Geração de Metadados para o Apoio ao Teste Estrutural de Componentes

Vânia Somaio Teixeira[1,2], Marcio Eduardo Delamaro[1]
[1]Programa de Pós-graduação em Ciência da Computação (PPGCC)
Centro Universitário Eurípides de Marília (UNIVEM) – Marilia, SP
[2]Faculdade Gennari&Peartree (FGP) – Pederneiras, SP
vania@fgp.com.br, delamaro@pesquisador.cnpq.br

Abstract. *The use of components in the software development brings benefits in terms of quality and productivity. On the other hand, it has added complexity to a few activities like software testing. In this context, there are two different perspectives: one from the component developer and other from the component user. Among the main problems related to the test of component-based applications is the lack of information exchanged between these two perspectives. The developer does not know the contexts where the component will be used and the user ignores how the component was validated or what its validation requirements are. In this work we propose the use of structural coverage measures produced by the developer as an aid to the integration of the component in a user application. Such information is appended to the component code as metadata generated by a tool that automatizes this process.*

Resumo. *O uso de componentes no desenvolvimento de software traz benefícios, em termos de qualidade e produtividade, mas, por outro lado, acrescenta complexidade em algumas atividades, em particular, para a atividade de teste. Nesse contexto, podem-se destacar as perspectivas do desenvolvedor e do usuário do componente. Dentre os principais problemas relacionados ao teste de aplicações baseadas em componentes está a falta de informação. O desenvolvedor não conhece os contextos de utilização do componente e o usuário ignora os requisitos e a forma de validação do componente. Propõe-se neste trabalho a utilização de medidas de cobertura de teste estrutural, fornecidas pelo desenvolvedor, para auxiliarem na integração do componente nas aplicações do usuário. Tais informações são agregadas ao componente na forma de metadados e geradas por meio de uma ferramenta que auxilia na automatização deste processo.*

1. Introdução

O objetivo da Engenharia de Software é a construção de software com qualidade e com baixo custo. Para tanto, diversos métodos, técnicas e ferramentas têm sido propostos e utilizados buscando o aumento da produtividade no desenvolvimento de software. Uma das formas para tal foi o reaproveitamento de artefatos de software. Assim, em vez de iniciar-se sempre um projeto de software "do zero", são reutilizadas soluções e experiências adquiridas em projetos anteriores. É o que se costuma chamar de reúso de software.

Brereton e Budgen (2000) argumentam que o desenvolvimento baseado em componentes tende a trazer grandes benefícios na produção de software, favorecendo o reúso. Entretanto, eles apontam diversas questões que precisam ser tratadas, principalmente, pelos desenvolvedores. Dentre essas questões estão: 1) a descrição do componente – favorecendo sua localização, entendimento e avaliação; 2) o suporte automatizado – visando ao desenvolvimento de ferramentas para apoiar as atividades de busca, entendimento, avaliação e visualização das decisões no uso de componentes; e 3) a avaliação de componentes – fornecendo mecanismos para facilitar a sua avaliação e comparação.

Embora o reúso de software não seja uma prática recente, a adoção do paradigma de desenvolvimento baseado em objetos popularizou a adoção de novas técnicas de reúso como o desenvolvimento de software baseado em componentes, que é a reutilização de software por meio do encapsulamento de funcionalidades em unidades de software que são independentes e podem ser acopladas a um sistema por meio de uma interface conhecida.

Beydeda e Gruhn (2003) reafirmam os benefícios no uso de componentes, mas, por outro lado, admitem que existam dificuldades que precisam ser transpostas, principalmente na fase de testes. Dificuldades essas, como a falta de informação sentida tanto pelo desenvolvedor quanto pelo usuário do componente, para que a atividade de teste seja realizada de forma sistemática e com qualidade. O desenvolvedor do componente não conhece todos os contextos de utilização do componente e, portanto, o componente pode ter um comportamento em determinados ambientes conforme o esperado e em outros não. O usuário sofre com a falta de documentação adequada sobre o componente e como ele foi validado, agravada, em geral, pelo fato de o código fonte não lhe estar disponível. A troca mais efetiva de informações entre desenvolvedor e usuário do componente pode colaborar para se definir uma estratégia de teste adequada para o desenvolvimento baseado em componentes.

Neste contexto, o foco deste trabalho é estabelecer uma estratégia de teste, na qual o desenvolvedor teste o componente e forneça ao usuário as informações sobre como essa atividade foi desenvolvida, na forma de metadados. Propõe-se, para isso, a utilização da técnica de teste estrutural, e a geração de metadados que sumarizem a cobertura de requisitos estruturais alcançados pelo desenvolvedor. Essa informação deve auxiliar o usuário do componente a avaliar a adequação dos seus próprios conjuntos de testes em exercitar os requisitos estruturais do componente. Para viabilizar essa estratégia, propõe-se, também, uma ferramenta para geração (pelo desenvolvedor) e utilização (pelo usuário) desses metadados.

Na próxima seção são destacados alguns trabalhos relacionados ao teste de componentes, em particular aqueles que abordam a geração e utilização de algum tipo de informação extra que facilite o teste de aplicações baseadas em componentes. A Seção 3 descreve a estratégia de teste proposta. A Seção 4 resume as funcionalidades da ferramenta FATEsC (Ferramenta de Apoio ao Teste Estrutural de Componentes), que apóia a aplicação da estratégia. A Seção 5 discute o estudo de caso realizado e a Seção 6 contém as considerações finais deste artigo.

2. Trabalhos relacionados

Partindo-se da constatação de que a falta de informação é um entrave para a atividade de teste de componentes, encontram-se na literatura diversas abordagens que procuram solucionar ou minimizar esse problema. Dentre elas, Liu e Richardson (1998) propõem

o *retro-component*, que fornece informações por meio de um mecanismo chamado retrospector, que tem por objetivo incorporar ao componente, dados estáticos e dinâmicos sobre os testes e execuções do mesmo. O *retrospector* é uma entidade de software que tem a capacidade de comunicar-se com as ferramentas usadas pelo desenvolvedor, testador e usuário do componente, captando informações sobre os testes realizados, recomendações de testes para o usuário do componente, critérios utilizados, histórico dos testes, casos de teste, parâmetros utilizados na execução e essas informações podem estar gravadas e disponíveis para o desenvolvedor e usuário do componente.

