FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
PROGRAMA DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

VÂNIA SOMAIO TEIXEIRA

GERAÇÃO DE METADADOS PARA APOIO AO TESTE ESTRUTURAL DE COMPONENTES

MARÍLIA
2007

VÂNIA SOMAIO TEIXEIRA

GERAÇÃO DE METADADOS PARA APOIO AO TESTE ESTRUTURAL DE COMPONENTES

Dissertação apresentada ao Programa de Mestrado do Centro Universitário Eurípides de Marília, mantido pela Fundação de Ensino Eurípides Soares Rocha, para obtenção do Título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Márcio Eduardo Delamaro

MARÍLIA

2007

Aos meus pais meus maiores incentivadores
e referência na minha vida.

AGRADECIMENTOS

Agradeço a Deus e aos meus pais pela vida.

Ao meu orientador Dr. Marcio Eduardo Delamaro pela ajuda, paciência e incentivo.

Aos meus companheiros de mestrado, pela amizade, apoio e bons momentos.

A todos que contribuíram para que eu pudesse realizar esse trabalho.

TEIXEIRA, Vânia Somaio e, **Geração de metadados para apoio ao teste estrutural de componentes**. 2007. 125 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

RESUMO

O reúso de software e, em particular, o desenvolvimento baseado em componentes têm como objetivo aumentar a qualidade e a produtividade no desenvolvimento de software por meio da reutilização de artefatos. Apesar dos benefícios, alguns aspectos do desenvolvimento baseado em componentes ainda devem ser alvo de pesquisa para que melhores resultados possam ser alcançados. Entre eles, a atividade de teste tem sido foco de pesquisa e recebido a atenção de muitos grupos de pesquisa que procuram avançar o estado da arte e solucionar os problemas a ela relacionados. Um dos pontos abordados pelos pesquisadores é a falta de troca de informação entre desenvolvedor e usuário de um componente, o que pode dificultar a atividade de teste em aplicações baseadas em componentes. Por um lado o desenvolvedor não conhece todos os possíveis ambientes em que seu componente será utilizado o que significa que o software testado em determinadas condições pode não se comportar corretamente. Por outro lado, o usuário não tem informação de como o componente foi desenvolvido ou validado pelo desenvolvedor, o que pode dificultar a validação de sua própria aplicação. Este trabalho contribui com uma possível solução deste problema propondo uma estratégia de teste que prevê a geração, por parte do desenvolvedor, de informações sobre o teste por ele realizado. Tais informações, disponibilizadas na forma de metadados como parte do componente, permitem ao usuário avaliar a integração entre sua aplicação e o código do componente. Para que isso seja possível, propõe-se a utilização de critérios de teste estrutural como metadados de teste. Ainda, para tornar a abordagem prática, é essencial que essa estratégia seja apoiada por uma ferramenta que permita ao desenvolvedor gerar e ao usuário utilizar os metadados. Assim, desenvolveu-se como parte deste trabalho uma ferramenta de teste chamada FATEsC (Ferramenta de Apoio ao Teste Estrutural de Componentes) que serve tanto ao desenvolvedor quanto ao usuário na tarefa de teste, de acordo com a estratégia proposta. Para avaliar a estratégia proposta e a ferramenta desenvolvida, foi realizado um estudo de caso utilizando como componente um pacote da ferramenta JaBUTi. Desenvolveu-se, também, um aplicativo que utiliza algumas funcionalidades desse componente proposto.

Palavras-chave: Teste de componentes, Metadados de teste, Ferramenta de teste, FATEsC.

TEIXEIRA, Vânia Somaio e, **Geração de metadados para apoio ao teste estrutural de componentes**. 2007. 125 f. Dissertação (Mestrado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2007.

ABSTRACT

Software reuse and, particularly, the component based development aims at increasing the quality and productivity on the software development through artifact reuse. Despite the undeniable contribution in this sense, some aspects of the component based development still need to be researched to achieve better results. Among these aspects, the test activity has been the center of the research and received the attention of many research teams that seek for an improvement of the state of art and to solve the problems related to it. One of the topics approached by the researchers is the lack of information exchange between the user and the developer of the component that may difficult the test activity in a component based application. On one hand the developer doesn't know all the possible environments in which his/her component will be used. This means that the software tested in a specific condition may not work on the proper way. On the other hand, the user doesn't have the information about how the component was developed or validated by the developer, and this can make the validation of his/her own application more difficult. This paper contributes to the solution for this problem, proposing a test strategy that precludes the generation of information about the test executed by the developer. Such information, available as component metadata, allows the user to evaluate the integration between his/her application and the component code. To make this possible, it is proposed the use of structural test criteria as test metadata. Also to make the approach practical, it is essential that this strategy can be supported by a tool which allows the developer to generate and the user to use the metadata. Therefore, as part of this work it is developed a tool named FATEsC (Ferramenta de Apoio ao Teste Estrutural de Componentes) which can serve both the user and the developer, on the task of testing, according with the proposed strategy. To evaluate the strategy proposed and the tools developed, it was done a case study using as the component a package of tools called JaBUTi. It was developed, also, an application that uses some functionalities of the proposed component.

Keywords: component test, test metadata, test tools, FATEsC

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

O objetivo primordial 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 vislumbradas 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.

Embora o reúso de software não seja uma prática recente, formas mais elaboradas e eficientes de reutilização têm sido propostas nos últimos anos. Em particular, a popularização e adoção do paradigma de desenvolvimento baseado em objetos representaram um significativo impulso para a adoção de uma forma de desenvolvimento chamado “Desenvolvimento Baseado em Componentes” (DBC), que permite a reutilização. (BARROCA; GIMENES; HUZITA, 2005).

O pressuposto básico do desenvolvimento baseado em componentes é a reutilização de software através do encapsulamento de funcionalidades em unidades de software que são independentes e podem ser acopladas a um sistema através de uma interface conhecida (PRESSMAN, 2002; SZYPERSKI, 1997; ORSO, HARROLD e ROSENBLUM, 2000).

Com o desenvolvimento de software baseado em componentes, veio a necessidade de metodologias que tratassem as características desse tipo de desenvolvimento, como a proposta por Cheesman e Daniels (2000) que tem duas etapas de modelagem: modelagem do domínio e modelagem da especificação. Também pode-se citar a metodologia proposta por Pressman (2002), que consiste de duas etapas executadas em paralelo, uma que é a engenharia do domínio e a outra o desenvolvimento baseado em componente.

Novas tecnologias como a orientação a objetos e o desenvolvimento baseado em componentes tendem a trazer grandes benefícios na produção de software, mas, por outro lado, adicionam dificuldades que precisam ser transpostas. No caso do desenvolvimento baseado em componentes, verificou-se que os benefícios ocorreram, mas também alguns problemas surgiram. Dentre eles, estão as dificuldades relacionadas à aplicação de técnicas de teste no escopo de DBC. Em particular, nota-se que a falta de informação, sentida tanto pelo

desenvolvedor do componente como pelo usuário do componente, para que sejam realizados testes de forma sistemática e com qualidade é um importante aspecto a ser estudado.

O desenvolvedor não conhece todos os contextos de utilização do componente e o usuário (aquele desenvolvedor que utiliza o componente para desenvolver o seu sistema) sofre com a falta de uma documentação adequada sobre o componente agravada, muitas vezes, pelo fato do código fonte não estar disponível. Várias propostas surgiram a fim de solucionar esses problemas, podendo ser classificadas em duas categorias: as que procuram lidar com a falta de informação e as que propõem facilitar a troca de informação entre o desenvolvedor e usuário do componente (BEYDEDA; GRUHN, 2003). Algumas dessas propostas são discutidas adiante, neste trabalho.

A atividade de teste é importante em todo processo de desenvolvimento de software, pois, tem como objetivo principal revelar defeitos em produtos que estão sendo testados. Apesar dos vários aspectos do desenvolvimento baseado em componente, as técnicas tradicionais podem ser utilizadas nos testes de componentes (VINCENZI et al, 2005). Nesse contexto, este trabalho tem como proposta contribuir com a proposição de uma estratégia para o teste de componentes que utiliza a técnica de teste estrutural.

1.1 Objetivo

Como citado anteriormente, um dos grandes problemas na atividade de teste para software baseado em componentes é a falta de informação que existe, em particular, quando o usuário do componente integra e testa o componente no seu produto. Diversos fatores contribuem para dificultar o teste neste contexto, principalmente o fato que o usuário não tem informações de como se realizou o desenvolvimento e o teste do componente que se propõe a reutilizar. Para suprir essa carência de informações, algumas propostas têm sido apresentadas, como Component meta-data (ORSO; HARROLD; ROSENABLUM, 2000), Retro-Components (LIU; RICHARDSON, 1998), Reflective wrapper (EDWARDS, 2001) e outras. Um problema com essas propostas é que elas estabelecem como podem ser trocadas informações entre desenvolvedor e usuário, mas não determinam exatamente o tipo de informação a ser fornecida ou como ela deve ser utilizada.

Este trabalho tem como objetivo propor uma estratégia de teste e automatizar essa estratégia implementando uma ferramenta para gerar e disponibilizar metadados encapsulado com o código do componente. Os metadados propostos são informações de cobertura dos requisitos de teste para critérios da técnica estrutural, realizados pelo desenvolvedor do componente e disponibilizados ao usuário do componente.

Para a execução dos casos de teste e geração das informações de cobertura dos requisitos utiliza-se a ferramenta JaBUTi que permite a automatização de determinados processos da fase de teste estrutural, para programas escritos na linguagem de programação Java. Para a realização das atividades dessa ferramenta não é necessário que o código fonte esteja disponível, visto que a ferramenta utiliza o bytecode. Essa característica permite que os dados de cobertura de teste do componente, alcançada durante os testes efetuados pelo desenvolvedor, possam ser reutilizados pelo usuário. Por outro lado, a abordagem aqui proposta aplica-se apenas a componentes Java, ou mais precisamente componentes cujo código objeto seja bytecode Java.

1.2 Justificativa

Observa-se na literatura que um certo investimento tem sido feito na área de teste de componentes. Alguns trabalhos apontam a necessidade de melhores canais de troca de informação entre o desenvolvedor e o usuário do componente como forma de aprimorar a atividade de teste no contexto de DBC.

Apesar de apontarem formas de promover tal comunicação como Component meta-data (ORSO; HARROLD; ROSENABLUM, 2000), Retro-Components (LIU; RICHARDSON, 1998), Reflective wrapper (EDWARDS, 2001) nota-se a carência de uma abordagem mais pragmática que mostre que tipo de informação deve ser trocada e como deve ser utilizada. Assim, este trabalho concretiza esta proposta, por meio da utilização de critérios estruturais na geração de metadados de teste para componentes Java.

1.3 Organização do trabalho

Este trabalho está organizado da seguinte maneira: nos Capítulos 2 e 3 é apresentado o embasamento teórico para o desenvolvimento desse trabalho. No Capítulo 2 são apresentados os principais aspectos sobre o desenvolvimento de software baseado em componente. Nas seções desse capítulo são apresentados: conceitos básicos sobre componentes, desenvolvimento de componentes, modelos de componentes, processos de desenvolvimento de software baseados em componentes, benefícios e problemas em desenvolvimento baseado em componentes. No Capítulo 3 são apresentados os principais aspectos de teste para desenvolvimento de software baseado em componente. Nas seções desse capítulo são apresentados: uma revisão das fases de teste e técnicas de teste, problemas com a falta de comunicação entre desenvolvedor e usuário do componente, propostas que visam a facilitar a troca de informação entre desenvolvedor e usuário do componente e propostas que lidam com os efeitos da falta de informação. No Capítulo 4 é apresentada a estratégia de teste para desenvolvimento de software baseado em componente proposta por esse trabalho. Nas seções desse capítulo são apresentados: o embasamento teórico da estratégia, a proposta do desenvolvimento da ferramenta FATEsC que apóia a estratégia, a apresentação das funcionalidades da ferramenta por meio de um exemplo simples.

No Capítulo 5 é apresentado um estudo de caso que utilizou um pacote da ferramenta JaBUTi como componente e uma aplicação que utiliza algumas funcionalidades do componente.

No Capítulo 6 são apresentadas as considerações finais sobre o trabalho, enfatizando-se as principais contribuições e apresentando-se propostas para trabalhos futuros.

Nos Apêndices A e B são mostrados resultados gerados pela ferramenta FATEsC que foram utilizados para se montar resumos, apresentados no trabalho em forma de tabelas. No Apêndice C é mostrada a descrição do componente do estudo de caso do Capítulo 5, no formato Javadoc.

2. DESENVOLVIMENTO DE SOFTWARE BASEADO EM COMPONENTES

Reutilizar soluções não é uma modalidade nova para várias áreas, como matemática e a física. Para a Engenharia de Software, a reutilização passou a ser uma realidade com a chegada da abordagem orientada a objetos, pois esta provê boas perspectivas para a construção de componentes. Muitos benefícios podem vir com o conceito de reutilização, empregado no desenvolvimento de software, como aumento da qualidade e redução do esforço de desenvolvimento (BARROCA; GIMENES; HUZITA, 2005).

Neste capítulo serão abordados alguns dos temas relacionados ao desenvolvimento de software baseado em componentes, relevantes para esse trabalho. Inicialmente será discutido o termo componente, tipos de componentes e comunicações entre componentes. Em seguida, serão discutidos o desenvolvimento e modelos de componentes, abordagens de processos, benefícios e problemas do desenvolvimento baseado em componentes.

2.1 Conceitos básicos

Para Pressman (2002, p.705), não existe uma única definição e emprego do termo componente e ele define componente como sendo “uma parte não-trivial de um sistema, praticamente independente e substituível, que preenche uma função clara no contexto de uma arquitetura bem-definida”.

Para Szyperski (1997), “componentes são unidades binárias de produção, aquisição e distribuição independente que interagem para dar forma a um sistema funcionando”. Lewis (1996), afirma que “componentes são módulos que encapsulam dados e funcionalidades e são configuráveis por parâmetros em tempo de execução”. Orso, Harrold e Rosenblum (2000) definem componentes como “um sistema ou subsistema desenvolvido por uma organização e distribuído para uma ou mais organizações, possivelmente em diferentes domínios de aplicação”. Uma definição para Barroca, Gimenes e Huzita (2005, p.4) do termo componente é: “uma unidade de software independente, que encapsula, dentro de si, seu projeto e

implementação, e oferece serviços, por meio de interfaces bem definidas, para o meio externo".

