do objeto, tarefa muitas vezes difícil utili-

zando apenas programação OO. Além disso, a implementação de casos de teste por meio de aspectos facilita a adição e remoção de teste ao sistema.

Mao e May [16] utilizaram a POA para o teste de integração para softwares baseados em componentes. O foco da abordagem é nos componentes adaptadores, que coordenam a interação entre os diversos componentes que compõem o sistema. Ao contrário da abordagem apresentada anteriormente, que tem o objetivo de testar unidades e verificar os contratos dos componentes, esta busca revelar uma outra categoria de erros: aqueles relacionados à comunicação entre componentes.

Além das abordagens apresentadas, a POA também tem sido aplicada para simular pseudo-controladores (*drivers*) e variar o comportamento de objetos no sistema (especialmente em condições de falha), gravar entradas e comportamento de métodos [11]. Alguns autores também têm utilizado a POA para o teste de invariantes, pré e pós-condições na técnica de Projeto por Contrato [8] e instrumentação de programas [6, 7, 21].

Apesar dos trabalhos publicados nessa área apontarem a adequação dessa técnica ao teste de programas, ainda são desconhecidas propostas que utilizem as técnicas e critérios estabelecidos pela comunidade de teste, como as técnicas estrutural e funcional e seus critérios. As iniciativas existentes enfatizam apenas as características práticas dessa atividade e as implementações, ainda que bastante úteis, limitam-se a abordagens *ad-hoc* nas quais conceitos importantes de teste de software são ignorados. Essa perspectiva, em que desconsideram-se princípios importantes de teste, pode levar a testes incompletos ou muitas vezes dispendiosos.

5. A ferramenta J-FuT

Conforme descrito anteriormente, a POA possui características que podem auxiliar na atividade de teste. Especificamente no que diz respeito ao teste funcional e ao AspectJ, percebe-se uma clara relação entre o modelo de pontos de junção adotado pela linguagem e os pontos de interesse desse tipo de teste. Rajan e Sullivan [23] referem-se aos pontos de junção do AspectJ como "pontos de junção caixa-preta", tendo em vista a sua granularidade.

Os denominados "pontos de junção caixa-preta" encaixam-se perfeitamente no tipo de teste executado pelos critérios funcionais, nos quais consideram-se principalmente as entradas e saídas do programa e não, necessariamente, sua implementação. Além disso, a captura do contexto dos pontos de junção, fornecida pelo AspectJ, oferece subsídios para analisar o comportamento interno do programa e responder a questões como:

- Dado um conjunto de casos de teste, quais classes de equivalência de uma operação[1] esse conjunto exercita?
- Dado um conjunto de casos de teste que sensibiliza uma determinada operação e um método chamado por ela, quais classes de equivalência desse método são exercitadas por esse conjunto?

Dessa forma, o uso da POA além de auxiliar o teste de sistema, permite a execução do "teste funcional de unidade", no qual considera-se o método como a unidade de teste que será analisada conforme sua especificação.

A J-FuT tem o objetivo de apoiar o teste funcional de programas desenvolvidos em Java e oferece suporte aos critérios Particionamento em Classes de Equivalência, Analise de Valor-Limite e Teste Funcional Sistemático. Além disso, permite que pré e pós-condições

[1]Operações são algoritmos executados em resposta a eventos externos ao sistema e são geralmente implementados como métodos em linguagens OO.

do sistema sejam avaliadas. A ferramenta enfatiza a análise de cobertura segundo critérios funcionais e permite a avaliação da adequação de um determinado conjunto de casos de teste a um critério específico. Utilizando a J-FuT o testador pode efetivar as atividades básicas para a aplicação de critérios funcionais: instrumentação, seleção e execução de casos de teste e, por fim, análise de cobertura.

5.1. Arquitetura

A arquitetura da ferramenta é mostrada na Figura 2, na qual observa-se a comunicação entre os vários módulos que a compõem, os casos de teste (especificados no JUnit) e o programa em teste. No projeto da ferramenta buscou-se a segmentação em camadas que refletissem as suas principais funcionalidades.