Orso *et al.* (2000) propõem que informações adicionais (metadados), de vários tipos, sejam fornecidas pelo desenvolvedor ao usuário do componente. Três propriedades básicas dos metadados são definidas: i) o desenvolvedor do componente deve estar envolvido na produção dos metadados; ii) os metadados devem ser empacotados com o componente e; iii) o desenvolvimento e a apresentação dos metadados devem ser apoiados por ferramentas. Também propõem para cada tipo de metadado que seja criada uma identificação única (DTD – *Document Type Definition*) para facilitar sua utilização.

Bundell et al. (2000), da mesma forma, propõem que se forneçam informações adicionais sobre o componente para apoiar as atividades de análise e teste. Para isso, sugerem preparar uma especificação de teste que descreva implementações do componente, interfaces fornecidas para cada implementação e um conjunto de teste apropriado para testar as interfaces. Essa especificação de teste deve ser armazenada em arquivos XML (*Extensible Markup Language*), pois esse tipo de arquivo é facilmente processado por várias ferramentas, de modo que a especificação de teste possa ser lida, interpretada e modificada.

Edwards (2001) propõe que informações sejam empacotadas pelo desenvolvedor do componente utilizando a técnica de "*wrapper*" e fornecidas para o usuário do componente. O funcionamento da técnica *wrapper* está baseado na definição de um componente, denominado de *wrapper*, que encapsula o componente original e atua como filtro para as requisições recebidas, determinando o comportamento do componente como desejado. As informações seriam sobre a especificação do componente e as ações de garantia da qualidade executadas pelo desenvolvedor.

Nenhum desses trabalhos, porém, explicita quais informações devem ser fornecidas pelo desenvolvedor, ou como o usuário do componente pode utilizar essas informações. Dessa forma, o trabalho corrente contribui ao definir uma estratégia de utilização de informações de cobertura, resultantes do teste estrutural conduzido pelo desenvolvedor, que seriam disponibilizadas ao usuário, juntamente com o componente, na forma de metadados.

3. Estratégia de teste estrutural para aplicações baseadas em componentes

A técnica de teste estrutural, conhecida também por teste caixa-branca tem, tradicionalmente, como requisito básico a necessidade do uso do código fonte, visto que seu objetivo é exercitar as estruturas internas do programa. Os casos de teste derivados dessa técnica visam a garantir que estruturas de um programa (comandos, caminhos, variáveis, etc) sejam exercitadas de maneira a garantir alguma propriedade que indique certa qualidade do conjunto de teste utilizado. Sabe-se que nenhuma das técnicas de teste é completa, no sentido que a cada uma podem-se associar características específicas em termos de efetividade em revelar defeitos. No contexto de teste de

componente isso não muda; assim, a existência de outras estratégias de teste que utilizem, por exemplo, a técnica de teste funcional, não invalida a estratégia aqui proposta, pelo contrario, a complementa.

Tradicionalmente, para se utilizar a técnica de teste estrutural é requisito básico que o código fonte esteja disponível e isso é uma contraposição para desenvolvimento baseado em componente se olharmos pela ótica do usuário do componente, pois na maioria das vezes o código fonte não está disponível. Pela ótica do desenvolvedor do componente a atividade de teste é muito parecida com as desempenhadas em desenvolvimento tradicional [VINCENZI *et al.*, 2005].

Para que a estratégia proposta seja viável faz-se necessária a análise de cobertura estrutural mesmo sem a disponibilidade do código fonte, quando realizada pelo usuário do componente. Isso é possível em alguns cenários por meio da análise do código objeto, como faz a ferramenta JaBUTi [VINCENZI *et al.*, 2006], que foi desenvolvida para apoiar o teste estrutural em programas Java. Com o uso desta ferramenta, que efetua toda a análise estática e dinâmica do programa em teste a partir do bytecode Java, é permitido ao usuário do componente avaliar a cobertura do seu conjunto de teste sobre o código do componente, sem que o código-fonte lhe esteja disponível.

3.1. Premissa

Considere um critério de teste estrutural **C**, um componente **k** e um conjunto de teste $\mathbf{T_1 = \{t_1, t_2, ..., t_5\}}$, **C**-adequado[1] para **k**. Se uma aplicação **A** reutiliza o código do componente **k**, pode-se utilizar as medidas de cobertura do código **k** para avaliar o nível de integração obtido por um conjunto $\mathbf{T_2 = \{t'_1, t'_2\}}$ utilizado para testar a aplicação **A**. Isso implica que se $\mathbf{T_1}$ está adequado, segundo o critério utilizado, para o código do componente **k**, e $\mathbf{T_2}$ tem as mesmas medidas de cobertura obtidas por $\mathbf{T_1}$, então pode-se supor que $\mathbf{T_2}$ também é bom para exercitar a integração do componente na aplicação **A**.

Porém, é preciso lembrar que muitas vezes não é possível que o usuário do componente obtenha a mesma cobertura conseguida pelo desenvolvedor do componente utilizando um critério **C**, visto que nem sempre todas as funcionalidades do componente são utilizadas pela aplicação do usuário. Um resumo dessa idéia pode ser visto na Figura 1. No lado do desenvolvedor, um *driver* de teste é usado para cobrir um conjunto de requisitos do componente. Os requisitos pontilhados são os que o desenvolvedor conseguiu cobrir com seus casos de teste. Do lado do usuário, esses requisitos aparecem como círculos contínuos e serão utilizados para avaliar a cobertura obtida por um conjunto de teste do usuário. Certamente o usuário não cria um conjunto de teste para o componente, mas sim para sua aplicação. Porém, a cobertura obtida indiretamente por seus casos de teste sobre o código do componente dá a idéia de integração entre sua aplicação e o componente.