Nas definições apresentadas, os autores concordam que componente é um módulo independente e que pode ser utilizado nas soluções de desenvolvimento de software para desempenhar funções específicas.

Podem-se destacar duas categorias de componentes quanto ao seu desenvolvimento, que são os componentes desenvolvidos por terceiros, ou seja, os chamados "componentes de prateleira" (*commercial off-the-shelf,* COTS) disponíveis no mercado e pronto para o uso, e os componentes reusáveis desenvolvidos pelas empresas para seu uso próprio.

O grande atrativo para as empresas em utilizar componentes COTS, em vez de desenvolvê-los, é poder usá-los logo após a aquisição (*plug-and-play*), reduzir o custo no desenvolvimento de software e ter disponível uma solução desenvolvida por um especialista nas funcionalidades oferecidas pelo componente. Porém, existem alguns aspectos que devem ser levados em consideração e que podem se tornar um problema. Os componentes COTS podem fornecer funcionalidades que não serão utilizadas por uma aplicação, requerendo assim que se escrevam utilitários de software para restringir essas funcionalidades, a fim de que as mesmas não causem problemas no futuro. Tem-se também a dependência do fornecedor, pois o código fonte na maioria das vezes não está disponível e então, caso o fornecedor encerre suas atividades ou descontinue o produto, isso pode ser um grande problema para o cliente (VOAS, 1998).

O propósito de se usar componentes é poder reutilizá-lo em vários projetos. Quer seja componente COTS ou desenvolvido pelas empresas para uso próprio, são necessários alguns cuidados para que o reúso seja realmente um benefício. Weyuker (1998) destaca alguns aspectos que são importantes para o reúso de componentes:

- armazenar e identificar os componentes em um repositório, estando disponíveis aos vários projetos;
- ter profissionais responsáveis pela manutenção dos componentes;
- fazer uma boa gerência de configuração, controlando e divulgando as novas versões dos componentes;
- manter uma documentação atualizada da especificação do componente, especificação de testes, dados de teste, etc. e;
- executar testes a cada reutilização do componente nas novas aplicações.

Os componentes, a fim de potencializar os benefícios do reúso, devem ter algumas características importantes. Entre elas, Sommerville (2003, p.269) cita:

- refletir abstrações estáveis de domínio;
- ocultar a representação do seu estado, oferecendo apenas operações de acesso e atualização do mesmo;
- ter independência de outro componente para entrar em operação ou com dependência mínima, e;
- como cada aplicação trata as exceções a sua maneira, é importante que nos componentes as exceções façam parte de suas interfaces.

Segundo Barroca, Gimenes e Huzita (2005, p.4), interfaces são pontos de interação dos componentes que têm uma série de serviços relacionados. Portanto, os serviços relacionados a um componente são definidos pelas suas interfaces. Há dois tipos de interfaces: as interfaces fornecidas, que especificam os serviços fornecidos pelo componente e as interfaces requeridas, que especificam quais serviços devem estar disponíveis em outros componentes. Assim, a interface requerida de um componente conecta-se à interface fornecida do outro. Wu, Pan e Chen (2001) definem interface como pontos de acesso aos quais o usuário do componente pode requerer os serviços. Uma interface é composta pelo nome do serviço, a assinatura de uma função que o fornece e um conjunto de parâmetros.

Na Figura 2.1, utilizando a notação da UML, estão mostrados dois componentes e suas interfaces (requerida e fornecida). Na notação UML, as interfaces são representadas pelo relacionamento de dependência (seta pontilhada), não oferecendo representação específica para interfaces requeridas e interconexões entre componentes.

A Figura 2.2 apresenta uma outra forma de representar o componente e suas interfaces. As interfaces fornecidas definem os serviços fornecidos pelo componente e as interfaces requeridas, especificam os serviços que devem estar disponíveis a partir do sistema que está utilizando o componente.

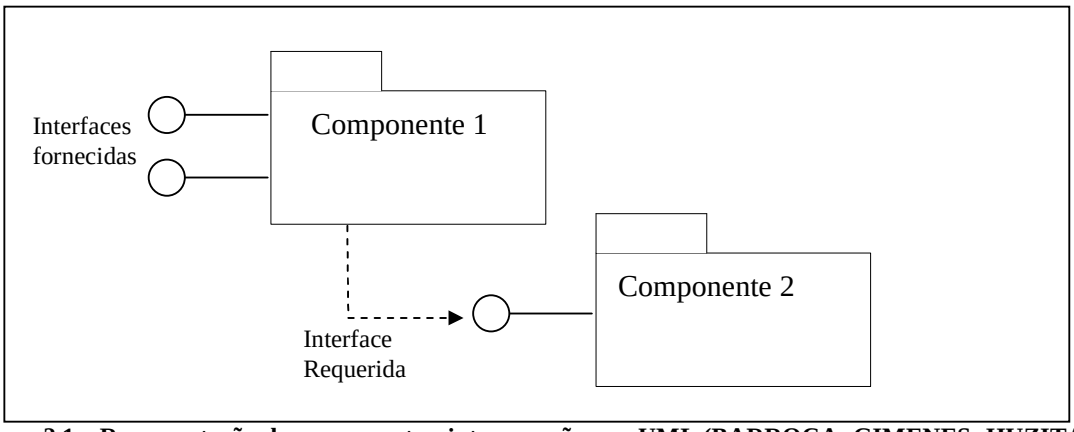


Figura 2.1 – Representação de componente e interconexões em UML (BARROCA; GIMENES; HUZITA, 2005)

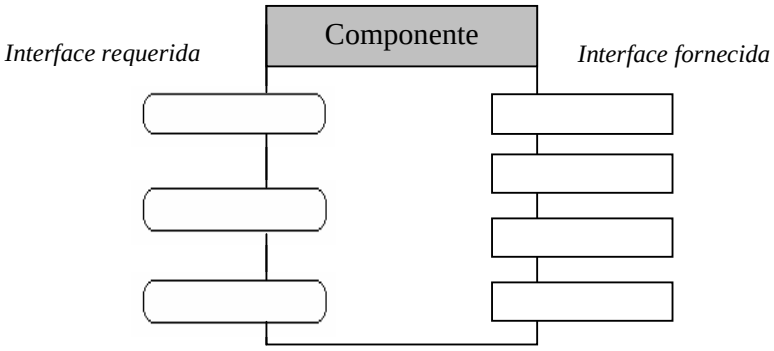


Figura 2.2 – Interfaces de componentes (SOMMERVILLE, 2003)

Um exemplo da interação entre componentes é mostrado na Figura 2.3 usando a notação UML. O componente de nome `ComponenteEndereço` processa endereços de diferentes países, com funcionalidades de criar e manter endereços postais em diferentes formatos e fornece duas interfaces: `IRUEndereço` e `IEUAEndereço`, que processam por meio de suas operações, respectivamente endereços do Reino Unido e dos Estados Unidos da América. O `ComponenteEndereço`, por sua vez, conecta-se com os componentes de nomes `ComponenteCódRU` e `ComponenteCódEUA`, responsáveis por verificarem a existência e o formato dos endereços postais e cujas interfaces fornecidas são compatíveis com a interface requerida do componente `ComponenteEndereço`.

A interação dos componentes, conforme visto anteriormente, se dá por meio de suas interfaces, portanto, Sommerville (2003, p.269) afirma que para um componente ser reutilizável é necessário que haja um equilíbrio entre fornecer uma interface genérica, com operações representando diferentes formas, pelas quais os componentes possam ser utilizados e interfaces que sejam mínimas e simples de fácil compreensão. Belloir, Bruel e Barbier (2003) citam que o componente deve ter a capacidade de ser sistematicamente e facilmente combinado com outros componentes. Essa capacidade de combinação é básica para a reutilização embora essa nem sempre seja a preocupação de quem desenvolve componentes, visto que a dependência entre componentes nem sempre é especificada.

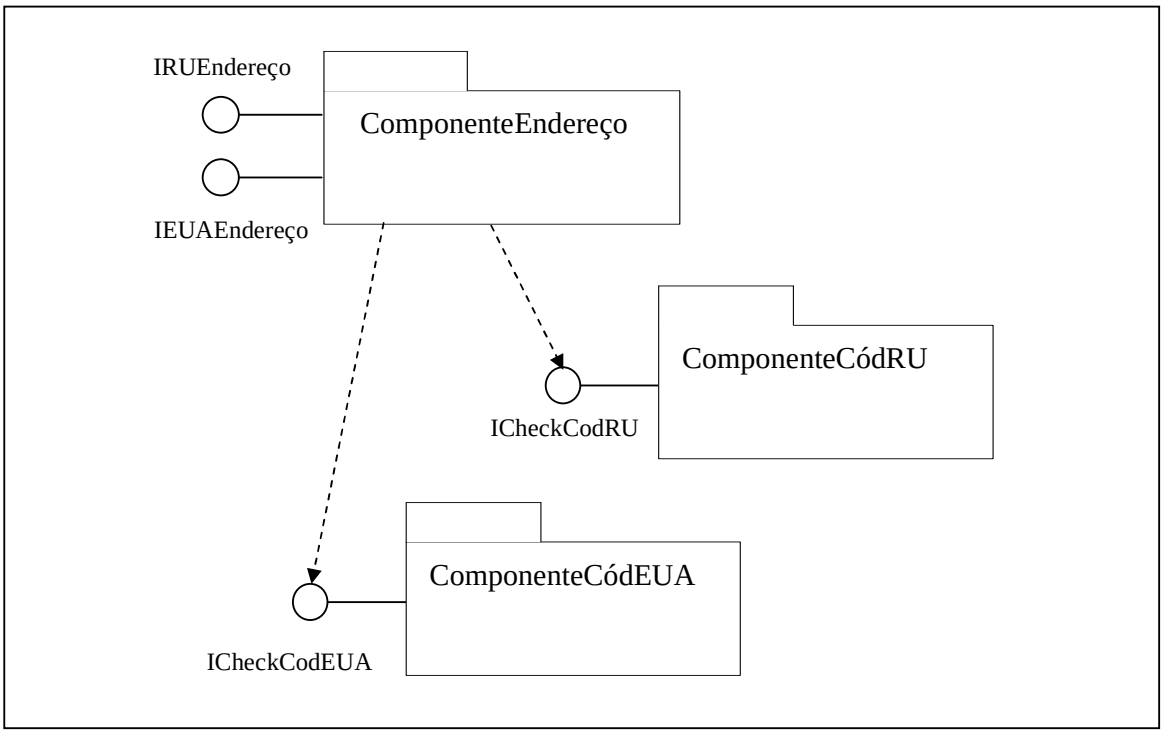


Figura 2.3 – Ilustração do ComponenteEndereço (BARROCA;GIMENES; HUZITA, 2005)

2.2 Modelos de componentes

Para Barroca, Gimenes e Huzita (2005, p.21), a tecnologia de componentes é uma realidade e encontram-se disponíveis métodos, ferramentas e *frameworks*, que apóiam os

projetos de desenvolvimento de software baseado em componentes. A padronização em projetos é muito importante, pois traz uma série de benefícios, dentre eles uma documentação sobre informações importantes tanto para quem desenvolve o componente, quanto para quem o utiliza.

Os modelos de componentes definem formas e interfaces padronizadas entre os componentes. Destacam-se como mais difundidos os seguintes modelos de componente: CORBA (*Common Object Request Broker Architecture*), COM (*Component Object Model*) e JB (*Java Beans*) (CRNKOVIC; LARSSON, 2001).

O modelo CORBA foi desenvolvido pela OMG (*Object Management Group).* Permite, independente da localização do componente no sistema, a comunicação entre componentes reusáveis (objetos) e outros componentes, por meio de serviços oferecidos pelo negociador de solicitação entre objetos (*Object Request Broker*, ORB). Para garantir a integração dos componentes em um sistema, devem ser criadas interfaces para cada componente, em linguagem de definição de interface (*Interface Definition Language*, IDL).

O modelo COM foi desenvolvido primeiramente pela Microsoft, mas é um padrão aberto. Promove a comunicação entre componentes produzidos por diferentes fornecedores em uma aplicação. Existem dois elementos na tecnologia COM, as interfaces e os mecanismos responsáveis por registrar a passagem de mensagens entre as interfaces.

O modelo JavaBeans foi desenvolvido, na linguagem Java, pela Sun Microsystems e é uma tecnologia independente de plataforma. Essa tecnologia possui um conjunto de ferramentas (*Bean Development Kit*) permitindo aos desenvolvedores muitos benefícios, tais como desenvolver componentes personalizados para aplicações específicas e avaliar o comportamento do componente.

As tecnologias CORBA e JavaBeans são independentes de plataforma, diferente da tecnologia COM que é específica para plataforma Microsoft. Todas propiciam o desenvolvimento e a integração de componentes. Pressman (2002 p.714-715) relata que, motivadas pelos benefícios de reúso e desenvolvimento baseado em componente, as empresas propuseram normas para o desenvolvimento de componentes, mas que não é possível afirmar quem dominará o mercado. O que se nota é que, dependendo do domínio da aplicação ou do ambiente de produção, um modelo pode ser mais adequado que os demais.

2.3 Processos de desenvolvimento de software baseados em componentes

Nos processos de desenvolvimentos de software tradicionais, o produto final é projetado e construído a partir do zero. Em desenvolvimentos de software baseados em componentes, os processos tradicionais devem ser adaptados para acomodar a possibilidade de nem tudo ser construído, atendendo aos requisitos do sistema utilizando-se componentes existentes (BARROCA;GIMENES; HUZITA, 2005).

Segundo Pressman (2002, p.704), o processo de software baseado em componente segue os processos tradicionais até o projeto arquitetural. Nesse ponto, os requisitos são analisados, verificando se há a possibilidade de atendê-los utilizando componentes existentes.

As seções seguintes apresentam duas abordagens de processos para o desenvolvimento de software baseado em componentes: a de Cheesman e Daniels e a de Pressman. Foram escolhidas essas abordagens, pois ambas são muito conhecidas e citadas no meio acadêmico.