A J-FuT disponibiliza uma interface gráfica por meio da qual o testador tem acesso às funcionalidades do software. A manipulação dos casos de teste é feita por meio dessa interface, que fornece funções para execução, edição, habilitação e desabilitação de casos de teste. É importante notar que não há acesso direto ao programa em teste e sua execução sempre é feita a partir dos casos de teste implementados.

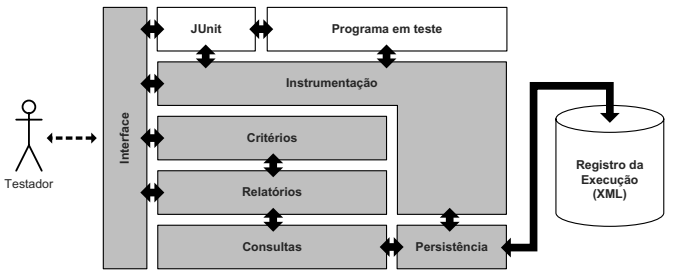

Figura 2. Arquitetura da J-FuT

Abaixo da camada que contém o programa em teste e os casos de teste encontra-se o módulo que implementa a instrumentação. Na instrumentação são utilizados aspectos, que são responsáveis por capturar operações específicas do programa em teste, além de executar os critérios funcionais. As informações necessárias para instrumentação também são fornecidas pelo testador, utilizando a interface gráfica. O relacionamento entre esse módulo e o módulo de persistência permite que os dados observados durante a execução do programa possam ser registrados. Esses dados são armazenados em um repositório XML utilizando o módulo de persistência, que implementa a interface de acesso ao registro da execução. O módulo de consulta, por sua vez, utiliza o módulo de persistência para efetuar consultas específicas sobre o repositório, usadas para a geração dos relatórios. Os relatórios são gerados de acordo com o critério de teste utilizado, que também é selecionado pelo testador por meio da interface gráfica.

O teste de uma aplicação usando a J-FuT é feito a partir de um projeto de teste, que armazena todos os dados relativos ao teste, incluindo seu nome, classes e métodos em teste, bibliotecas necessárias, requisitos de teste, condições de entrada e registro de execução. Os critérios de teste são descritos externamente por um arquivo XML. Esse arquivo contém os valores dos atributos de cada critério, assim como o critério padrão utilizado pela ferramenta.

A configuração externa permite que novos critérios possam ser criados e os critérios existentes alterados de forma simplificada. Os critérios são carregados automaticamente quando a ferramenta é iniciada e se tornam disponíveis para uso.

Um exemplo de configuração é mostrado no código da Figura 3. O elemento `criteria` agrega todos os critérios cadastrados e define o critério padrão ser utilizado. Cada critério é especificado por meio de um elemento `criterion` que, por sua vez, possui marcadores (*tags*) específicos para descrição de suas características. Os marcadores representam requisitos de teste e podem ser de cinco tipos, descritos posteriormente na Tabela 2. Cada marcador possui dois atributos, que especificam o mínimo de casos de teste necessários para exercitá-lo (`minTestsToCover`) e o máximo de requisitos que cada caso de teste pode exercitar (`maxRequirementPerTest`). O testador pode criar critérios personalizados por meio de novos elementos `criterion` contendo um ou mais desses marcadores.

```
<criteria default="Equivalence Partition">
  <criterion name="Equivalence Partition">
    <valid maxRequirementPerTest="2147483647" minTestsToCover="1"/>
    <invalid maxRequirementPerTest="1" minTestsToCover="1"/>
  </criterion>
  <criterion name="Systematic Functional Testing">
    <valid maxRequirementPerTest="2147483647" minTestsToCover="2"/>
    <invalid maxRequirementPerTest="1" minTestsToCover="2"/>
    <boundary maxRequirementPerTest="2147483647" minTestsToCover="1"/>
  </criterion>
</criteria>
```

Figura 3. Exemplo de configuração dos critérios de teste

5.2. Instrumentação e teste

A POA permite identificação de pontos específicos na execução de um programa e por isso trata-se de uma solução elegante para a sua instrumentação. Seguindo essa premissa, a J-FuT utiliza aspectos no processo de instrumentação e teste. Além de fundamental ao cálculo de cobertura, a instrumentação é essencial para o registro do comportamento do programa em teste.