[1] Dado um conjunto de teste T e um critério C, caso T contenha casos de teste capazes de exercitar todos os requisitos de teste exigidos por C, diz-se que T é adequado ao critério de teste C, ou, em outras palavras, T é C-adequado.

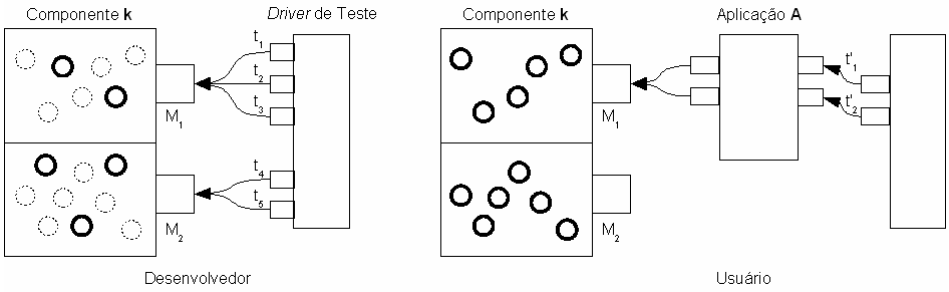

Figura 1 – Utilização de dados de teste gerados pelo desenvolvedor.

3.2. Proposta

Para a realização desta estratégia propõe-se que o desenvolvedor do componente forneça metadados empacotados com o componente ao usuário. Nesse contexto, entendem-se metadados como sendo as informações de: 1) medidas de cobertura obtidas para cada caso de teste, por método público para os critérios de teste estrutural; e 2) uma descrição das características de cada caso de teste. Para facilitar a compreensão, apresenta-se a seguir um exemplo simples considerando a classe Fat um componente, que faz o cálculo do fatorial de um número e a classe Agrupamentos uma aplicação, que faz o cálculo de combinação simples, utilizando o componente Fat. A Figura 2 mostra o código fonte da classe Fat e a Figura 3 o código fonte da classe Agrupamentos. Considera-se neste exemplo o critério Todos-Arcos, que gera 12 requisitos de teste para o método Fat, ou seja, possui 12 desvios (branchs) numerados de 0 a 11, conforme apresentado na Figura 4 e que devem ser cobertos pelos casos de teste. Pressman (2002) explica que os critérios baseado no fluxo de controle usam apenas características de controle de execução do programa, como comandos ou desvios, para determinar quais estruturas são requeridas. O critério Todos-Arcos requer que cada aresta do grafo (desvios) seja executada pelo menos uma vez.

```
public class Fat
{

    /* calcula o fatorial de um número inteiro.
       O valor passado deve ser >= 0
       e <= 20.
       O valor retornado é um long
    */

        public long fat(int x)
    {
        if ( x < 0 || x > 20 )
            return -1;
        if ( x == 0 )
            return 1;
        if ( x <= 2 )
            return x;
        long s = 1;
        while ( x > 1 )
        {
            s *= x;
            x--;
        }
     }
}
```

Figura 2. Código fonte do componente Fat

```
public class Agrupamentos {

    /* Calcula o número de combinações de k
       elementos n a n.
       Os valores passados devem ser >=1 e de
       deve ser >= n
    */

    public long combinacao(int k, int n)
    {
        Fat f = new Fat();
         if (k < 1 || n < 1 || k < n )
               throw new IllegalArgumentException("Argumento invalido");
        long s = f.fat(k);
        s /= f.fat(n);
        s /= f.fat(k-n);
        return s;
    }
}
```

Figura 3. Código fonte do aplicação Agrupamentos que utiliza o componente Fat

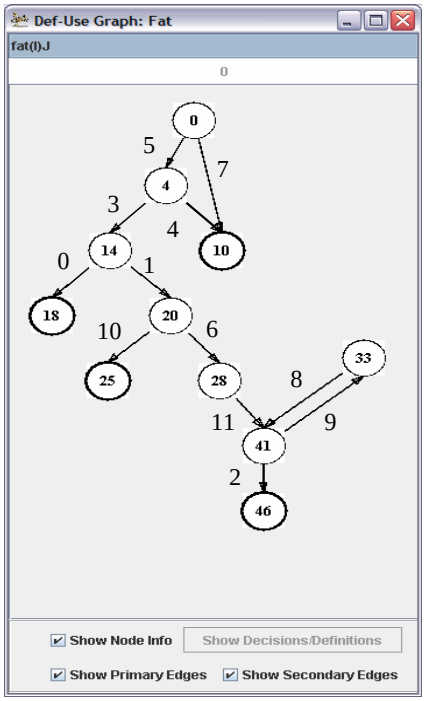

Figura 4. Grafo do método `fat` do componente gerado pela ferramenta JaBUTi

Com o apoio da ferramenta FATEsC (descrita posteriormente) o desenvolvedor associa casos de teste ao método **Fat**, conforme mostra a Tabela 1. Os casos de teste são definidos utilizando-se o formato JUnit (*framework* para desenvolvimento de teste de unidade em Java) [MASSOL e HUSTED, 2005] e os dados de cobertura são obtidos com a sua execução pela ferramenta JaBUTi.

O desenvolvedor pode também associar a cada caso de teste uma descrição textual que mostre qual o comportamento que se espera do componente, como mostra a Tabela 2. Os dados de cobertura e as descrições dos casos de teste são empacotados juntamente com o código do componente e, então, disponibilizados ao usuário.

O usuário do componente, por sua vez, define um conjunto de teste para sua aplicação e, de posse das informações descritas acima, pode avaliar sua adequação também em relação aos requisitos do componente ou, de forma mais precisa, pode

avaliar a correta integração do próprio código do componente. Suponhamos, por exemplo, que tenha definido um conjunto de quatro casos de teste conforme mostra a Tabela 3.

Tabela 1 – Exemplo de medidas de cobertura fornecidas como metadados.