2.3.1 Abordagem de Cheesman e Daniels

Cheesman e Daniels (2000) propõem um processo para o desenvolvimento baseado em componente, conforme Figura 2.4. O processo tem as etapas de trabalho (*workflows*) representadas pelas caixas e os fluxos dos artefatos produzidos e consumidos pelas etapas, representados pelas setas.

O processo tem como entrada os requisitos de negócio de um sistema iniciando a etapa de requisitos, em que são produzidos os modelos: conceitual de negócio e casos de uso. Para o levantamento dos requisitos de negócio, Cheesman e Daniels (2000) deixam claro que pode ser utilizada qualquer técnica e esses requisitos são a base para a elaboração do modelo conceitual de negócio.

O modelo conceitual de negócio introduz uma série de termos relacionados ao sistema alvo. Por exemplo, em um sistema de vídeo locadora, esses termos podem ser: sócio, filme, emprestimo, etc. Os autores sugerem que se use o diagrama de classes da UML para representar esses termos, conforme apresentado na Figura 2.5.

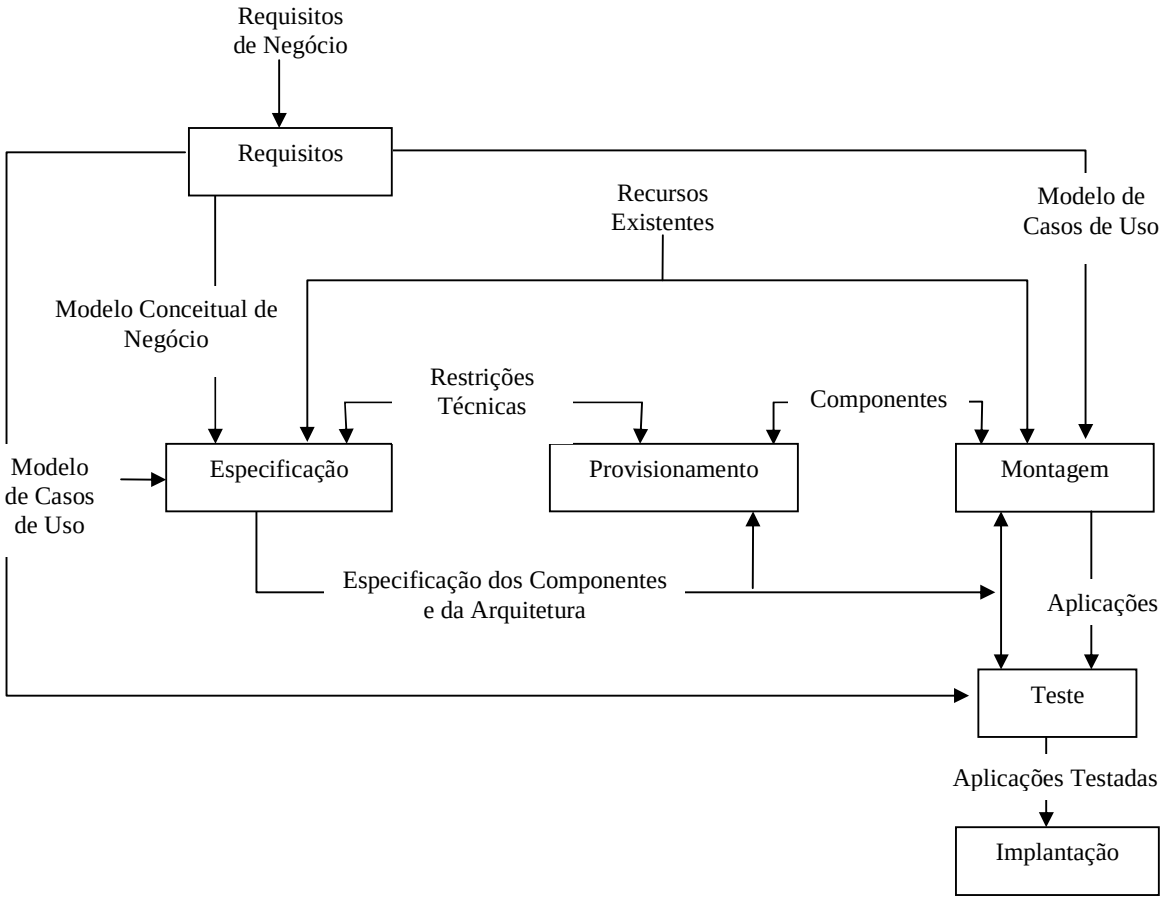


Figura 2.4 – Processo proposto por Cheesman e Daniels (adaptado CHEESMAN; DANIELS, 2000)

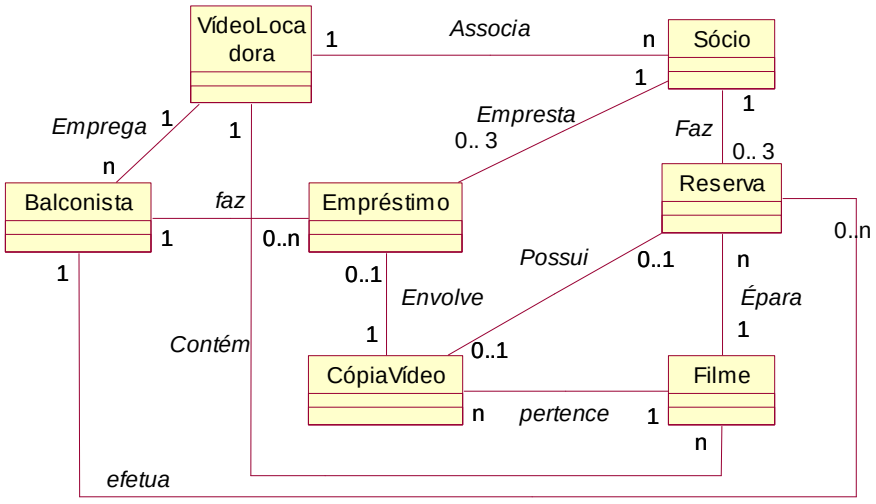


Figura 2.5 – Modelo de negócio para vídeo locadora (BARROCA;GIMENES; HUZITA, 2005).

Após a construção do modelo conceitual de negócio, inicia-se a construção do modelo de casos de uso que é a representação das diferentes funções executadas no domínio da aplicação. Na Figura 2.6 tem-se o ator **Balconista** que, por exemplo, é responsável por **Fazer Reserva**. Também são elaboradas as descrições dos casos de uso, que podem conter os objetivos, as pré-condições e as ações de cada função.

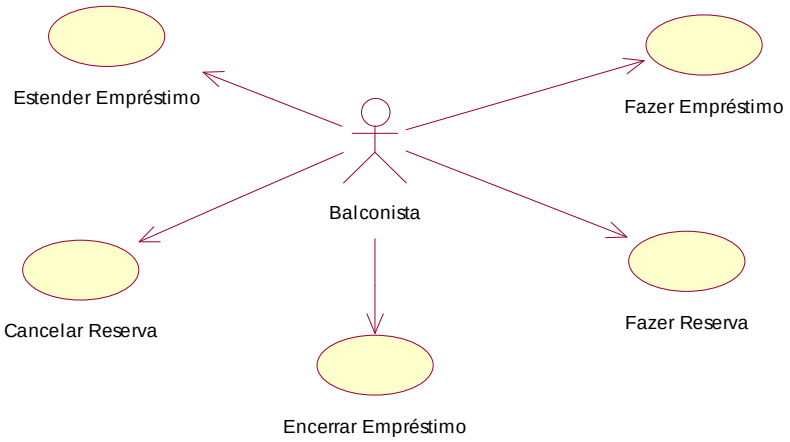


Figura 2.6 – Modelo de casos de uso da vídeo locadora

A etapa de especificação tem como entrada o modelo conceitual de negócio e o modelo de casos de uso e seu objetivo principal é definir os serviços oferecidos pelos componentes. Essa etapa tem três estágios: identificação de componentes, interação entre componentes e a especificação de componentes.

No estágio “identificação de componentes”, as interfaces para os componentes de negócio e componentes de sistema são identificadas. As interfaces e operações do sistema são descobertas analisando os casos de uso. Por exemplo, para o sistema de vídeo locadora sugerem-se as interfaces: IFazEmp (Fazer Empréstimo), IEstEmp (Estender Empréstimo), IEncEmp (Encerrar Empréstimo), IFazRes (Fazer Reserva) e ICancRes (Cancelar Reserva).

A partir do modelo conceitual de negócio é gerado um modelo de tipo de negócio, que pode ser visto como uma evolução do modelo conceitual procurando-se os tipos principais (núcleo) que têm existência independente. Por exemplo, para o sistema de vídeo locadora podem ser identificados os tipos principais: Filme, CópiaVídeo e Sócio. Para cada tipo pode-se definir uma interface de negócio. Por exemplo, para o tipo principal Filme pode-se identificar a interface IFilmeMgt que é responsável pelas operações de adição, remoção,

recuperação de informações e outras. Os demais tipos que não foram classificados como principais devem ser associados às interfaces definidas para os tipos principais.

Também nessa etapa a especificação da arquitetura do componente começa a ser construída, mostrando os componentes e seus relacionamentos com as interfaces, por exemplo, a arquitetura do componente apresentada na Figura 2.7.

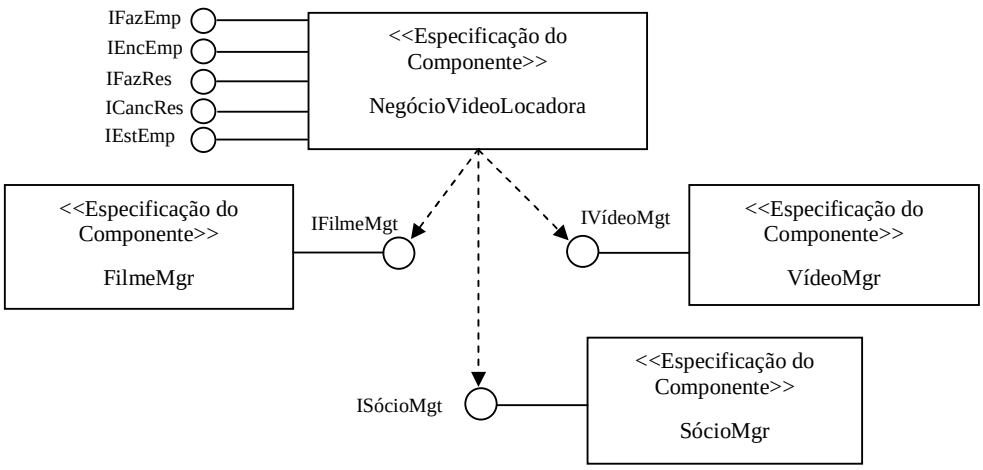


Figura 2.7 – Especificação da arquitetura do componente (BARROCA;GIMENES; HUZITA, 2005)

Os resultados obtidos no estágio de identificação de componentes são utilizados na análise de interação entre os componentes. No estágio “interação entre componentes”, é utilizado o diagrama de colaboração para representar as interações do sistema e, então, definir cada operação do sistema.

As responsabilidades e o detalhamento das interfaces ocorrem na etapa de “especificação de componentes”. Associadas a cada interface são especificadas as operações necessárias, assinatura, pré e pós-condições. A especificação gerada, nesse estágio, propicia uma visão precisa das operações, interfaces e componentes do sistema. Na Figura 2.8 é apresentada a especificação da interface IVídeoMtg e também os tipos referenciados por ela e que pertencem a outra interface. As próximas etapas são a montagem, os testes e a implantação da aplicação. Para a montagem utiliza-se o diagrama de casos de uso e os componentes para compor uma nova aplicação ou integrá-los em uma aplicação existente.

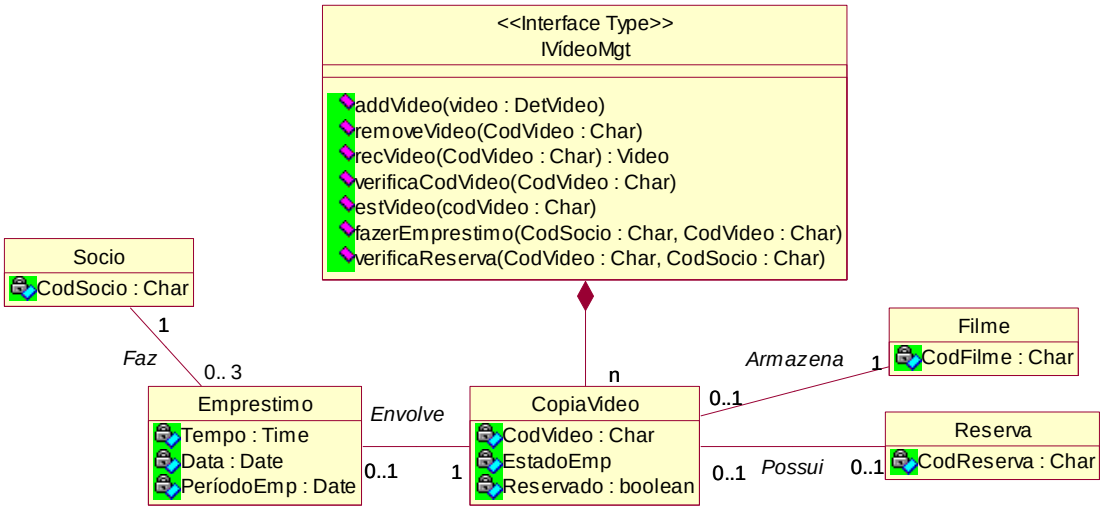


Figura 2.8 – Diagrama de especificação da interface IVídeoMgt (BARROCA;GIMENES; HUZITA, 2005)

2.3.2Abordagem de Pressman

Pressman (2002 p.706-707) propõe um modelo de processo composto pela engenharia de domínio e o desenvolvimento baseado em componentes, cujas atividades acontecem de forma paralela. Conforme mostrado na Figura 2.9, a engenharia de domínio faz toda gestão de um conjunto de componentes, permitindo o compartilhamento dos mesmos (reúso) em projetos existentes ou futuros. Podem-se destacar três atividades principais nesse processo: análise, construção e disseminação do domínio.