Na J-FuT, a instrumentação é feita por meio de aspectos especializados definidos pelo testador. No diagrama da Figura 4 encontram-se as classes (em branco) e os aspectos (em cinza) utilizados na etapa de instrumentação. Os elementos na parte superior do diagrama pertencem à ferramenta, enquanto que aqueles na parte inferior são implementados pelo testador de acordo com a aplicação em teste. O aspecto abstrato `Instrumentation` implementa as características e comportamento básicos para a instrumentação de operações da aplicação em teste e execução de critérios funcionais e possui um conjunto de junção abstrato denominado `operationCall`, que deve ser concretizado em cada aspecto filho. Nesse conjunto de junção são definidas as regras para seleção das operações que serão testadas, e sobre ele atua um adendo substitutivo (*around*), que é responsável por exercitar os critérios escolhidos pelo testador.

Dois outros aspectos têm papéis importantes na ferramenta. O aspecto `Inheritance` modifica a hierarquia sob a classe `TestCase` (pertencente ao JUnit), introduzindo uma classe intermediária (`CustomTestCase`). Essa estrutura permite que o testador utilize normalmente o JUnit na elaboração dos casos de teste e a J-FuT utilize uma versão instrumentada dessa classe. O aspecto `PolicyEnforcement`, por sua vez, efetua verificações

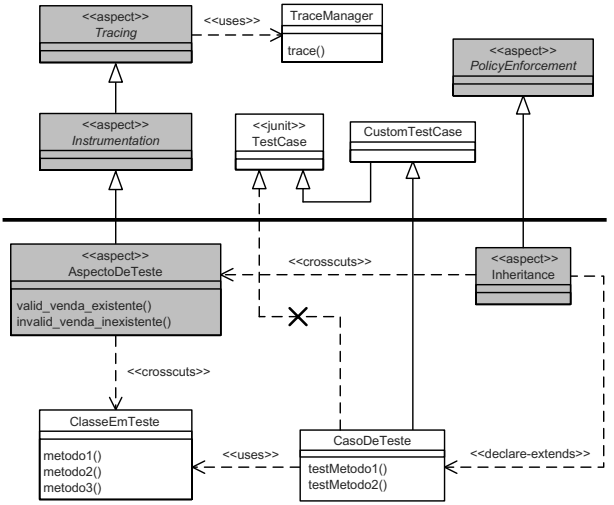

Figura 4. Principais classes e aspectos da J-FuT

sobre os aspectos implementados pelo testador (descendentes de `Instrumentation`) e observa se as regras de instanciação foram seguidas.

Tabela 2. Prefixo para os métodos de teste

Prefixo	Descrição
valid	Classe de equivalência válida
invalid	Classe de equivalência inválida
boundary	Valor-limite
pre	Pré-condição
pos	Pós-condição

A J-FuT implementa a técnica funcional por meio de predicados que representam os requisitos funcionais da aplicação em teste. Cada requisito de teste (classe de equivalência, valor-limite, pré-condição ou pós-condição) referente a uma operação é mapeado em um método (denominado método de teste) cujo nome obedece a uma nomenclatura especial, conforme descrito na Tabela 2. Os métodos de teste possuem retorno booleano e implementam a expressão lógica que avalia se os respectivos requisitos funcionais foram exercitados (retorno *true*) ou não (retorno *false*). No código da Figura 5 encontra-se um exemplo de um método de teste que representa um requisito do tipo *classe de equivalência válida* (definido por meio do prefixo "valid") e verifica se o saldo do objeto `conta` é positivo.

```
1 public boolean valid_saldoPositivo() {
2   return conta.getSaldo() >= 0;
3 }
```

Figura 5. Exemplo de método de teste

Os métodos de teste são codificados nos diversos aspectos definidos pelo testador. Cada aspecto intercepta uma operação do programa em teste e, por reflexão, executa todos os requisitos codificados. Os resultados colhidos a partir da execução desses requisitos permite avaliar a cobertura funcional do programa. O testador portanto deve implementar os aspectos de teste e os predicados que avaliam a satisfação dos requisitos funcionais. Os casos de teste, por serem especificados no JUnit, podem originar-se de outras ferramentas ou outras etapas do teste.