Nome dos casos de teste	Requisitos cobertos para o método Fat
testFat1	7
testFat2	4 e 5
testFat3	0, 3 e 5
testFat4	1, 3, 5 e 10
testFat5	1, 2, 3, 5, 6, 8, 9 e 11

Tabela 2 – Exemplo dos descritivos dos testes fornecidos como metadados.

Conjunto de teste	Descrição informal dos testes
testFat1	Passado o valor -1 como argumento para o método Fat, pois esse trata de modo particular valores de entrada negativos.
testFat2	Passado o valor 25 como argumento para o método Fat, pois esse trata corretamente valores de entrada inteiros até 20 apenas. Para esse valor passado o resultado deveria ser -1, conforme especificado na interface do componente.
testFat3	Passado o valor 0 como argumento para o método Fat: valor com tratamento especial.
testFat4	Passado o valor 2 como argumento para o método Fat, pois esse trata de modo particular valores maiores que zero e menores ou iguais a 2.
testFat5	Passado o valor 15 como argumento para o método Fat para efetuar o cálculo do seu fatorial.

Tabela 3 – Exemplo de medidas de cobertura obtidas pelo usuário do componente.

Conjunto de teste	Dados de teste	Requisitos cobertos para o método Fat
testAgrupamentos1	k = 0, n = 1	-
testAgrupamentos2	k = 1, n = 0	-
testAgrupamentos3	k = 5, n = 7	-
testAgrupamentos4	k = 10, n = 3	1, 2, 3, 5, 6, 8, 9 e 11

Ao comparar as coberturas dos requisitos obtidas pelo desenvolvedor e usuário do componente, é constatado que em alguns casos a cobertura não foi a mesma. Por exemplo, **testFat3** cobriu 3 requisitos, dos quais os casos de teste do usuário cobriram apenas 2, conforme apresentado na Tabela 4.

Tabela 4 – Sumário dos dados de cobertura obtidos pelo usuário e desenvolvedor do componente.

Conjunto de teste	Quantidade de requisitos cobertos no componente	Quantidade de requisitos cobertos na aplicação	Porcentagem de cobertura
testFat1	1	0	0%
testFat2	2	1	50%
testFat3	3	2	66%
testFat4	4	3	75%
testFat5	8	8	100%

De posse dessas informações o usuário do componente deve analisar cada caso onde a cobertura não foi a mesma obtida pelo desenvolvedor, utilizando para apoiá-lo as descrição dos casos de testes disponibilizados como metadados. Em sua avaliação o usuário do componente pode chegar a essas conclusões:

- Os testes **testFat1** e **testFat3** não podem ser reproduzidos para sua aplicação, visto que ela não aceita valores negativos e nem menores ou igual a 1;
- Para o teste **testFat2** o usuário do componente perceberá que o componente tem o limite de cálculo do fatorial para números até 20, portanto, deveria ter tratado essa situação em sua aplicação;
- Para alcançar a mesma cobertura do teste **testFat4**, bastaria que o usuário acrescentasse um caso de teste que passasse, por exemplo, 10 e 2 como argumento, visto que o fatorial de 2 é tratado de forma diferenciada.

Essas análises mostram como a estratégia adotada por esse trabalho ajuda a revelar defeitos e como o usuário pode verificar a adequação de seus casos de teste em exercitar a integração do componente em sua aplicação e com isso melhorá-la.

4. FATEsC – Ferramenta de Apoio ao Teste Estrutural de Componentes

Apresenta-se a seguir, utilizando o mesmo exemplo acima, as funcionalidades da ferramenta FATEsC. A partir de um arquivo *jar* contendo o bytecode do componente, cuja estrutura hierárquica é apresentada na parte esquerda (1) da Figura 5, o desenvolvedor do componente pode criar casos de teste para cada interface pública do componente. Os casos de teste são exibidos na parte direita da tela (3), podendo a qualquer tempo ser selecionado. Após a seleção o código fonte do caso de teste, que está no formato JUnit, bem como sua descrição podem ser vistos e editados na parte central da tela (2).

Após a geração dos casos de teste, a FATEsC disponibiliza a funcionalidade de juntar todos os casos de testes em um arquivo único e compilá-lo. Os casos de teste devem então ser executados na Ferramenta JaBUTi. As informações de cobertura dos testes, realizados na Ferramenta JaBUTi, são capturadas pela Ferramenta FATEsC e disponibilizadas no formato XML, ou seja, para cada caso de teste utilizado para testar uma determinada interface pública do componente, obtém-se os requisitos cobertos separados por critério de teste. Um requisito coberto pode ser um comando, um desvio, uma associação definição-uso ou outro elemento estrutural, definido pelo critério de teste que está sendo usado pelo desenvolvedor.

Antes de liberar o componente, os dados de cobertura dos casos de testes e as descrições dos casos de testes são disponibilizados junto ao componente, o que significa adicionar metadados XML gerados pela FATEsC ao arquivo contendo o bytecode do componente no formato .jar, que será usado pelo usuário do componente.

O usuário do componente, por sua vez, também utiliza a FATEsC para criar os casos de testes para a sua aplicação que utiliza o componente, o processo é semelhante ao executado pelo desenvolvedor do componente, conforme Figura 6. Após a obtenção dos dados de cobertura dos seus casos de testes executados na Ferramenta JaBUTi, o usuário do componente, poderá fazer a análise da adequação de seus casos de testes, comparando-os com os dados de cobertura obtidos pelo desenvolvedor do componente.

O usuário seleciona o critério e a interface pública do componente para o qual os resultados serão comparados, conforme apresentado na Figura 7. Na primeira coluna são listados os casos de testes gerados pelo desenvolvedor do componente. Na segunda coluna, são apresentados os dados de cobertura do componente, ou seja, quantidade de

requisitos cobertos pelo caso de teste. Na terceira coluna são apresentadas as quantidades de requisitos cobertos pelos casos de teste do usuário em relação à cobertura obtida pelo desenvolvedor do componente e na quarta coluna o cálculo da porcentagem de cobertura obtida pelo usuário do componente em relação à cobertura obtida pelo desenvolvedor do componente. No exemplo, o caso de teste **testFat5** cobriu 8 requisitos e os casos de teste do usuário também cobriram os mesmos requisitos, portanto, a cobertura foi de 100%. Os requisitos cobertos, neste exemplo, referem-se a desvios do programa, visto que o critério escolhido é Todos-arcos.