Na análise do domínio faz-se o levantamento dos requisitos comuns de um domínio de aplicação específico, com o objetivo de reúso em várias aplicações semelhantes. Pressman (2002, p.729-732) afirma que o desenvolvimento baseado em componente, geralmente utiliza a arquitetura cliente/servidor e que o engenheiro de software deve decidir como será a distribuição dos componentes, que constituem os subsistemas, entre o cliente e o servidor.

A interação da engenharia de domínio com o desenvolvimento baseado em componentes se dá por meio de:

- um modelo do domínio de aplicação que é utilizado para a análise dos requisitos do usuário;

- um modelo estrutural ou arquitetura de software genérica, é que fornece a entrada para o projeto da aplicação;
- repositório, onde os componentes reusáveis estão disponíveis para serem utilizados.

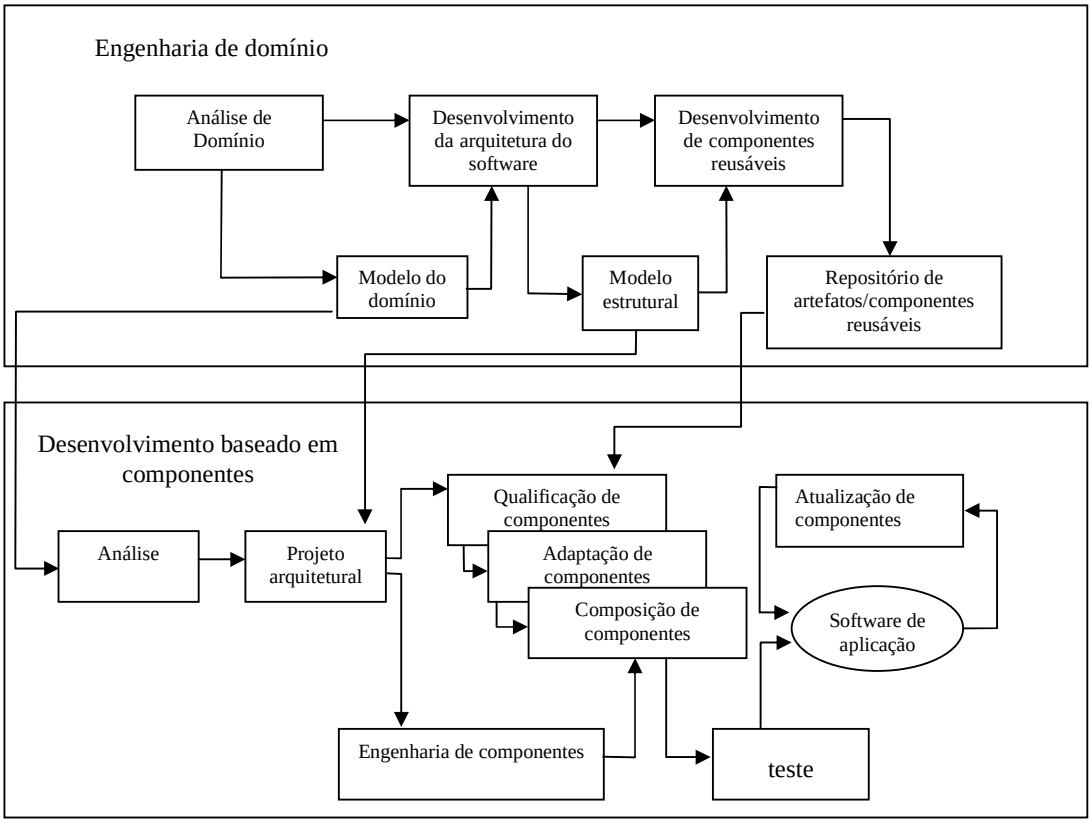


Figura 2.9 – Um modelo de processo que suporta CBSE (*component-based software engineering*) (PRESSMAN, 2002)

No processo de desenvolvimento baseado em componentes, a arquitetura é adequada ao modelo de análise da aplicação, onde deve ser preenchida por componentes. Nesse ponto do processo, podem existir componentes do repositório que satisfaçam às necessidades da aplicação, ou pode ser que seja necessária a construção de componentes novos.

2.4 Benefícios e problemas no desenvolvimento baseado em componentes

Existem vários benefícios na utilização de componentes e na sua reutilização, assim como problemas. Sommerville (2003 p.261) aponta a confiabilidade como sendo um benefício, pois quanto mais o componente é reutilizado, mais experimentado e testado ele foi, reduzindo assim a possibilidade de falhas. Já, Barroca, Gimenes e Huzita (2005 p.8) alertam que para se ter confiabilidade nos componentes, é necessário garantir que testes foram realizados em ambiente afim, antes de utilizá-los, bem como o fornecimento de documentação sobre os testes realizados. Prova disso, segundo Weyuker (1998) é o acidente com "Ariane5", foguete que explodiu após seu lançamento devido a uma falha relacionada à falta de teste em um componente reutilizado de um projeto anterior.

Para Barroca, Gimenes e Huzita (2005, p.7), a utilização de componentes facilita a manutenção e atualização nos sistemas, visto que se torna mais fácil a identificação e a localização da modificação. Por outro lado, Sommerville (2003, p.262) afirma que a manutenção pode ser dificultada quando se utilizam componentes cujo código-fonte não está disponível e com isso se eleva o custo com manutenção.

Os riscos de processo, como estouro de prazo e orçamento, podem ser reduzidos com o reúso de componentes, pois as incertezas relacionadas aos custos são menores. Em adição, o reúso de componentes aumenta as possibilidades de fornecer sistemas mais rapidamente para o mercado, em função da redução de tempo no seu desenvolvimento (SOMMERVILLE, 2003, p.261).

Gerenciar a complexidade pode se tornar mais fácil com a utilização de componentes pois possibilita que se lide com um número reduzido de componentes de cada vez e, também, é possível decompor o sistema em componentes independentes, viabilizando a terceirização de parte deles, concentrando o desenvolvimento nas interfaces e conectores entre os componentes (BARROCA; GIMENES; HUZITA, 2005, p.7).

Barroca, Gimenes e Huzita (2005, p.8) e Sommerville (2003, p.262), concordam que pode ser um problema encontrar componentes adequados em bibliotecas de componentes e que, muitas vezes, engenheiros de software tendem a criar suas próprias soluções, relutando em utilizar códigos desenvolvidos por terceiros.

Para Vincenzi, *et al* (2005) a atividade de teste é importante em todo processo de desenvolvimento de software, servindo para revelar defeitos em produtos que estão sendo

testados. Os problemas com teste de componentes, visto sob a perspectiva do desenvolvedor do componente, são similares aos problemas com testes para outras abordagens de desenvolvimento, que não seja a de componentes. Sob a perspectiva do usuário do componente, os problemas se agravam, visto que nem sempre o código fonte do componente está disponível e então, somente algumas técnicas de teste podem ser utilizadas.

Discute-se mais sobre teste de componentes no Capítulo 3.

2.5 Considerações finais

Neste capítulo foram apresentados conceitos básicos sobre componentes para melhor compreensão do contexto de desenvolvimento baseado em componentes. Para tanto foram apresentadas definições de componente, a forma pela quais os componentes interagem e os modelos de componentes disponíveis no mercado.

Em seguida, apresentaram-se modelos de processo, vantagens e problemas para o desenvolvimento de software baseado em componente, iniciando-se assim um estudo do problema com teste neste contexto. Apesar dos modelos de processo estudados apresentarem em seu contexto etapas para teste, não trazem detalhes de como seriam realizados.

3. TESTE EM DESENVOLVIMENTO BASEADO EM COMPONENTES

Para minimizar a ocorrência de erros, algumas atividades têm sido incorporadas ao longo do processo de desenvolvimento de software, entre elas a de teste. Em desenvolvimento baseado em componentes, é importante utilizar componentes confiáveis e que se relacionem de forma segura. Para isso, deve ser possível testar os componentes individualmente e integrados. Também, o fato de se reutilizar componentes não descarta a necessidade de testes, quer sejam COTS ou não, a fim de garantir que os mesmos tenham o comportamento esperado na nova aplicação. Um exemplo clássico deste fato é o desastre ocorrido em 1996, quando o foguete Ariane 5 explodiu poucos minutos após o seu lançamento. A explosão ocorreu por um comportamento inesperado de um componente reutilizado do foguete Ariane 4 sem os devidos testes (WEYUKER, 1998).

A atividade de teste para o desenvolvedor do componente é muito parecida com a desempenhada em desenvolvimento tradicional, pois testes de unidade e integração devem ser executados para aumentar a credibilidade do usuário com relação ao componente (VINCENZI *et al*, 2005).

São basicamente quatro etapas no processo de teste: planejamento, elaboração dos casos de teste, execução e avaliação dos resultados e três fases de teste podem ser utilizadas: de unidade, integração e sistema.

Para Vincenzi *et al* (2005), o teste de unidade objetiva revelar erros de lógica e implementação nas menores unidades do projeto, que em programas procedimentais podem ser uma sub-rotina ou um procedimento e em programas orientados a objetos, o método. O teste de unidade pode se tornar dispendioso, visto que uma unidade não é um programa isolado e como é mostrado na Figura 3.1, muitas vezes é necessária a implementação de *drives* e *stubs* para que se possa realizar os testes na unidade (*F*). O *driver* é responsável por coordenar o teste da unidade, pois recebe os dados de entrada e passa-os por parâmetro para a unidade (*F*), depois coleta e apresenta as saídas produzidas pela unidade (*F*). O *stub* é uma simulação do comportamento das unidades que interagem com a unidade (*F*).

Ao término do teste de unidade, pode-se iniciar o teste de integração, que visa a testar a comunicação entre as unidades, procurando detectar erros associados às interfaces das mesmas. É uma forma de garantir que funcionem bem em conjunto e que um comportamento inesperado de uma unidade não venha a prejudicar o todo.

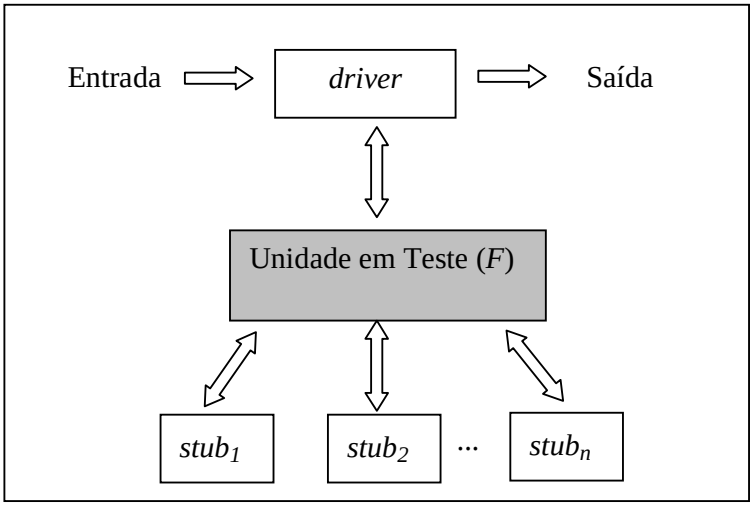


Figura 3.1 – Ambiente para teste de unidade (VINCENZI, et al, 2005)

Após a integração do software, o teste de sistema é iniciado e nele procura-se encontrar erros de funcionalidades e desempenho que não estejam conforme a especificação, ou seja, é uma forma de garantir que software, hardware e banco de dados tenham um desempenho global desejado.

Na Figura 3.2 apresenta-se o relacionamento entre teste de unidade, de integração e sistema, comentados anteriormente, bem como os elementos utilizados nas fases de teste para os dois contextos: programas procedimentais e orientados a objetos.

A eficiência dos testes, ou seja, aumentar a probabilidade de encontrar erros está fortemente ligado aos casos de teste utilizados. Para tanto, existem critérios de teste de software que apóiam a elaboração dos mesmos, visto que é impraticável testar todo o domínio de entrada (MALDONADO, *et al* 2004).

Apesar das peculiaridades do software baseado em componentes, critérios de teste tradicionais podem ser utilizados nos testes de componentes. Três técnicas de teste são as mais utilizadas: funcional, estrutural e baseada em erro, que diferem entre si pela origem da informação utilizada na avaliação e construção dos casos de teste.

A seguir algumas técnicas e critérios tradicionais serão abordados. Em seguida são apresentados alguns trabalhos que abordam especificamente o teste de componente e que possuem maior relação com o presente trabalho, ou seja, que procuram tratar do problema da troca de informação entre desenvolvedor e usuário.

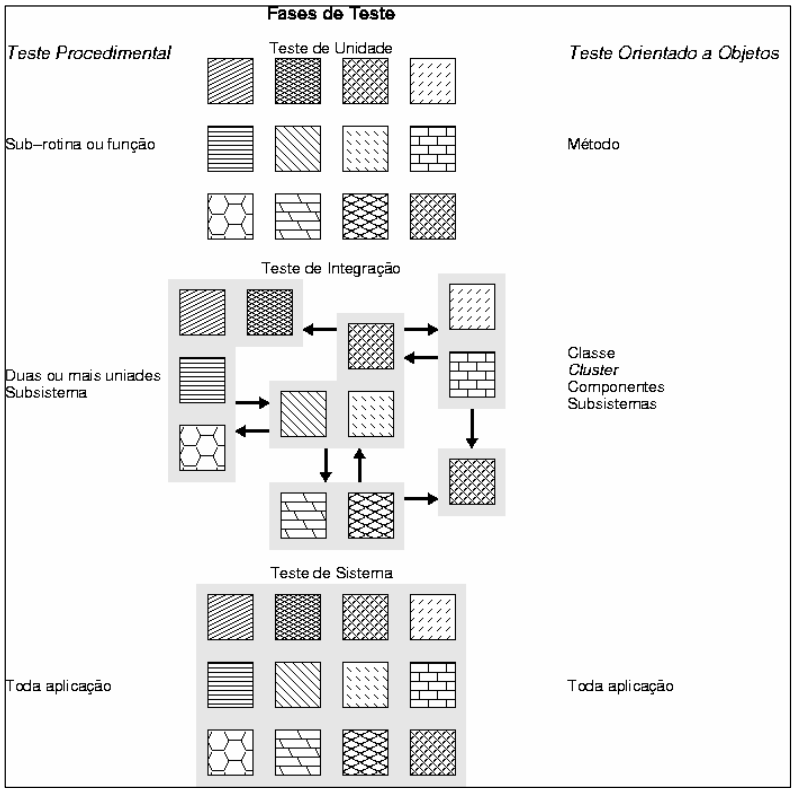


Figura 3.2 – Relacionamento entre teste de unidade, de integração e de sistema: programas procedimentais e orientados a objetos (VINCENZI et al, 2005)

3.1 Técnica Funcional

Segundo Sommerville (2003 p.378), na técnica funcional os requisitos de teste são derivados a partir da especificação do programa ou componente. Essa técnica também é chamada de caixa-preta, pois o comportamento do sistema só pode ser avaliado por meio de suas entradas e saídas. O objetivo principal está focado nas funcionalidades do sistema e não em aspectos de implementação. Pressman (2002 p.450) afirma que a técnica funcional procura revelar erros nas funções incorretas ou omitidas, erros de interface, erros de estrutura de dados ou de acesso à base de dados externa, erros de comportamento ou desempenho e erros de inicialização e término. A técnica funcional pode ser utilizada em qualquer fase de teste.