6. Exemplo de uso

Nesta seção apresenta-se o uso da ferramenta J-FuT em um programa exemplo, abordando a implementação dos aspectos de teste e a análise de cobertura funcional. O exemplo é baseado no programa *Identifier* [15], e seu teste é executado por meio do critério Particionamento de Equivalência. O mesmo raciocínio pode ser aplicado aos critérios Análise de Valor-limite e Teste Funcional Sistemático, apoiados pela J-FuT.

6.1. O programa *Identifier*

O programa *Identifier* deve determinar se um identificador é ou não válido na linguagem *Silly Pascal*, e sua especificação é descrita a seguir:

"Um identificador válido deve começar com uma letra e conter apenas letras ou dígitos. Além disso, deve ter no mínimo 1 caractere e no máximo 6 caracteres de comprimento".

A implementação em Java utiliza uma classe denominada `Identifier`, que possui um método estático responsável pela verificação do identificador (`verify`). Esse método recebe como parâmetro uma *string* e retorna verdadeiro caso seu valor represente um identificador válido, ou falso caso contrário. A assinatura do método é mostrada abaixo:

```
public static boolean verify(String)
```

A análise da especificação do programa permite extrair seis classes de equivalência para o método `verify`. Essas classes de equivalência são agrupadas em três condições de entrada, conforme apresentado na Tabela 3. A partir das classes de equivalência e assinatura da operação `verify`, é possível especificar e implementar os casos de teste para avaliação do programa. Para isso foi elaborado no JUnit um conjunto de quatro casos de teste, que exercitam os seis requisitos apresentados na Tabela 3: *T={(a1; válido), (2B3; inválido), (Z-12; inválido), (A1b2C3d; inválido)}*. É importante notar que esse conjunto obedece às regras do Particionamento de Equivalência, que obriga que seja criado um caso de teste para cada classe de equivalência inválida e um número mínimo de casos de teste para testar as classes válidas.

Tabela 3. Condições de entrada e classes de equivalência para o método `verify`

	Classes de equivalência	
Condição de entrada	**válidas**	**inválidas**
Tam. do identificador (t)	$1 \leq t \leq 6$	$t > 6$
Primeiro caractere é uma letra	Sim	Não
Contém somente caracteres válidos	Sim	Não

6.2. Utilização da J-FuT

As três condições de entrada que foram derivadas a partir da especificação do programa `verify` são convertidas em três aspectos de teste. Cada aspecto de teste, por sua vez, contém métodos que avaliam as diversas classes de equivalência (predicados). O código da Figura 6 implementa o aspecto `PrimeiroCaractere`, que testa a classe de equivalência "Primeiro caractere é uma letra". Como é possível observar, o conjunto de junção `operationCall` foi concretizado para interceptar chamadas ao método `verify` e as duas classes de equivalência da condição de entrada estão refletidas nos métodos de teste `valid_primeiro_letra` e `invalid_primeiro_nao_letra`.

```
1  public aspect PrimeiroCaractere extends AbstractInstrumentation {
2
3    private String valor;
4
5    public void setUp(java.lang.String valor) {
6      this.valor = valor;
7    }
8
9    public pointcut operationCall() :
10       call(public static boolean Identifier.verify(java.lang.String));
11
12   // o identificador é capturado no valor
13   // verifica se o primeiro caractere do identificador é uma letra
14   public boolean valid_primeiro_letra() {
15     return Character.isLetter(valor.charAt(0));
16   }
17
18   // o identificador é capturado no valor
19   // verifica se o primeiro caractere do identificador não é uma letra
20   public boolean invalid_primeiro_nao_letra() {
21     return !valid_primeiro_letra();
22   }
23 }
```

Figura 6. Aspecto para teste da classe de equivalência "Primeiro caractere é uma letra"

Nota-se também a implementação do método `setUp`, que captura o parâmetro do método em teste e o armazena em um atributo homônimo do aspecto (`valor`), tornado-o disponível aos métodos de teste. O `setUp` é executado automaticamente pela ferramenta J-FuT sempre que o aspecto intercepta chamadas ao método `verify`, e antes dos métodos de teste serem executados. A J-FuT cria automaticamente a estrutura dos aspectos de teste, concretizando o conjunto de junção `operationCall`, criando o método `setUp` e os esqueletos dos métodos de teste. Cabe ao testador portanto codificar os métodos de teste, implementando a lógica dos predicados que avaliam os requisitos.