Já nos outros casos de testes, a cobertura não foi a mesma, ou seja, os casos de testes do usuário do componente, não conseguiram cobrir os mesmos requisitos cobertos pelos casos de testes do desenvolvedor do componente. Nesse caso é necessário que se faça uma análise com o objetivo de verificar se os casos de teste do usuário não estão adequados, ou se os casos de teste do componente não podem ser reproduzidos no contexto da aplicação.

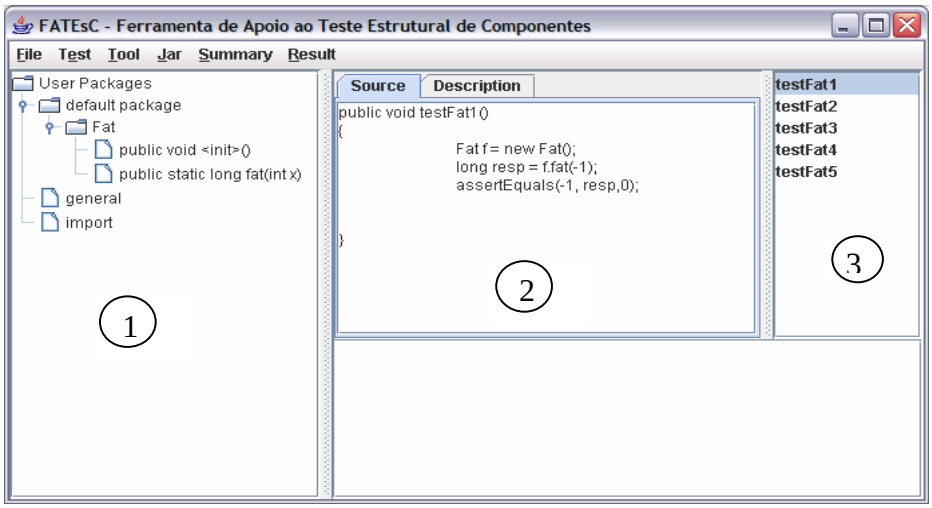

Figura 5. Tela principal da FATEsC: (1) hierarquia do componente, (3) lista de casos de teste e (2) código fonte do caso de teste testFat1.

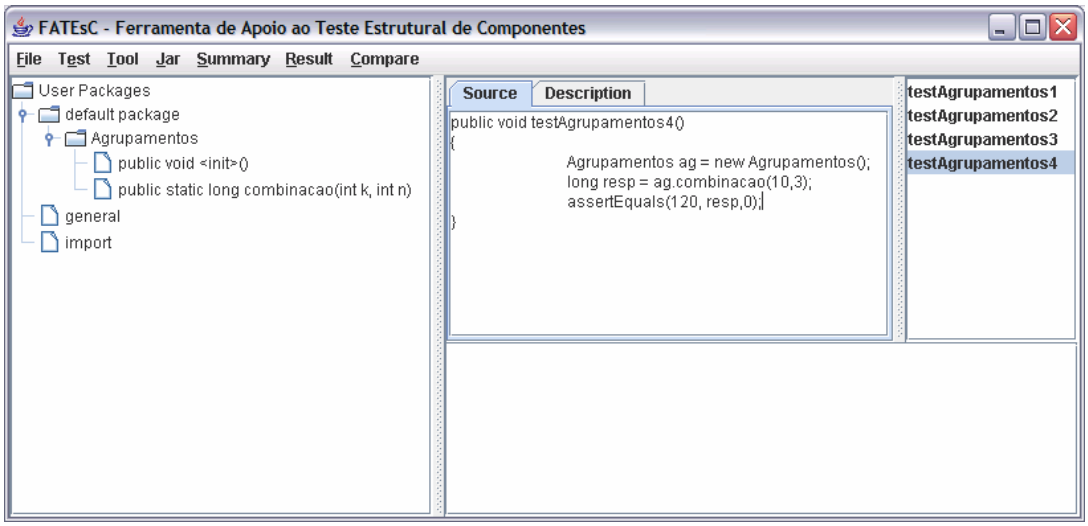

Figura 6. Tela principal da FATEsC: hierarquia da aplicação, lista de casos de teste e código fonte do caso de teste testAgrupamentos4.

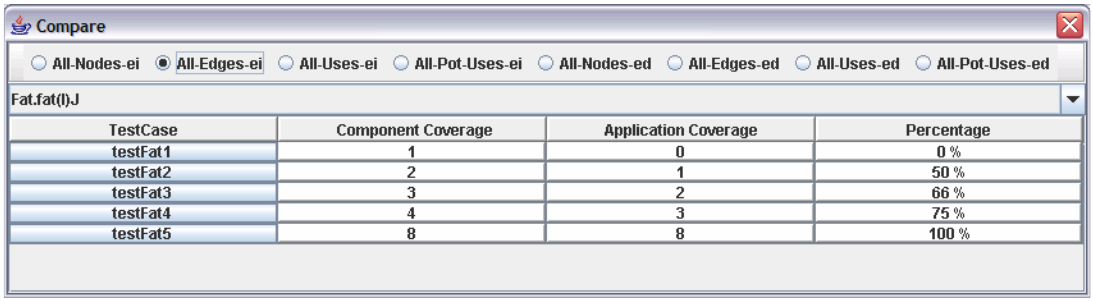

Figura 7. Apresentação dos dados de cobertura obtidos pelo usuário e pelo desenvolvedor do componente.

Para ajudar o usuário do componente em sua avaliação, é possível visualizar as descrições dos casos de testes disponibilizadas pelo desenvolvedor, bastando para isso selecionar o caso de teste desejado. Para o caso de teste **testFat1,** conforme apresentado na Figura 8, a descrição diz que foi passado um argumento negativo para o método **fat** e como a aplicação do usuário não faz acesso à interface pública do componente para parâmetros k < 1, n < 1 e k < n conclui-se que esse caso de teste não pode ser reproduzido para a aplicação. Nesse caso marca-se "infeasible" na tela de apresentação da descrição do caso de teste e o caso de teste é apresentado com uma cor diferente (cinza) representando a avaliação feita, conforme mostrado na Figura 9.