Sommerville (2003 p.378) explica que a técnica funcional, como mostrado na Figura 3.3, baseia-se na execução do componente ou sistema com dados de entradas que possuam grande chance de provocarem falhas. As saídas são analisadas e no caso de não estarem conforme o previsto, o teste revelou um defeito. O grande problema com essa abordagem é selecionar entradas com possibilidade de provocar comportamento anômalo (Ie). Muitas vezes as entradas são selecionadas com base na experiência dos engenheiros de teste, porém os critérios da técnica funcional podem complementar esse conhecimento prévio.

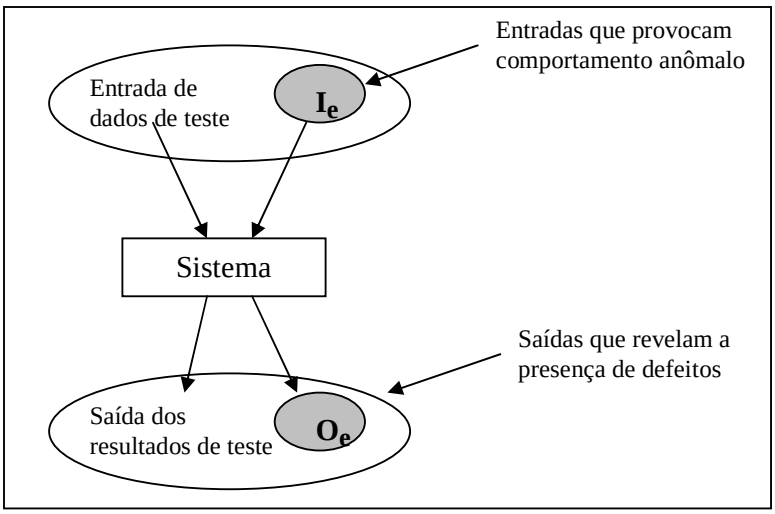


Figura 3.3 – Teste funcional (caixa-preta) (SOMMERVILLE, 2003)

Um exemplo de critério da técnica funcional é a construção do grafo causa-efeito. Primeiramente, são levantadas as possíveis condições de entradas (causas) e as possíveis ações (efeitos) do programa. Em seguida é construído um grafo, com simbologia própria, relacionando as causas e os efeitos levantados. Esse grafo é convertido em uma tabela de decisão que é utilizada na construção dos casos de teste (PRESSMAN, 1995 p.820-824).

Outro exemplo é o particionamento em classe de equivalência, que procura dividir o domínio de entrada em classes que têm características comuns. A execução do programa com qualquer elemento de uma determinada classe produz o mesmo resultado, ou seja, se um dado de teste encontra um erro, todos os outros dados pertencentes à mesma classe devem encontrar o mesmo erro. Exemplos de características comuns que podem ser levadas em

consideração para se constituir as classes de equivalência: números positivos, números negativos, etc. (SOMMERVILLE, 2003 p.378-379).

A análise do valor limite acrescenta algumas características ao critério de partição em classes de equivalência. Consiste em escolher um representante da classe, mas levando-se em consideração os valores que compõem as fronteiras entre as classes, pois é onde concentram-se um grande número de erros. Além disso, o espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída (SOMMERVILLE, 2003 p.379-380).

3.2 Técnica Estrutural

Pressman (2002 p.436) explica que a técnica estrutural, conhecida também por teste caixa-branca, em oposição à técnica funcional (caixa-preta) tem como requisito básico o código fonte do programa, visto que o objetivo é exercitar as estruturas internas do mesmo. Os casos de teste derivados da técnica estrutural visam a garantir que todos os caminhos independentes de um programa sejam exercitados pelo menos uma vez, exercitar todas as decisões lógicas e exercitar também as estruturas de dados garantindo sua validade. A técnica estrutural é mais utilizada em testes de unidade.

Maldonado *et al* (2004), afirmam que, em geral, a técnica de teste estrutural utiliza uma representação para programas conhecida como grafo de fluxo de controle ou grafo de programa. O grafo de programa é um grafo orientado onde cada vértice representa um bloco indivisível de comandos e cada aresta representa um desvio de um bloco para outro. Um bloco desse tipo tem as seguintes características: não existem desvios para o meio do bloco e uma vez que o primeiro comando do bloco seja executado, todos os demais comandos do bloco são executados sequencialmente.

Com base no grafo de programa podem ser escolhidos os componentes a serem executados, caracterizando assim os critérios de teste estrutural. Destacam-se dois tipos de critérios: baseado no fluxo de controle e baseado no fluxo de dados.

O critério baseado no fluxo de controle usa apenas características de controle de execução do programa, como comandos ou desvios, para determinar quais estruturas são requeridas. Como exemplo, pode-se citar os critérios **todos os nós** e **todos os arcos**. O

primeiro exige que a execução do programa passe pelo menos uma vez por cada vértice do grafo de programa e o segundo requer que cada aresta do grafo (desvios) seja executada pelo menos uma vez (PRESSMAN, 2002 p.436).

O critério baseado no fluxo de dados explora associações entre pontos do programa onde é atribuído um valor a uma variável (definição da variável) e pontos onde esse valor é utilizado (referência ou uso da variável). Com base nessas associações são determinados os caminhos a serem exercitados. Por exemplo, o critério **todos os usos** que exige que pelo menos um caminho entre todas as associações definição-uso de cada variável seja exercitado pelos casos de teste (PRESSMAN, 2002 p.447).

3.3 Técnica Baseada em Erros

DeMillo (1980) afirmam que a técnica baseada em erros utiliza a informação de erros mais cometidos pelos programadores, durante o desenvolvimento de software, para estabelecer os requisitos de teste. Dentre os critérios de testes baseados em erros, pode-se citar: Semeadura de Erros e Análise de Mutante.

Burnstein (2002 p.116) explica que o critério Análise de Mutantes consiste em aplicar pequenas alterações no código fonte do programa ou componente, que são denominados mutantes. Essas pequenas alterações (mutações) são falhas instaladas em cada mutante. Segundo Delamaro e Maldonado (1993), existem quatro passos para aplicar-se esse critério, partindo da premissa de que se tem um programa e um conjunto de casos de teste os quais se deseja avaliar. Os passos são:

- geração dos mutantes;
- execução do programa original;
- execução dos mutantes;
- análise dos mutantes vivos.

Burnstein (2002 p.116-117) explica que executam-se os testes no programa original e nos programas mutantes com os mesmos casos de teste e avaliam-se os resultados obtidos. Caso os resultados obtidos no programa original sejam diferentes dos resultados obtidos nos

programas mutantes, diz-se que o mutante está morto. Caso contrário o mutante vivo deve ser avaliado.

A avaliação dos mutantes vivos em geral requer a intervenção humana, na qual se pode chegar a duas conclusões: ou os casos de teste não foram capazes de revelar defeitos e então, é necessário elaborar mais alguns casos de teste, ou o programa original e o mutante são equivalentes e por isso apresentam os mesmos resultados.

DeMillo (1980) destaca a existência de uma medida para avaliar a adequação dos casos de teste utilizados para testar o programa, chamada escore de mutação. Por meio do escore de mutação, que relaciona o número de mutantes gerados com o número de mutantes mortos, pode-se avaliar a adequação dos casos de teste usados e, como conseqüência, a confiabilidade do programa testado.

Como citado anteriormente, o uso desses critérios de teste para a geração e avaliação de conjuntos de teste para desenvolvimento baseado em componente é perfeitamente factível. Tanto o desenvolvedor quanto o usuário do componente podem fazê-lo. O problema, nesse escopo não é apenas criar casos de teste adequados para uma aplicação baseada em componentes, mas garantir que a aplicação e componente se integram corretamente, coisa que as técnicas apresentadas, por si só, não fazem. Nas seções seguintes discutem-se trabalhos que abordam esse tema.

3.4 Desenvolvimento baseado em componentes: o problema de comunicação

Diversos autores identificaram um sério problema proveniente da característica do desenvolvimento baseado em componente que é a falta da troca de informação entre desenvolvedor e usuário do componente. Essa falta de informação afeta principalmente o usuário do componente na realização dos testes. Como esse trabalho dá ênfase a este aspecto, esta seção destaca trabalhos relacionados a esse tema.

Beydeda e Gruhn (2003) afirmam que testar componente e aplicações baseadas em componentes nem sempre é uma tarefa fácil e para abordar as dificuldades encontradas, deve-se olhar o problema sob duas perspectivas: a do desenvolvedor do componente e a do usuário do componente. A troca de informações entre eles é fundamental para o sucesso da utilização do componente, mas nem sempre isso ocorre. A falta de informação, sob a ótica das duas

perspectivas, é considerada como o maior problema ao se testar componentes. O desenvolvedor muitas vezes trabalha com suposições a respeito do ambiente no qual o componente será utilizado e essas suposições influenciam nos testes, visto que o componente pode ser dependente de um contexto e, consequentemente, o mesmo ser testado apenas nesse contexto. O problema da dependência de contexto é que o comportamento do componente em determinados ambientes será o esperado e em outros, não.

Para Beydeda e Gruhn (2003), desenvolvimento de software baseado em componentes requer uma documentação detalhada do componente que está sendo utilizado, porém nem sempre isso ocorre, pois pode ser que a documentação fornecida pelo desenvolvedor do componente esteja incompleta, portanto, não adequada às necessidades do usuário do componente.

A falta de informação pode requerer, também, que o usuário do componente tenha que testar o componente antes de usá-lo e, nesse caso, as dificuldades podem ser ainda maiores, pois nem sempre o código fonte está disponível. Se forem detectadas falhas nos testes realizados pelo usuário do componente, a falta de documentação adequada, código fonte e planos de teste, por exemplo, torna inviável a localização e remoção das mesmas. Quando isso ocorre, o usuário do componente se torna dependente do desenvolvedor do componente (BEYDEDA; GRUHN, 2003).

A seguir serão apresentadas algumas propostas encontradas na literatura para diminuir o problema da falta de informação que afetam os testes em desenvolvimento baseado em componentes. De acordo com Beydeda e Gruhn (2003), essas propostas podem ser classificadas em duas categorias: as que procuram lidar com a falta de informação e outras que propõem facilitar a troca de informações entre o desenvolvedor e o usuário do componente.

3.5 Propostas para facilitar a troca de informações

Para Beydeda e Gruhn (2003), as propostas dessa categoria tratam das causas da falta de informação. Algumas delas facilitam a troca de informações, visando apenas a suprir as carências do usuário do componente. Outras promovem a troca de informação em ambos os sentidos: do desenvolvedor para o usuário e do usuário para o desenvolvedor.

Para Orso, Harrold e Rosenblum (2000), quando os componentes são integrados em uma aplicação, várias atividades são executadas, como análises, testes do sistema e avaliação de qualidade. Para tais atividades, são necessárias informações que vão além do código binário e uma descrição superficial do componente. Os autores propõem o *component meta-data* que consiste na geração de dados adicionais (metadados) de vários tipos, ou seja, os metadados não seriam, necessariamente, iguais em todos os componentes, ficando a critério do desenvolvedor a escolha de quais informações seriam disponibilizadas, levando-se em conta as tarefas ou um contexto específico. Os metadados gerados pelo desenvolvedor são disponibilizados para o usuário do componente.

A idéia de utilizar metadados com os componentes está relacionada com o que a engenharia elétrica faz, por exemplo, com o resistor, que não seria útil sem que algumas características como o valor da resistência, tolerância e empacotamento o acompanhassem. O mesmo se dá com componentes de software que, para serem úteis em diferentes contextos, é necessário que algumas informações os acompanhem. Beydeda e Gruhn (2003) afirmam que as informações disponibilizadas podem ser quaisquer artefatos do desenvolvimento do componente. Informações destinadas a auxiliar o usuário do componente em atividades de testes podem ser, por exemplo, o grafo de programa, já que nem sempre o código fonte está disponível ao usuário do componente para possibilitar a geração do mesmo.

Orso, Harrold e Rosenblum (2000) explicam que há três propriedades básicas dos metadados:

- o desenvolvedor do componente está envolvido na produção dos metadados;
- os metadados são empacotados com o componente de forma padronizada, e;
- o desenvolvimento e a apresentação dos metadados devem ser apoiados por ferramentas.

Beydeda e Gruhn (2003) afirmam que a proposta de gerar metadados pode ser vista como uma generalização do mecanismo de introspecção disponível na maioria dos modelos de componentes, por exemplo, JavaBeans, mas diferem em alguns pontos como o tipo e a flexibilidade de informações. O mecanismo de introspecção é a capacidade de se descobrir características do componente em tempo de execução. Essas características podem ser propriedades, métodos e eventos. A proposta em gerar metadados inclui informações semânticas do componente, além das sintáticas e não existe uma determinação de quais devem ser as informações disponibilizadas, pois serão escolhidas pelo desenvolvedor do

componente. A proposta de geração de metadados, assim como o mecanismo de introspecção, visa a percorrer um caminho unilateral, ou seja, apenas o desenvolvedor do componente disponibiliza metadados para o usuário do componente.