Após a implementação dos aspectos de teste, os casos de teste podem ser executados diretamente na J-FuT. Durante a execução dos casos de teste os aspectos implementados pelo testador interceptam chamadas ao método `verify` e executam os métodos de teste. Os resultados desses métodos são tratados pelo aspecto `TraceManager` e armazenados em um arquivo de registro. Os dados desse arquivo são utilizados na análise de cobertura. O diagrama da Figura 7 mostra a seqüencia de funcionamento do aspecto `PrimeiroCaractere`.

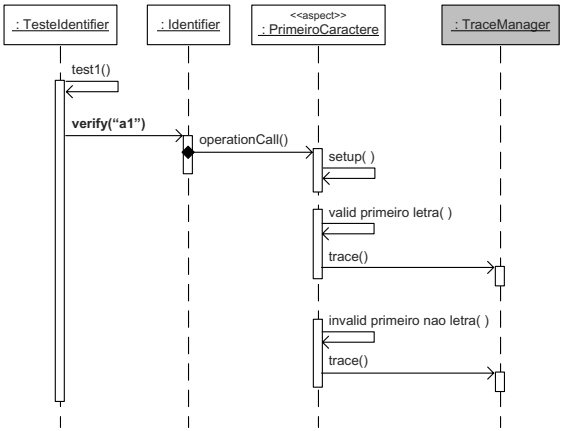

Figura 7. Execução dos métodos de teste

6.3. Cobertura funcional

A análise dos dados registrados durante a execução do programa em teste permite extrair informações a respeito da cobertura dos requisitos de teste. O cálculo de cobertura é feito a partir da interpretação do arquivo de registro, que é carregado em uma árvore DOM (*Document Object Model*) na memória e serve como fonte de dados para os algoritmos de cobertura. Durante o processo de análise dos dados, os resultados da execução dos requisitos de teste são analisados, sendo possível avaliar a cobertura alcançada de acordo com o critério de teste escolhido e o total de resultados do tipo verdadeiro (*true*).

O critério de teste escolhido define os valores que serão utilizados para efetuar a análise de cada requisito de teste, por meio dos valores dos atributos `maxRequirementPerTest` e `minTestsToCover`. Assim, um determinado requisito de teste é exercitado se possui o número mínimo de casos de teste que o executam com retorno verdadeiro (*true*). Um caso de teste, por sua vez, possui um número máximo de requisitos de teste que pode exercitar. Utilizando esse conceito, a J-FuT permite a quantificação da atividade de teste segundo dois tipos de cobertura, descritos na Tabela 4.

Tabela 4. Tipos de cobertura disponíveis na J-FuT

Tipo de cobertura	Descrição
Condição de entrada (aspecto)	Uma condição de entrada de um método é coberta quando todos os seus requisitos de teste (classes de equivalência, pré e pós-condições e valores-limite) são exercitados. É recomendável que cada aspecto de teste represente uma condição de entrada
Método	Um método é coberto quando todas as suas condições de entrada são cobertas

A cobertura de um aspecto (condição de entrada) fornece uma medida para avaliar o quanto uma determinada restrição referente àquela operação foi satisfeita. A cobertura de um método (ou operação), por outro lado, mede a quantidade de restrições daquele método que foi cumprida. Na Tabela 5 é apresentado um exemplo de cobertura para um método fictício

de uma classe qualquer. Para o teste desse método foram implementados três aspectos de teste, cada um com uma quantidade distinta de predicados (requisitos de teste). O cálculo de cobertura para os aspectos de teste é feito de forma direta, dividindo-se o número de requisitos cobertos pelo total de requisitos do aspecto. Para o aspecto A_1, por exemplo, tem-se a cobertura de 60%. A cobertura do método, por sua vez, é obtida a partir da divisão do somatório dos requisitos de cada aspecto pelo somatório dos requisitos cobertos em cada aspecto.