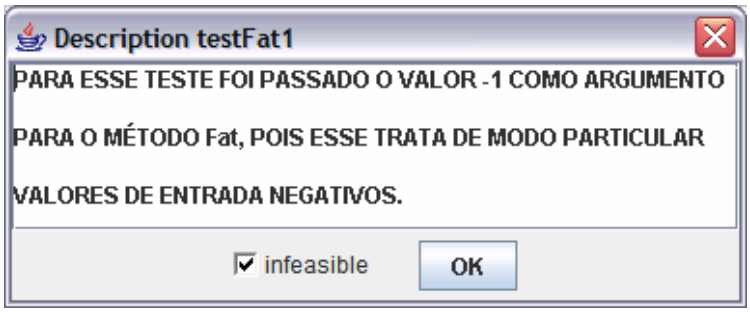

Figura 8. Apresentação da descrição do caso de teste testFat1.

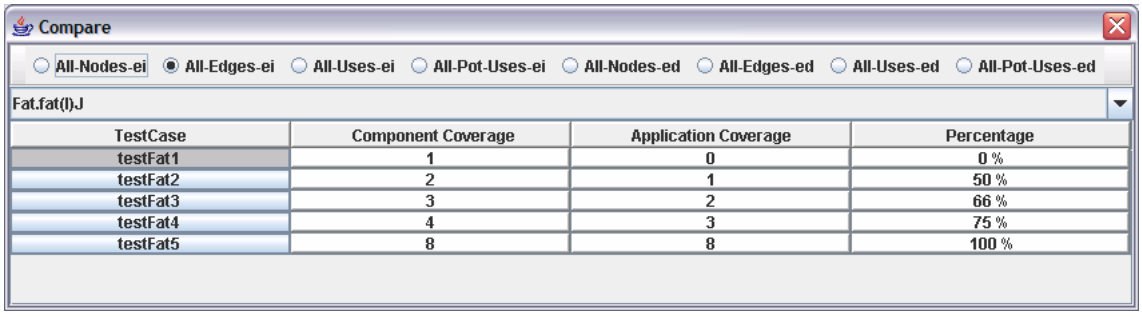

Figura 9. Aparência de um caso de teste (testFat1) que não poderá ser reproduzido na aplicação do usuário do componente.

Ao analisar o caso de teste **testFat4**, verificando a descrição do caso de teste apresentado na Figura 10**,** o usuário do componente perceberá que poderá alcançar a mesma cobertura que o caso de teste do componente, necessitando apenas criar um novo caso de teste que passe como parâmetros k = 10 e n = 2.

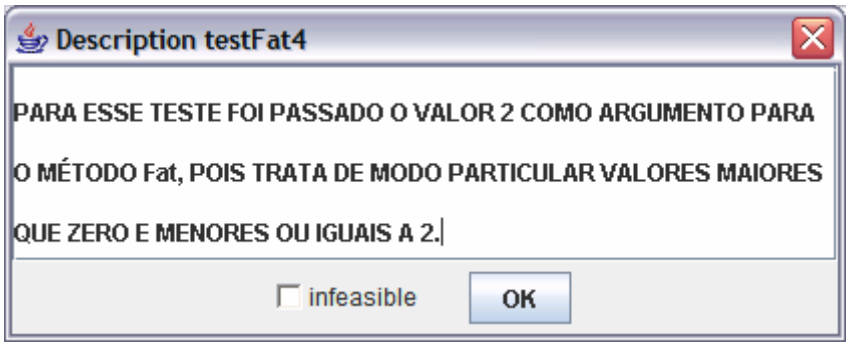

Figura 10. Apresentação da descrição do caso de teste testFat4.

Ao analisar o caso de teste **testFat2**, verificando a descrição do caso de teste apresentado na Figura 11**,** o usuário do componente perceberá que o cálculo do fatorial de 25 não é -1, então poderá se lembrar que o componente calcula o fatorial até o número 20 e que esta limitação deverá ser tratada em sua aplicação, isso mostra como a estratégia adotada por esse trabalho ajuda a revelar defeitos. Assim, o usuário do componente deveria continuar suas análises até que as coberturas se igualassem ou se concluísse que alguns casos de testes não poderiam ser reproduzidos para a aplicação.

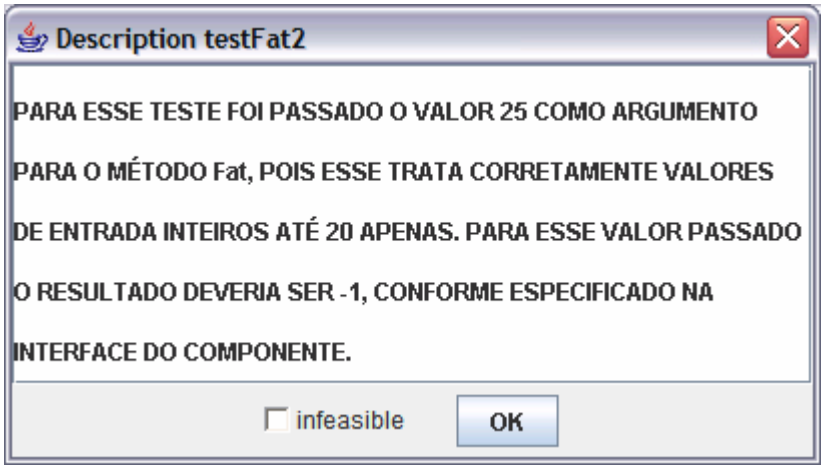

Figura 11. Apresentação da descrição do caso de teste testFat2.