Orso, Harrold e Rosenblum (2000) propõem que seja criada para cada tipo de metadado uma identificação única (DTD – *Document Type Definition*), a fim de que o usuário do componente ou uma ferramenta identifique o tipo de metadado e como manuseá-lo. Beydeda e Gruhn (2003) alertam que a complexidade, tamanho e dependência de contexto dos metadados devem ser analisadas, pois, dependendo do resultado dessas análises, por questões de flexibilidade, talvez seja necessário entregar os metadados separados do componente, ou seja, que os mesmos não estejam empacotados junto com o componente.

Ma *et al* (2001) desenvolveram o protótipo de um conjunto de ferramentas de teste para automatizar o processo para testes em COTS. Partem do pressuposto que o desenvolvedor do componente fornece metadados, que são informações adicionais disponibilizadas junto com o componente. Essas informações podem ser, por exemplo, sobre as interfaces do componente, descritivos sobre as funcionalidades do componente, ou sobre os testes realizados. Nesse último caso podem incluir a descrição dos testes, conjunto de testes, *stubs* e ainda fornecer informações sobre cobertura obtida com os testes e controle de dependências.

Algumas dessas informações ajudam a executar os testes funcionais no componente sem que se tenha o entendimento de detalhes de implementação e a construir seus casos de teste. Porém, construir seus casos de teste pode aumentar significativamente o esforço despendido pela área de teste. Outras informações, como dados de cobertura, podem ser utilizados para verificar se os casos de teste elaborados pelo desenvolvedor do componente estão adequados segundo o critério pré-estabelecido.

O processo proposto por Ma *et al* (2001) para o teste de COTS consiste em 3 etapas: na primeira etapa a área de teste fornece as diretrizes para os testes ao desenvolvedor do componente. Nessas diretrizes constam informações sobre atividades requeridas, informações sobre as técnicas de teste que serão usadas, qual deve ser o conteúdo do pacote de teste, o método pelo qual o pacote de teste deve ser elaborado e um guia do usuário contendo as informações sobre as ferramentas de suporte. Na segunda etapa o desenvolvedor do componente gera o pacote de teste de acordo com as diretrizes fornecidas pela área de teste, que engloba as atividades de geração de casos de teste, execução dos testes, elaboração de descrições dos casos de teste, captura dos dados de cobertura, etc. Na terceira etapa a área de teste de posse do pacote de teste fornecido pelo desenvolvedor do componente, faz a auditoria

nesse pacote, executa os testes e avalia a cobertura obtida com a informada no metadado. Com o valor de cobertura calculado é possível avaliar o quanto o desenvolvedor do componente seguiu as diretrizes fornecidas pela área de teste. Por fim, é gerado um relatório com as informações sobre a execução dos testes, passos seguidos, falhas encontradas e detalhes da execução de cada caso de teste.

Apesar da proposta de Ma *et al* (2001) não se tratar da troca de informações entre desenvolvedor e usuário do componente e sim entre desenvolvedor do componente e área de teste, considera-se pertinente tratar essa proposta aqui, em função de se trocar informações sobre o componente utilizando metadados.

Bhor (2001) fez um estudo geral sobre teste de componente e avaliou várias estratégias de teste, dentre elas a proposta de utilização de metadados. Para a estratégia que utiliza metadados, chegou à seguinte avaliação:

- aumenta a precisão para análises do programa;
- os metadados podem ser customizados e fornecidos de acordo com as necessidades do usuário do componente, mostrando flexibilidade no armazenamento das informações;
- requer o desenvolvimento de notação padrão para os metadados, no que acrescenta uma grande dificuldade, visto que os produtores de COTS devem segui-las;
- nos trabalhos analisados sobre a utilização de metadados, apenas foram referenciados testes em pequenos programas. Escalabilidade não foi mencionada em nenhum deles, e;
- a proposta de utilizar metadados traz facilidades ao usuário, de modo que os testes podem ser realizados de maneira sistemática e conveniente.

Porém, Bhor (2001) em sua conclusão alerta que apesar da estratégia de teste que propõe fornecer metadados ser uma solução com grande potencial, padronizar os metadados e a forma de disponibilizá-los depende muito da cooperação e coordenação de vários produtores de componentes.

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.

Liu e Richardson (1998) explicam que um *retro-component* é um componente de software com um *retrospector*, no qual o *retrospector* é uma generalização do *introspector*, que é um mecanismo utilizado em alguns modelos, como por exemplo, no JavaBeans onde informações estáticas do componente são gravadas. A diferença entre os dois mecanismos, *retrospector* e *introspector*, está no tipo de dados gravados e disponibilizados e o fluxo da informação entre desenvolvedor e usuário do componente. No *introspector* apenas dados estáticos são gravados e disponibilizados, já no *retrospector*, dados estáticos e dinâmicos.

O fluxo da informação no *introspector* é unilateral, ou seja, do desenvolvedor para o usuário e no *retrospector* é bilateral, portanto os benefícios oferecidos pelo *retrospector*, contemplam tanto o desenvolvedor, como o usuário do componente. O desenvolvedor consegue conhecer melhor o contexto de utilização do componente e o usuário tem acesso a informações importantes, facilitando os testes de sua aplicação.

Harrold, Liang e Sinha (1999) propõem que o desenvolvedor do componente execute seus testes e análise para o componente e forneça um resumo dessas atividades junto com o componente, a fim de facilitar a integração do componente à aplicação do usuário do componente. Para facilitar a atividade de teste e análises dinâmicas do usuário do componente em sua aplicação, é necessária a utilização de uma ferramenta que capture as informações do componente em uma determinada execução, separando as entradas de cada operação em um conjunto de subdomínios e associando esses subdomínios às informações fornecidas pelo desenvolvedor do componente. Sugerem também para esse método o uso do *retrospector*.

O resumo das informações dos testes e análises fornecidas pelo desenvolvedor do componente pode estar em uma linguagem independente da utilizada para desenvolver o componente e sugerem três tipos de informação: 1) as produzidas pela técnica *Program slicing* representada, por exemplo, pelas informações de dependência entre as variáveis de entrada e saída para cada operação; 2) as produzidas por análises de dependência de controle, representada, por exemplo, pelas exceções que podem ser causadas pelo componente; e 3) as produzidas por teste de fluxo de dados, representada, por exemplo, pelas informações do

conjunto de usos das variáveis de entrada, no início de um módulo e também o conjunto de definições que alcançam o final do módulo associados ao subdomínio coberto pelas entradas.

Edwards (2001) propõe *reflective wrapper*, no qual as informações são empacotadas pelo desenvolvedor do componente utilizando a técnica *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 são sobre a especificação do componente e ações de garantia da qualidade utilizada pelo desenvolvedor, podendo ser disponibilizada, como um outro componente ou um conjunto de componentes. As informações podem ser recuperadas em uma forma legível para humanos ou processadas por ferramentas, tipo *browsers* e o fluxo de informação dessa proposta é unilateral, ou seja, do desenvolvedor do componente ao usuário do componente.

Edwards (2001) explica que a técnica *wrapper* encapsula o componente, mas de forma transparente para o usuário do componente e que após a implementação de algumas interfaces e processamentos, o usuário do componente pode requisitar informações, que serão apresentadas em forma de metadados e utilizá-las nos testes. Usar a técnica *wrapper* possibilita ao usuário do componente acrescentar ou remover informações sem acessar o código fonte.

Bundell *et al* (2000) propõem o *component test bench*, que da mesma forma que nas outras propostas comentadas anteriormente, sugere que se forneçam informações adicionais sobre o componente para apoiar as atividades de análises e testes. Para isso, deve-se preparar uma especificação de teste que descreva implementações do componente, interfaces fornecidas para cada implementação e um conjunto de testes apropriados para testar as interfaces. Um elemento do conjunto de testes é chamado de operação de teste, que é uma seqüência de passos para a execução de teste em um método específico em uma das interfaces do componente. Essas informações ajudam o usuário do componente na realização dos testes.

Bundell *et al* (2000) também propõem que a especificação de teste seja 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.

3.6 Propostas que lidam com os efeitos da falta de informação

Para Beydeda e Gruhn (2003), as propostas dessa categoria procuram lidar com prováveis dificuldades que podem ser encontradas ao executarem-se os testes nos componentes, ou seja, não tem como objetivo atacar as causas e sim, os efeitos.

Beydeda e Gruhn (2003) relatam que os componentes podem ser acrescidos de casos de teste ou possuir a capacidade de gerar casos de teste. Esses casos de teste podem ser acessados pelo usuário do componente ou o componente pode usá-los para se auto-testar. Esse mecanismo é chamado de *built-in test* (BIT). Desta forma, o usuário do componente não precisa gerar casos de teste para a integração do componente em sua aplicação.

Wang, King e Wickburg (1999) propõem dois modos de operação para o BIT: modo normal e modo manutenção. No modo normal o BIT não está habilitado no componente e no modo manutenção o usuário do componente pode usar o BIT para executar os testes no componente, bastando para isso efetuar a chamada aos métodos do componente e, logo após, avaliar os resultados dos mesmos.

Hörnstein e Edler (2002) relatam que a proposta de usar BIT pode ocasionar um acréscimo significativo ao componente, visto que os casos de teste e suas descrições são armazenados no próprio componente e muitos destes casos de teste não agregam valor até que sejam executados no contexto da aplicação em que o componente será utilizado. Esse acréscimo pode ser diminuído armazenando-se os casos de teste fora do componente, sendo o BIT apenas fornecedor das informações necessárias para testar o componente. Assim, Hörnstein e Edler (2002) propõem o *component+*, que é uma arquitetura constituída por três tipos de componentes: BIT *components*, *testers* e *handlers*. Os BIT *components*, são os componentes capacitados com o mecanismo *built-in test* e fornecem uma ou mais interfaces, denominadas BIT *interfaces. Testers* são componentes que contém os casos de teste e usam as BIT *interfaces* para testar e verificar o componente. Os *testers* podem ser desenvolvidos e mantidos independentes dos componentes que estão sendo testados e, portanto, evita-se o acréscimo ao código do componente. *Handlers* não contribuem efetivamente nos testes, mas podem ser vistos como notificadores de erros e usados para obter um sistema tolerante a falhas.

Gao, Gupta e Shim (2002) explicam que a proposta *testable Beans* está ligada à testabilidade do componente, que depende da habilidade do mesmo em suportar a execução

dos testes e promover a sua observação. Isso pode ser possível aumentando-se o componente com essa capacidade para atender essas atividades. O modelo de componente e *framework* dessa proposta é o *Enterprise JavaBeans* (EJB) que precisa satisfazer certos requisitos e aspectos para se tornar um componente *testable beans.* Dentre esses requisitos e aspectos, pode-se ressaltar:

- deve ser usado exatamente como qualquer outro componente e não requerer preparativos específicos para sua operação;
- deve ser rastreável, ou seja, permitir que o usuário do componente observe o seu comportamento durante os testes;
- deve implementar interfaces consistentes e bem definidas que permitam o acesso a capacidade de teste e;
- deve possibilitar a interação com ferramentas externas de teste.

Para o usuário do componente, o teste de interface é a grande diferença entre um componente comum e um componente *testable beans*. O teste de interface declara três métodos, um que inicializa os testes na classe e método a serem testados, bem como os casos de teste, outro que executa os testes e o último que avalia os testes. Esse teste de interface pode ser utilizado por ferramentas de teste ou para auto-testar o componente *testable beans* (GAO; GUPTA;SHIM, 2002).

3.7 Outros trabalhos

Diversos aspectos do teste de software baseado em componentes têm sido estudados, além dos destacados neste capítulo; nesta seção são comentados alguns desses trabalhos.

Voas (1998) destaca a necessidade de avaliar o quanto um componente tem qualidade e qual é o impacto que o componente causa ao ser utilizado na aplicação. O autor propõe uma metodologia de certificação para COTS. Foram elaboradas três questões: A) O componente tem as funcionalidades necessárias para a aplicação? B) O componente apresentou qualidade? C) O componente produziu um impacto positivo na aplicação?

Fazendo uma combinação de respostas sim e não para as questões elaboradas, o autor chegou a cinco cenários possíveis, conforme a Tabela 3.1. Os componentes que se enquadram

nos cenários 1 e 3 são certificados, no 4 e 5 não podem ser certificados e no 2 devem ser analisados.

Para a certificação são utilizadas três técnica de avaliação de qualidade: o teste da caixa-preta, injeção de falha e o teste de sistema. O teste da caixa-preta é utilizado para determinar se o componente tem qualidade respondendo a pergunta B da Tabela 3.1. A injeção de falha em nível de sistema é utilizada para saber qual a tolerância da aplicação em relação as possíveis falhas do componente. O teste de sistema serve para verificar a tolerância da aplicação em relação às funcionalidades do componente. Os resultados da injeção de falha e do teste de sistema são utilizados para responder as questões A e C da Tabela 3.1.

Voas (1998) afirma que a metodologia determina um cenário para o componente que está sendo certificado e assim, pode-se decidir se o componente será utilizado na aplicação ou não. Por exemplo, se após a certificação o cenário estabelecido para o componente candidato fosse o 4, chega-se a conclusão que o componente é pobre em qualidade e causa um impacto negativo na aplicação.

Tabela 3.1 – Questões relacionadas ao uso do componente na aplicação (VOAS, 1998)

Cenários	Pergunta A	Pergunta B	Pergunta C
1	Sim	Sim	Sim
2	Sim	Sim	Não
3	Sim	Não	Sim
4	Sim	Não	Não
5	Não	N/A	N/A

Wu, Pan e Chen (2001) afirmam que a atividade de teste aprimora a qualidade do software, mas alertam que ao testar software baseado em componente usando as técnicas tradicionais, pode-se encontrar problemas e que novas técnicas são necessárias. Os autores propõem um novo modelo de teste com base em uma nova família de critérios de teste, baseados em prováveis defeitos em desenvolvimento baseado em componente e na

identificação de elementos do sistema que têm grande probabilidade de detectar defeitos nos testes.

Os prováveis tipos de defeitos levantados estão caracterizados em três grupos. O primeiro grupo leva em consideração os defeitos ligados à interação dos componentes que compõem o sistema, o segundo ligado à interoperabilidade dos componentes que são heterogêneos e o terceiro aqueles defeitos tratados nas técnicas tradicionais. Com relação à identificação dos elementos que aumentam a probabilidade de detectar defeitos nos testes foram relacionados: as interfaces, eventos, relações com dependência de contexto e relações com dependência de conteúdo.