Tabela 5. Exemplo de análise de cobertura

Aspecto	Requisitos	Requisitos cobertos	Cobertura aspecto	Cobertura método
A_1	5	3	60%	63%
A_2	4	2	50%	
A_3	10	7	70%	

Ao contrário dos critérios estruturais, nos quais é relativamente simples avaliar se um requisito de teste foi coberto, nos critérios da técnica funcional essa análise apresenta algumas peculiaridades. Um fato que merece destaque é o de que um caso de teste pode exercitar ao mesmo tempo classes de equivalência válidas e inválidas, o que requer uma avaliação criteriosa. Para lidar com essa característica, a J-FuT implementa algumas estratégias para análise dos dados coletados. As quatro estratégias definidas na ferramenta atendem a diferentes contextos e são descritas sucintamente na Tabela 6. Maiores detalhes podem ser obtidos em outro trabalho [25].

Tabela 6. Estratégias de teste oferecidas pela J-FuT

Nome	Descrição
Invalidar casos de teste	Estabelece que casos de teste que excedem o máximo de requisitos de um tipo que podem exercitar (atributo `maxRequirementPerTest`) são invalidados e desconsiderados na análise de cobertura. Deve ser adotada para avaliar o conjunto de casos de teste disponível de forma mais conservadora, independente da ordem de execução dos casos de teste ou aspectos de teste
Primeiros válidos	São considerados os requisitos de teste exercitados por um caso de teste até que este atinja o máximo de requisitos que pode exercitar (atributo `maxRequirementPerTest`), quando é então invalidado. O uso dessa estratégia é recomendado quando deseja-se preservar a ordem em que os casos de teste e aspectos de teste são executados
Otimizar casos de teste	Rearruma os dados contidos no registro de teste visando a aumentar a sua cobertura. A otimização leva em consideração o fato de que uma reorganização dos requisitos exercitados pelos diversos casos de teste, preservando alguns exercícios e descartando outros, pode levar a uma nova (maior) cobertura. Nessa reorganização a prioridade é dada aos casos de teste, ou seja, tenta-se fazer com que cada caso de teste exercite o máximo de requisitos possível até atingir seu limite (atributo `maxRequirementPerTest`), quando é invalidado
Otimizar requisitos de teste	Assemelha-se à estratégia anterior. A implementação, no entanto, foca na otimização dos requisitos de teste e não mais dos casos de teste

É possível observar que cada estratégia analisa os dados de forma distinta e o testador deve escolher a que melhor lhe convém. Um ponto importante é que nas quatro estratégias são desconsiderados os exercícios das classes de equivalência válidas que são exercitadas

por casos de teste que também exercitam classes de equivalência inválidas. Essa regra tem o objetivo de evitar o "mascaramento" de erros no programa em teste. O exercício de uma classe de equivalência inválida por um caso de teste já aponta um erro no programa e portanto esse mesmo caso de teste não pode ser utilizado para exercitar classes de equivalência válidas.

Na Figuras 8 e 9 são exibidos, respectivamente, os relatórios de cobertura de condições de entrada e de métodos utilizando a estratégia "invalidar casos de teste". Na Figura 8 é possível observar as três condições de entrada e as seis classes de equivalência implementadas para o teste da aplicação. Como todos os requisitos (classes de equivalência) foram exercitados pelos casos de teste, as condições de entrada alcançaram cobertura de 100%. Na Figura 9 percebe-se que o método em teste (`verify`) também alcançou cobertura de 100%. Isso ocorreu porque todas as suas condições de entrada forma cobertas, como mostrado na figura anterior.

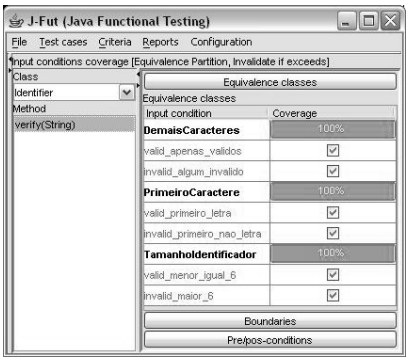

Figura 8. Cobertura de condições de entrada

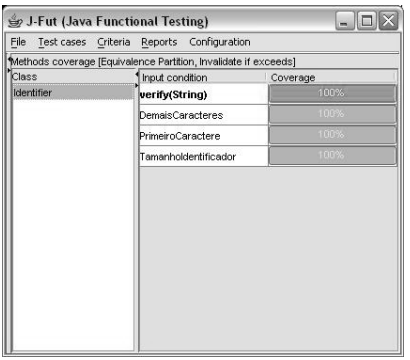

Figura 9. Cobertura de métodos

7. Considerações finais

A técnica funcional apresenta características importantes, que facilitam sua utilização. Como requer apenas a especificação do software para derivar os requisitos de teste, essa técnica pode ser aplicada indistintamente em programas procedimentais ou orientados a objetos. Além disso, os testes podem ser criados logo no início do processo de desenvolvimento e sua especificação é independente de uma implementação em particular.