5. Estudo de caso

Para o estudo de caso utilizou-se o pacote Lookup da ferramenta JaBUTi como componente e desenvolveu-se uma aplicação que utiliza algumas funcionalidades do componente. O componente Lookup é composto por quatro classes: 1) ***Program*** que é uma abstração de um programa com vários arquivos *.class*; 2) As classes de interesse dentro do programa são representadas pelo objeto ***RClassCode*;** 3) as classes periféricas aparecem como objetos ***RClass****;* 4) a classe ***ClassClosure*** é usada para explorar a estrutura do programa e contém alguns métodos para procurar as dependências de classes do programa.

O objetivo deste estudo de caso foi validar, com dados mais substanciais, as funcionalidades da ferramenta FATEsC e verificar se a estratégia de teste apresentada por esse trabalho apresenta benefícios. Neste estudo, a mesma pessoa que elaborou os casos de teste para o componente foi quem desenvolveu a aplicação e seus casos de teste. O componente Lookup tem 803 linhas de código e foram criados 82 casos de teste na ferramenta FATEsC para exercitar os requisitos de teste, segundo o critério de teste

Todos-Arcos, dos seus 47 métodos públicos. Os casos de testes foram executados na ferramenta JaBUTi e as informações de cobertura capturadas pela ferramenta FATEsC. Todos os requisitos foram cobertos, conforme apresentados na Tabela 5.

Também foram feitas as descrições de cada caso de teste com informações para apoiar as análises das coberturas e indicações de como os casos de teste foram elaborados. Essas informações de cobertura por caso de teste e as descrições foram acrescentadas ao componente e disponibilizadas para auxiliar a verificação da adequação dos casos de teste da aplicação em testar a integração do componente na aplicação.

Tabela 5 – Resumo geral por classes do componente

classes do componente	métodos públicos	quantidade de requisitos	requisitos cobertos	cobertura	casos de teste	linhas de código
Program	18	134	134	100%	32	419
RClass	13	22	22	100%	17	82
RClassCode	7	37	37	100%	9	130
ClassClosure	9	63	63	100%	24	172
Totais	47	256	256	100%	82	803

A aplicação, nomeada UseLookup, que usa o componente Lookup foi estruturada da seguinte forma: o pacote *gui* tem-se três classes ***FindCalls***, ***LoadTree*** e ***UseLookupGui***. A classe ***FindCalls*** é responsável pelas buscas dos métodos acessados a partir de um outro método, a classe ***LoadTree*** é responsável por montar a estrutura em árvore a partir de um arquivo .zip ou .jar. e a classe ***UseLookupGui*** é responsável pela interface gráfica. Ao selecionar um método da estrutura e ir ao menu ***Find Calls*** e selecionar ***Find***, são listados todos os métodos invocados pelo método escolhido. Outra forma de executar essa funcionalidade é selecionar ***Find Out*** no menu ***Find Calls*** e escolher um novo arquivo .jar. Após a escolha do arquivo, é possível digitar o nome do método, na caixa de *input*, do qual deseja-se saber os métodos invocados, conforme Figura 12.

Foram elaborados três casos de teste para os métodos ***find***, ***loadTreeJar*** e ***loadTreeZip*** da aplicação UseLookup e são esses métodos que invocam 7 métodos do componente. Nenhum caso de teste foi elaborado para os métodos da classe ***UseLookupGui***, pois essa classe é responsável apenas pela interface gráfica da aplicação e não tem nenhuma ligação com o componente Lookup. Os casos de teste foram executados na Ferramenta JaBUTi e capturadas as informações de cobertura obtidas para os métodos do componente utilizados pela aplicação.

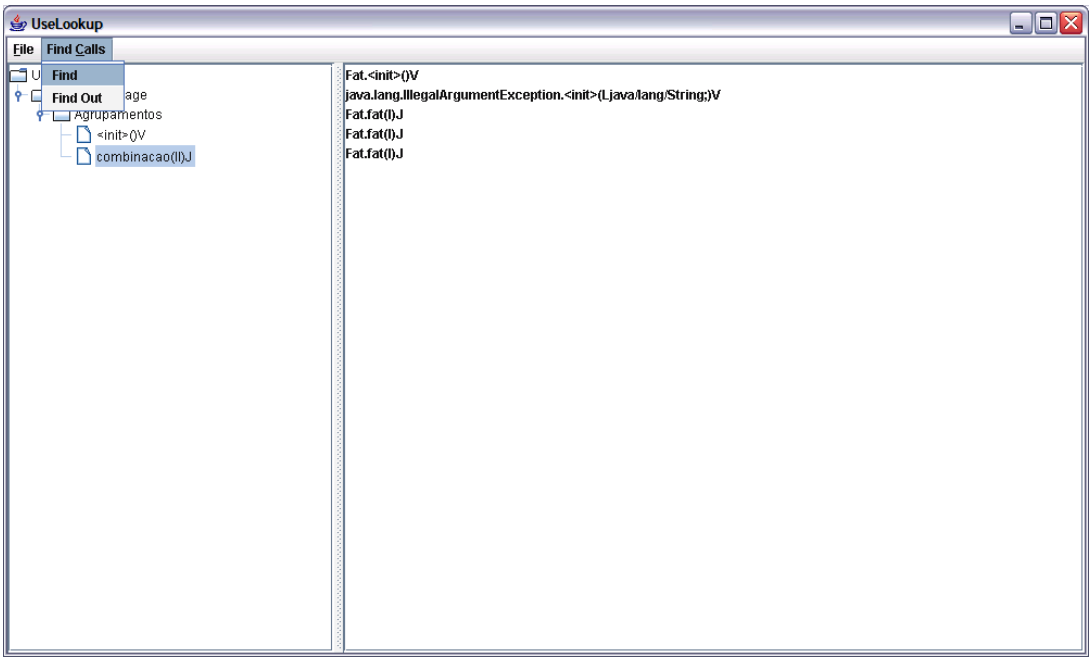

Figura 12. Menu Find Calls da aplicação UseLookup

A Tabela 6, que foi elaborada a partir das informações oferecidas pela FATEsC, apresentam-se os resultados da comparação entre a cobertura obtida pelos caso de teste do componente e a cobertura obtida pelos casos de teste da aplicação. Pode-se notar que na grande maioria as coberturas foram iguais, por exemplo, o caso de teste testProgram2 cobriu 13 requisitos e que os casos de teste da aplicação cobriram os mesmos requisitos, porém em 3 casos o mesmo não aconteceu.