Wu, Pan e Chen (2001) propõem como tema central desse modelo de teste o desenvolvimento do CIG (*Component Interaction Graph*) que é construído com base nos elementos de teste comentados anteriormente. Para a identificação dos elementos e elaboração do CIG podem-se utilizar informações conhecidas sobre os componentes e informações adquiridas com um pré-processamento. Por exemplo, para a identificação dos eventos precisam-se saber quais são as chamadas às interfaces que disparam eventos e que respondem a eles.

Como critérios para esse modelo têm-se dois níveis de critério: o nível básico que tem como requisitos todas as interfaces (*all-interfaces*) e todos os eventos (*all-events*) e em um nível mais avançado, no qual tem-se os requisitos onde as interfaces são acessadas em diferentes contextos (*all-context-dependence*) e os requisitos onde é identificada uma seqüência necessária para o acesso à interface, por exemplo, caminhos entre dois eventos dependentes de contexto (*all-context-dependences/some-content-dependences*).

Rocha e Martins (2004) propõem um modelo para a construção de um componente testável, cuja estrutura é desenvolvida pelo desenvolvedor do componente e disponibilizada para o usuário do componente. O componente testável inclui mecanismos de monitoramento e geração automática de casos de teste a partir da especificação.

O objetivo da proposta é aumentar a testabilidade do componente para a aplicação de teste utilizando a técnica funcional. Para isso, são disponibilizados com o componente os mecanismos de teste e monitoração, a especificação do componente na forma assertiva e modelo comportamental.

A arquitetura do componente é composta pelo componente a ser testado e suas interfaces públicas e pelos subcomponentes *Tracker* e *Tester*. O subcomponente *Tracker* é responsável pelas funcionalidades de monitoração e o subcomponente *Tester* é responsável pela disponibilização da especificação.

Freedman (1991) afirma que conhecer o quanto um componente é facilmente testável é importante para planejar os testes e estimar o processo, pois o componente que não é facilmente testável pode necessitar de várias interações de teste. O componente é altamente testável quando encontram-se algumas características: o conjunto de teste é pequeno e fácil de ser gerado, o conjunto de teste não é redundante, as saídas dos testes são fáceis de serem interpretadas e as falhas são facilmente localizadas.

Outra forma de atestar a testabilidade de um componente é feita pelo testador durante os testes, com base nas entradas e saídas, ou seja, se as entradas ou saídas estiverem inconsistentes o componente tem sua testabilidade comprometida.

Freedman (1991) define uma propriedade chamada domínio de testabilidade. Para o domínio de testabilidade são definidas duas propriedades: observabilidade e controlabilidade. Pode-se dizer que um componente tem a primeira propriedade se for fácil de determinar o quanto uma entrada influencia a saída. Pode se dizer que um componente tem a segunda propriedade se for fácil produzir uma especificação de saída para uma especificação de entrada.

3.8 Considerações finais

Neste capitulo fez-se um estudo sobre fases de teste, técnicas e critérios de teste tradicionais, verificando-se a possibilidade de aplicá-las em testes de componentes. Com esse embasamento teórico, procurou-se aprofundar e detectar as causas para o problema da falta de informação nos testes em desenvolvimento baseado em componentes, sentida pelo desenvolvedor e usuário do componente. Buscou-se na literatura propostas para solucionar o problema com falta de informação para a realização dos testes em desenvolvimento baseado em componentes. O estudo dessas propostas é essencial para esse projeto, apesar de não apresentarem de forma objetiva o tipo de metadados a serem fornecidos pelo desenvolvedor, ou como possam ser utilizados pelo usuário.

Por fim foram abordados outros trabalhos que tratam dos diversos aspectos do teste baseado em componente.

4. ESTRATÉGIA DE TESTE PARA DESENVOLVIMENTO BASEADO EM COMPONENTES

Um dos problemas encontrado em desenvolvimento de software baseado em componente é a falta de informações sofrida pelo desenvolvedor e usuário do componente. Uma das atividades mais afetadas por essa falta de informações sobre o componente é a de teste e apesar de existirem várias propostas na literatura abordando formas de solucionar o problema da falta de informação, nenhuma objetivamente trata quais as informações seriam necessárias e nem a forma de usá-las.

Baseado em propostas que descrevem como disponibilizar dados adicionais usando a associação de metadados ao componente, a fim de serem utilizados na integração do componente à aplicação do usuário do componente e na técnica estrutural de teste, será apresentado o embasamento teórico para a estratégia de teste proposta neste trabalho. Também por meio de um exemplo simples será apresentada a automação da estratégia e a forma pela qual pode-se obter benefícios com ela.

Para a aplicação desta estratégia propõe-se que o desenvolvedor do componente forneça metadados empacotados com o componente ao usuário. Nesse contexto, entende-se metadados como sendo as informações de: 1) medidas de cobertura obtidas para cada caso de teste, por método público do componente; e 2) uma descrição informal para cada caso de teste.

O usuário de posse dessas informações pode comparar os resultados de cobertura obtidos por seus casos de teste em relação às coberturas alcançadas pelos casos de teste do desenvolvedor, avaliando assim a adequação dos seus casos de teste em exercitar a integração de sua aplicação ao componente.

Para obter as medidas de cobertura fornecidas nos metadados propõe-se a utilização da análise de cobertura estrutural executada na ferramenta JaBUTi, que foi desenvolvida para apoiar o teste estrutural em programas Java e efetua toda a análise estática e dinâmica do programa em teste a partir do bytecode Java. Também propõe-se a utilização da ferramenta FATEsC, desenvolvida para apoiar a estratégia de teste proposta por este trabalho que possibilita a elaboração de casos de teste, capturar as coberturas fornecidas pela ferramenta

JaBUTi, anexar os metadados ao componente e promover as comparações das coberturas alcançadas pelo usuário em relação as coberturas do desenvolvedor do componente.

Neste capítulo serão apresentados: a premissa deste trabalho, uma breve descrição da ferramenta JaBUTi, a proposta da estratégia, abordando de que forma as ferramentas JaBUTi e FATEsC interagem e as funcionalidades da ferramenta FATEsC, desenvolvida para automatizar a estratégia proposta neste trabalho.

4.1 Premissa

A idéia é utilizar critérios de teste da técnica estrutural como forma de avaliar a qualidade do conjunto de casos de teste em exercitar o código do componente e a integração do componente na aplicação. Suponhamos um critério de teste estrutural ***C***, um componente ***k*** e um conjunto de teste $T_1 = \{t_1,t_2,t_3, t_4,t_5\}$, 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 de teste $T_2 = \{t'_1, t'_2\}$ utilizado para testar a aplicação ***A***. Isso implica que se T_1 está adequado, segundo o critério utilizado, para o código do componente, e T_2 tem as mesmas medidas de cobertura obtidas por T_1, então pode-se supor que T_2 também é bom para exercitar a integração do componente na aplicação.

Um resumo dessa idéia pode ser visto na Figura 4.1, na qual M_1 e M_2 são os métodos públicos do componente. No lado do desenvolvedor, um *driver* de teste k 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 exercitarem 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.

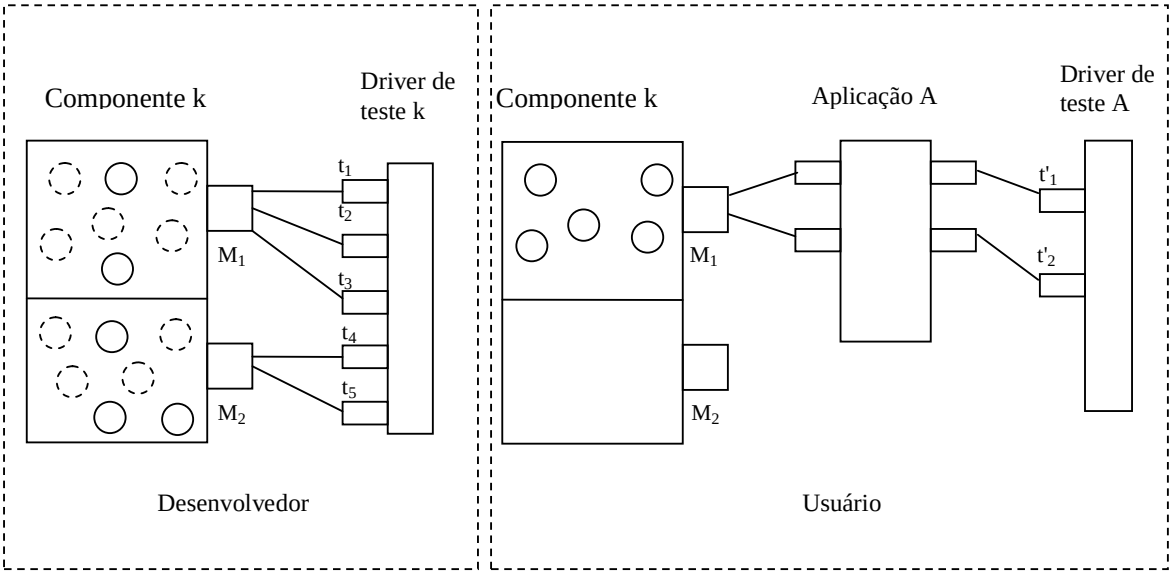


Figura 4.1 – Utilização dos dados de teste gerados pelo desenvolvedor do componente

Supondo-se, seguindo a Figura 4.1, que o desenvolvedor tenha criado cinco casos de teste para testar os dois métodos que fazem parte da interface pública do seu componente e que foram determinados para o critério C os requisitos de teste $\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9\}$ para o método M_1 e os requisitos $\{r_{10}, r_{11}, r_{12.}\ r_{13.}\ r_{14.}\ r_{15}\}$ para o método M_2. Supondo que o caso de teste t_1 cobre $\{r_1, r_2, r_3\}$, t_2 cobre $\{r_4, r_5, r_6\}$, t_3 cobre $\{r_7, r_8\}$, t_4 cobre $\{r_{10}, r_{11}, r_{12}\}$ e t_5 cobre $\{r_{11}, r_{12}, r_{13}\}$, uma forma de gerar os metadados de teste para que o usuário do componente possa utilizá-los seria a descrita na Tabela 4.1.

Tabela 4.1 – Proposta de metadados de teste

Método	Requisitos cobertos
M_1	$\{r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8\}$
M_2	$\{r_{10}, r_{11}, r_{12}, r_{13}\}$

Esses dados são fornecidos para o usuário do componente que pode verificar qual é a cobertura obtida, de acordo com esses requisitos, sobre o código do componente. Por exemplo, suponha-se que sejam definidos casos de teste t'_1 e t'_2 para testar métodos da aplicação que, por sua vez fazem chamadas a M_1, obtendo-se a cobertura descrita na Tabela 4.2.

Tabela 4.2 – Exemplo de cobertura obtida pelo usuário

Casos de teste	Método	Requisitos cobertos
t'_1	M_1	$\{r_1, r_2, r_3, r_4, r_5, r_6\}$
t'_2	M_1	$\{r_5, r_6\}$

Com essa informação sobre a execução da aplicação, pode-se perceber que nem todos os requisitos, que poderiam, foram executados. Isso dá uma idéia de certa fragilidade nos casos de teste do usuário, mas pode, também, indicar apenas que existem certas funcionalidades do componente que não estão sendo exercitadas devido às próprias características da aplicação, o que é bastante comum.

Assim, decidiu-se estender a descrição dos metadados de teste gerados pelo desenvolvedor, permitindo que o mesmo adicione uma descrição de cada um dos seus casos de teste. É o que se apresenta na Tabela 4.3.

Tabela 4.3 – Metadados incluindo a descrição dos casos de teste

Casos de teste	Método	Requisitos cobertos	Descrição
t_1	M_1	$\{r_1, r_2, r_3\}$	Este caso de teste ...
t_2	M_1	$\{r_4, r_5, r_6\}$	Este caso de teste ...
t_3	M_1	$\{r_7, r_8\}$	Este caso de teste ...
t_4	M_2	$\{r_{10}, r_{11}, r_{12}\}$	Este caso de teste ...
t_5	M_2	$\{r_{11}, r_{12}, r_{13}\}$	Este caso de teste ...

Com as informações da Tabela 4.3 o usuário pode perceber que seus casos de teste não cobriram o requisito r_8 e, além disso, pode obter uma descrição de quais seriam as características dos casos de teste do desenvolvedor que cobriram tal requisito. Com isso, poderia decidir se, no contexto de sua aplicação, tal requisito é ou não executável. A descrição fornecida pelo desenvolvedor pode ser apenas uma descrição textual, informal, como sugerida na Tabela 4.3 ou pode ser mais formal, utilizando-se, por exemplo, pré-condições que determinem os valores dos argumentos a serem fornecidos ao método invocado.

Com todo esse protocolo exigido entre o desenvolvedor e usuário para que a estratégia de teste possa ser implementada, é certo que o apoio automatizado de uma

ferramenta é essencial. Por isso desenvolveu-se uma ferramenta de suporte que será descrita na Seção 4.3.

Também deve-se levar em consideração que nem sempre o código fonte do componente está disponível para o usuário no momento da realização dos testes, portanto, propõe-se utilizar a ferramenta JaBUTi para a realização dos testes estruturais, visto que a ferramenta não necessita do código fonte para operar, ou poderia ser qualquer outra ferramenta com essas mesmas características. Na próxima seção serão abordadas algumas características da ferramenta JaBUTi.

4.2 Ferramenta JaBUTi (*Java Bytecode Understanding and Testing*)

A ferramenta JaBUTi (VINCENZI *et al*., 2006) foi desenvolvida para apoiar testes em programas, que foram escritos na linguagem Java, utilizando a técnica de teste estrutural. É possível executar teste, com base nos critérios de fluxo de controle e fluxo de dados, divididos em dois grupos: um em que os requisitos de teste tratam as exceções e, conseqüentemente, dependem da execução das exceções e outro que não requer a execução das exceções. Os critérios de teste implementado na JABUTi, são: Todos-Nós, Todos-Arcos, Todos-Usos e Todos-Potenciais-Usos, conforme apresentados na Tabela 4.4.