Percebe-se no entanto uma carência de ferramentas que apóiem o teste funcional e utilizem os critérios dessa técnica. A ferramenta J-FuT fornece apoio ao teste funcional e procura suprir essa carência, permitindo o uso de diversos critérios e oferecendo análise de cobertura, característica geralmente não encontrada nas ferramentas atuais. Além disso, utilizando a ferramenta é possível efetuar teste funcional de unidade (métodos) ou de sistema (operações) e avaliar a adequação de um determinado conjunto de casos de teste a um critério funcional específico. Para instrumentação e análise do comportamento do programa em teste a J-FuT utiliza a POA, que oferece recursos poderosos para implementação de interesses ortogonais. O uso dessa técnica permite que o código de teste fique totalmente separado do código do programa, facilitando a sua adição e remoção.

O projeto da ferramenta baseia-se em predicados que representam requisitos de teste (classes de equivalência e valores-limite) e são utilizados internamente para o teste de sa-

tisfação. O uso de predicados facilita a construção e manutenção do teste, no entanto, a representação dos requisitos sob a forma de predicados nem sempre é uma tarefa trivial e pode requerer a implementação de diversas funções auxiliares. Assim, predicados complexos, cujos custos de elaboração sejam altos, dificultam o uso da J-FuT. A implementação dos predicados constitui a principal limitação da ferramenta, pois ainda é feita manualmente pelo testador e pode demandar algum esforço.

Como continuidade deste trabalho, pretende-se analisar exemplos complexos que exijam a manipulação de requisitos de teste não-triviais, para avaliar a flexibilidade e robustez da J-FuT. Também será estudado o auxílio que da POA em outras fases de teste, nas quais acredita-se que essa técnica possa trazer benefícios como, por exemplo, o teste de regressão. O teste de sistema também deverá ser explorado de maneira detalhada, por meio de operações complexas. Além disso, pretende-se estudar a derivação dos requisitos de teste a partir de especificações OCL (*Object Constraint Language*) e geração automática dos aspectos de teste. Outro tópico que deve ser analisado refere-se ao uso da ferramenta para o teste de programas escritos em AspectJ. O teste funcional de programas escritos nessa linguagem parece ser possível pois o *bytecode* gerado pelo AspectJ é compatível com Java. No entanto, ainda é necessário um estudo mais aprofundado para verificar se essa linguagem permite a instrumentação de suas próprias construções.

Referências

1. AUTOMATEDQA CORP. TestComplete. 1999. Disponível em `http://www.automatedqa.com/products/testcomplete/index.asp` (Acessado em 07/12/2004)
2. BARBOSA, D. L.; ANDRADE, W. L.; MACHADO, P. D. L.; FIGUEREDO, J. C. A. SPACES – Uma Ferramenta para Teste Funcional de Componentes. *Anais da XI Sessão de Ferramentas – XVIII Simpósio Brasileiro de Engenharia de Software*, p. 55–60, 2004.
3. BINDER, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. 1 ed. Massachusetts: Addison Wesley, 1191 p., 1999.
4. BRUEL, J.-M.; ARAÚJO, J.; MOREIRA, A.; ROYER, A. Using Aspects to Develop Built-In Tests for Components. In: AKKAWI, F.; BOOCH, O. A. A. G.; CLARKE, S.; GRAY, J.; HARRISON, B.; KANDÉ, M.; STEIN, D.; TARR, P.; ZAKARIA, A., eds. *Proceedings of the 4th AOSD Modeling With UML Workshop*, San Francisco – CA, 2003.
5. CHAIM, M. L. *Poke-tool: Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseados em Análise de Fluxo de Dados*. Dissertação de Mestrado, DCA/FEE/UNICAMP, Campinas – SP, 1991.
6. DEBUSMANN, M.; GEIHS, K. Efficient and Transparent Instrumentation of Application Components using an Aspect-oriented Approach. In: *14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 2003)*, Heidelberg – Germany: Springer, 2003, p. 209–220 (*Lecture Notes in Computer Science (LNCS)*, v.2867).
7. DETERS, M.; CYTRON, R. K. Introduction of Program Instrumentation using Aspects. In: *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa – FL: ACM, 2001.
8. DIOTALEVI, F. Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ. 2004. Disponível em `http://www-106.ibm.com/developerworks/library/j-ceaop/` (Acessado em 25/08/2004)
9. ELRAD, T.; AKSIT, M.; KICZALES, G.; LIEBERHERR, K.; OSSHER, H. Discussing Aspects of AOP. *Communications of the ACM*, v. 44, n. 10, p. 33–38, 2001.