Utilizando as descrições dos casos de teste fornecidas para apoiar as análises chegou-se a:

- no caso de teste testProgram27 foi utilizado um pacote que não fazia parte da estrutura principal de um programa. Como para aplicação UseLookup só são buscadas as classes para os pacotes que fazem parte da estrutura principal das classes de um determinado arquivo selecionado, concluiu-se que esse caso de teste não pode ser reproduzido para a aplicação e, portanto, pode-se marcá-lo como "*infeasible*";
- no caso de teste testRClass14 foi utilizado como argumento para o teste, uma classe que não estava em um pacote. Essa situação não foi contemplada pelos casos de teste da aplicação, portanto foi criado um novo caso de teste. Após sua execução obteve-se a cobertura esperada;
- no caso de teste testRClassCode7 há a indicação de que argumento utilizado na chamada do método do componente não está nas classes do arquivo selecionado, portanto o retorno é null. Isso fez com que se percebesse uma falha na implementação da aplicação UseLookup, pois essa situação não estava sendo tratada. Após corrigir a aplicação e gerar um caso de teste para testar essa situação obteve-se a cobertura esperada.

Tabela 6 – Resumo das quantidades dos requisitos cobertos pelos casos de teste da aplicação em relação a cada caso de teste do componente

Casos de teste elaborados para o componente	Quantidade de requisitos cobertos no componente	Quantidade de requisitos cobertos na aplicação	Porcentagem de cobertura
testProgram2	13	13	100%
testProgram32	13	13	100%
testProgram1	13	13	100%
testProgram31	13	13	100%
testProgram5	0	0	-
testProgram10	7	7	100%
testProgram27	6	5	83%
testProgram11	4	4	100%
testRClass14	1	0	0%
testRClass15	1	1	100%
testRClassCode5	13	13	100%
testRClassCode7	6	4	66%

O estudo de caso serviu para validar a proposta deste trabalho e mostrar que as informações fornecidas nos metadados ajudam na validação dos casos de teste utilizados para testar a aplicação que utiliza o componente e que também apóiam o usuário do componente a encontrar defeitos na aplicação. Quando os casos de teste elaborados pela aplicação alcançam a mesma cobertura obtida pelos casos de teste do componente, há uma forte indicação de que eles estão adequados para testar a integração da aplicação ao componente. Também foi possível visualizar e implementar melhorias na FATEsC, mudando a forma da apresentação das informações e até mesmo perceber algumas mudanças na JaBUTi que facilitariam o processo.

6. Conclusão e trabalhos futuros

Diversos trabalhos têm reconhecido as dificuldades relacionadas ao teste de software baseado em componentes. Entre os problemas levantados está a necessidade de se criarem mecanismos que permitam, principalmente do ponto de vista do usuário do componente, conhecer detalhes sobre o desenvolvimento e validação do componente.

Alguns autores propuseram o uso de metadados encapsulados junto com o componente para esse fim. Poucos, porém, mostraram explicitamente quais informações devem ser agregadas aos componentes e como estas devem ser utilizadas. Nesse sentido, este trabalho apresentou uma solução objetiva para o problema, definindo uma estratégia que permita a utilização de dados de cobertura estrutural produzidos pelo desenvolvedor do componente como fonte de auxílio ao teste das aplicações que utilizam esse componente.

O trabalho apresenta também, uma ferramenta que permite a automatização deste processo, ou seja, apóia o desenvolvedor na geração dos metadados de teste e o

usuário do componente na sua utilização, conforme requisito explicitado por Orso *et al.* (2000).

Como forma de evolução deste trabalho, é necessária a condução de estudos de casos em ambientes de desenvolvimento de software baseado em componente, para verificar a eficácia da proposta apresentada, bem como elaborar estratégias para avaliação da qualidade da ferramenta desenvolvida e promover adequações necessárias.

Referências

Beydeda, S. e Gruhn, V. (2003). State of the art in testing components. In *Third International Conference on Quality Software – QSIC'03*, pág. 146–153, Washington, DC, USA. IEEE Computer Society.

Bundell, G. A., Lee, G., Morris, J., Parker, K., e Lam, P. (2000). A software component verification tool. In *1st International Conference on Software Methods and Tools (SMT'2000)*, pág. 137–147, Wollongong, Australia. IEEE Computer Society Press.

Edwards, S. H. (2001). Toward reflective metadata wrappers for formally specified software components. In *1st Workshop on Specification and Verification of Component-Based Systems – affiliated with OOPSLA'2001*, pág. 14–21, Tampa, Florida. ACM Press.

Liu, C. e Richardson, D. (1998). Software components with retrospectors. In *International Workshop on the Role of Software Architecture in Testing and Analysis*, Marsala, Sicily, Italy.

Massol, V. e Husted, T. (2005). *JUnit em Ação*. Rio de Janeiro, RJ. Ciência Moderna.

Orso, A., Harrold, M. J., e Rosenblum, D. S. (2001). Component metadata for software engineering tasks. In *Second International Workshop on Engineering Distributed Objects – EDO'00*, pág. 129–144, London, UK. Springer-Verlag.

Pressman, R.: *Engenharia de Software*. Editora McGraw-Hill, 2002.

Vincenzi, A. M. R., Delamaro, M. E., Wong, W. E., e Maldonado, J. C. (2006). Establishing structural testing criteria for Java bytecode. *Software Practice and Experience*, 36(14):1513–1541.

Vincenzi, A. M. R., Maldonado, J. C., Delamaro, M. E., Spoto, E. S., e Wong, W. E. (2005). *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, capítulo Software Baseado em Componentes: Uma Revisão sobre Teste, pág. 233–280. Ciência Moderna, Rio de Janeiro, RJ.