10. GAMMA, E.; BECK, K. JUnit, Testing Resources for Extreme Programming. 2002. Disponível em `http://www.junit.org/` (Acessado em 16/11/2004)
11. ISBERG, W. Get Test-Inoculated! Software Development Article, 2002. Disponível em `http://www.sdmagazine.com/documents/s=7360/sdm0205b/` (Acessado em 26/10/2004)
12. KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. An Overview of AspectJ. *Lecture Notes in Computer Science*, v. 2072, p. 327–355, 2001.
13. KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-Oriented Programming. In: AKSIT, M.; MATSUOKA, S., eds. *Proceedings of the European Conference on Object-Oriented Programming*, v. 1241, Berlin, Heidelberg, and New York: Springer-Verlag, p. 220–242, 1997.
14. LINKMAN, S.; VINCENZI, A. M. R.; MALDONADO, J. C. An Evaluation of Systematic Functional Testing Using Mutation Testing. In: *Proceedings of 7th International Conference on Empirical Assessment in Software Engineering*, Staffordshire – UK, 2003, p. 1–15.
15. MALDONADO, J. C.; BARBOSA, E. F.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S.; JINO, M. Teste de Software: Teoria e prática. In: *Minicurso – XVII Simpósio Brasileiro de Engenharia de Software (SBES 2003)*, Manaus – AM, 2003.
16. MAO, X.; MAY, J. A Framework of Integration Testing Using AspectJ. *International Workshop on Aspect-Oriented Software Development (WAOSD 2004)*, 2004.
17. MONK, S.; HALL, S. Virtual Mock Objects using AspectJ with JUnit. XProgramming.com, 2002. Disponível em `http://xprogramming.com/xpmag/virtualMockObjects.htm` (Acessado em 20/11/2004)
18. MYERS, G. J. *The Art of Software Testing*. 1 ed. New York: Wiley, 177 p., 1979.
19. OSTRAND, T. J.; BALCER, M. J. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, v. 31, n. 6, p. 676–686, 1988.
20. PARASOFT CORPORATION Jtest. 1997. Disponível em `http://www.parasoft.com/jtest` (Acessado em 05/11/2004)
21. PEARSON, C. *A Framework for the Aspect-Oriented Dynamic Instrumentation of Java Programs*. Relatório Técnico, Department of Computing - Imperial College London, London – France, 2003.
22. PRESSMAN, R. S. *Engenharia de Software*. 5 ed. Rio de Janeiro: McGraw-Hill, 843 p., 2002.
23. RAJAN, H.; SULLIVAN, K. *Generalizing AOP for Aspect-Oriented Testing*. Relatório Técnico CS-2004-30, University of Virginia, Virginia – USA, 2004.
24. RATIONAL CORP. IBM Rational XDE Tester. 2002. Disponível em `http://www-136.ibm.com/developerworks/rational/products/xdetester` (Acessado em 12/12/2004)
25. ROCHA, A. D. *Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas Java*. Dissertação de Mestrado, Universidade de São Paulo – ICMC/USP, São Carlos – SP, 2005.
26. VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. JaBUTi: A Coverage Analysis Tool for Java Programs. In: *Anais da Sessão de Ferramentas do XVII Simpósio Brasileiro de Engenharia de Software*, Manaus – AM, 2003.