Tabela 4.4 – Critérios disponibilizados pela Ferramenta JaBUTI

Nome do Critério	Explicação
All-Nodes-ei	Requer a cobertura de cada nó do grafo do programa sem que ocorra uma exceção.
All-Edges-ei	Requer a cobertura de cada arco do grafo de programa sem que ocorra uma exceção.
All-Uses-ei	Requer a cobertura de cada par def-uso sem que ocorra uma exceção.
All-Pot-Uses-ei	Requer a cobertura de cada par def-potencial-uso sem que ocorra uma exceção
Os critérios *All-Nodes-ed*, *All-Edges-ed*, *All-Uses-ed* e *All-Pot-Uses-ed* são parecidos com os requisitos acima, mas requerem a execução das estruturas que somente possam ser alcançadas pela ocorrência de uma exceção.	

Para a execução dos testes, utilizando a ferramenta, o código fonte não precisa estar disponível, visto que a ferramenta utiliza o *bytecode*. Na Figura 4.2 mostra-se a apresentação do *bytecode* na ferramenta, são utilizadas cores para representar partes do código que foram ou não exercitados pelos casos de teste para o critério Todos-Nós independentes de exceção (*All-Nodes-ei*) e desta forma priorizar os requisitos não executados.

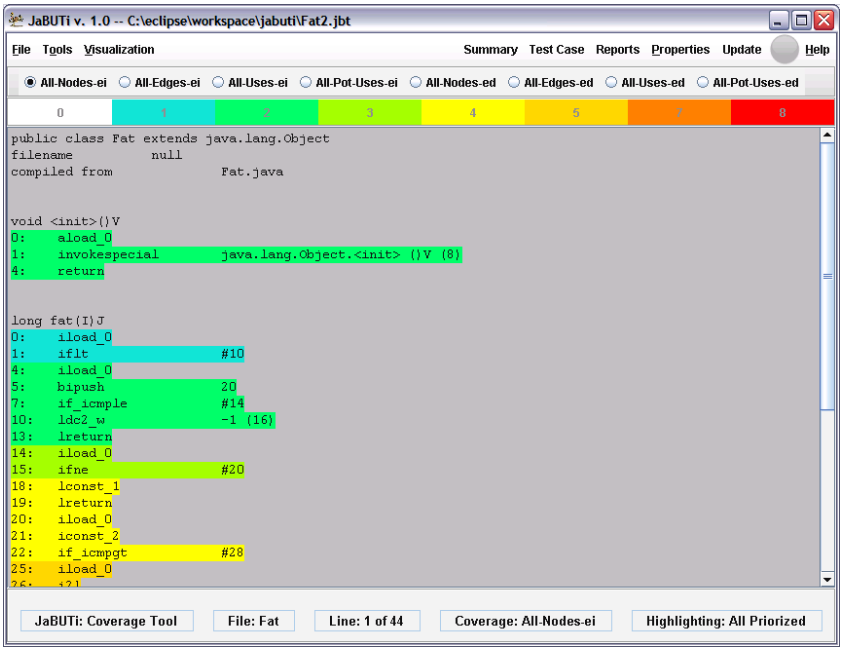


Figura 4.2 – Apresentação do *bytecode* na ferramenta JaBUTi

Na execução do programa em teste utiliza-se a instrumentação do código do programa para que possa coletar informações sobre os pontos que foram alcançados pela execução dos casos de teste. O instrumentador de teste é responsável por introduzir diretamente no *bytecode* instruções que registram a execução do trecho do programa.

A execução dos casos de teste ocorre em um processo separado da JaBUTi e comunica-se com ela por meio de um arquivo de rastro (*trace*). A execução de um caso de teste pode ser feita, por exemplo, instrumentando-se o programa em teste e executando-se um conjunto de teste no formato da ferramenta JUnit.

A JaBUTi apresenta várias informações sobre a condução de uma sessão de teste, como a cobertura obtida para cada um dos critérios estruturais implementados. Essa informação é fornecida utilizando-se uma representação textual do programa em teste ou na

forma de um grafo definição-uso. A ferramenta armazena toda a informação de cobertura, para cada caso de teste, num arquivo que caracteriza a sessão de teste, no formato XML. Isso permite que outras ferramentas tenham fácil acesso aos dados produzidos por ela.

4.3 FATEsC – Ferramenta de Apoio ao Teste Estrutural de Componentes

Parte da proposta deste trabalho é desenvolver uma ferramenta, nomeada FATEsC (Ferramenta de Apoio ao Teste Estrutural de Componentes), para gerar metadados, empacotá-los com o componente, permitir a extração e apresentação dos metadados, bem como promover a comparação entre as medidas de cobertura obtidas pelo usuário do componente com as medidas de cobertura do desenvolvedor do componente, conforme descrito na seção 4.1. As medidas de cobertura são fornecidas pela ferramenta JaBUTi, conforme mostrado na Figura 4.3.

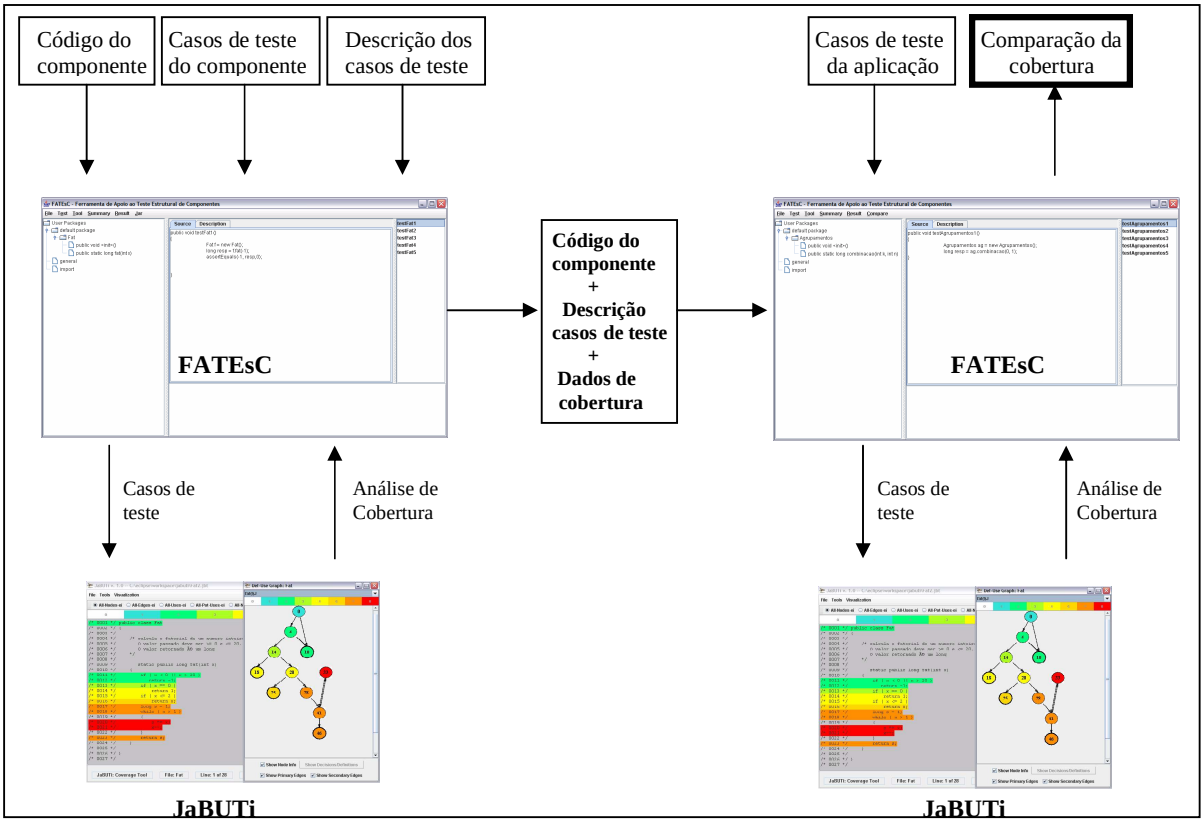


Figura 4.3 – Representação da proposta da ferramenta FATEsC

A FATEsC, segundo as duas perspectivas, a do desenvolvedor e usuário do componente, permite executar algumas funcionalidades e análises.

Na perspectiva do desenvolvedor do componente é possível:

a. criar, manipular e gerenciar casos de teste para cada método público do código;
b. desenvolver os casos de teste como um método Java, no formato JUnit (framework para desenvolvimento de teste de unidade em Java) (MASSOL e HUSTED, 2005);
c. definir para cada caso de teste as suas características, que poderão auxiliar o usuário do componente a obter a cobertura apontada. A princípio propõe-se uma descrição textual informal e;
d. empacotar na forma de metadados com o código do componente em um arquivo de extensão .JAR., as informações descritas acima (itens a, b e c) e as medidas de cobertura obtidas com a execução dos casos de teste.

Na perspectiva do usuário do componente é possível:

a. criar novos casos de teste e avaliar a adequação dos casos de teste e;
b. comparar medidas de cobertura obtidas, com as fornecidas no metadado, por exemplo, se na comparação obteve-se o mesmo resultado isso pode significar que realmente os casos de teste estão adequados. Caso contrário pode-se analisar os casos de testes descritos nos metadados e verificar se há a necessidade de criar outros ou se para a aplicação em questão, não é possível a cobertura.

A FATEsC é uma ferramenta *DeskTop,* escrita na linguagem de programação Java. No diagrama de casos de uso, Figura 4.4, são mostradas as interações dos atores usuário e desenvolvedor do componente com as funcionalidades que a ferramenta fornece. Percebe-se que existem atividades em comum para os dois atores e atividades específicas para cada ator.

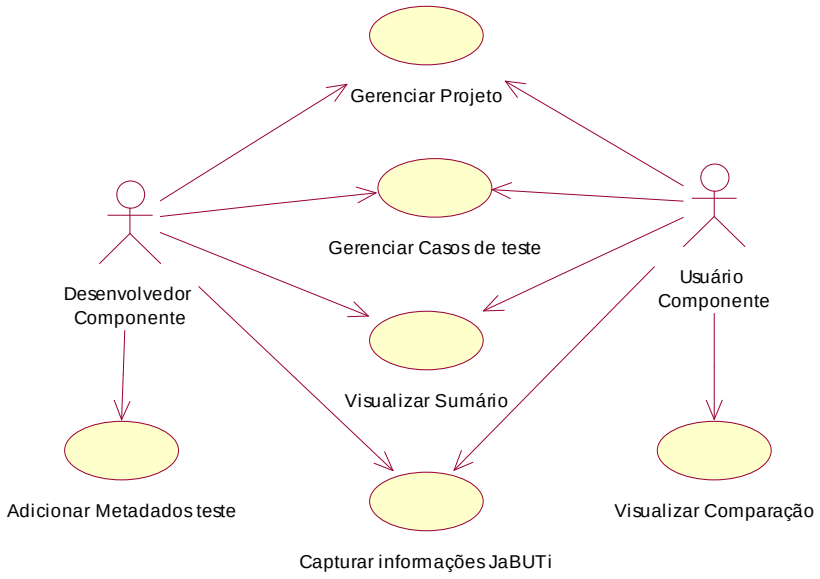


Figura 4.4 – Diagrama de caso de uso com as interações dos atores com as funcionalidades da FATEsC

Foram desenvolvidas duas visões: uma para ser utilizada pelo desenvolvedor do componente (FATEsC_D) e outra para ser utilizada pelo usuário do componente (FATEsC_U). Basicamente são compostas das mesmas funcionalidades, exceto pela comparação das coberturas obtidas pelos casos de teste do desenvolvedor e usuário do componente, que aparece apenas na visão do usuário e a associação dos metadados ao componente, que aparece apenas na visão do desenvolvedor do componente.

Na implementação da FATEsC reutiliza-se uma parte do código da ferramenta JaBUTi estabelecendo uma relação de dependência com os pacotes `br.jabuti.lookup`, `br.jabuti.project` e `br.jabuti.criteria`, conforme apresentam as Figuras 4.5 e 4.6.

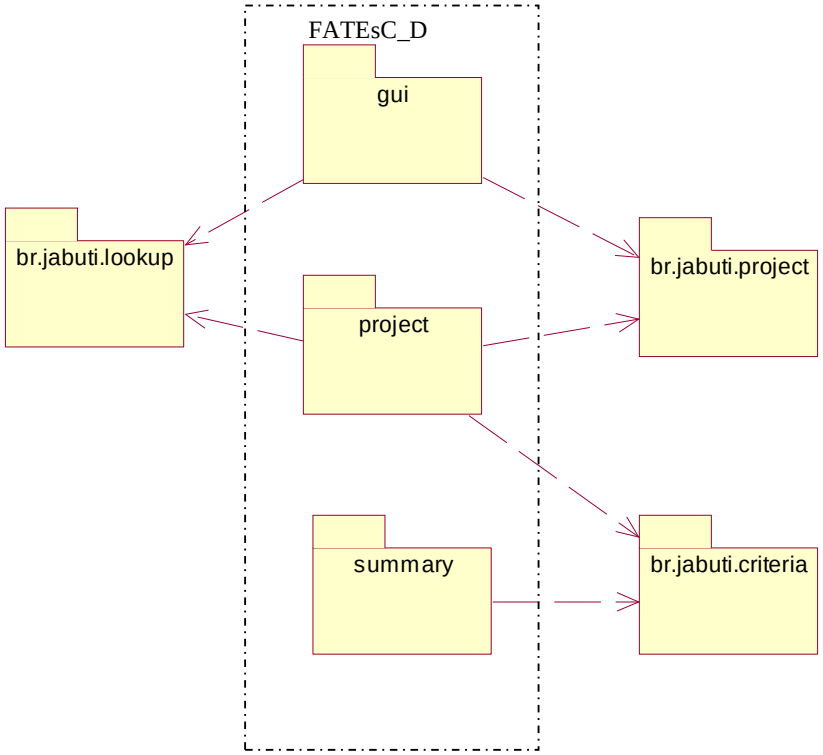


Figura 4.5 – Representação da relação de dependência entre os pacotes da ferramenta FATEsC_D em relação aos pacotes da ferramenta JaBUTi

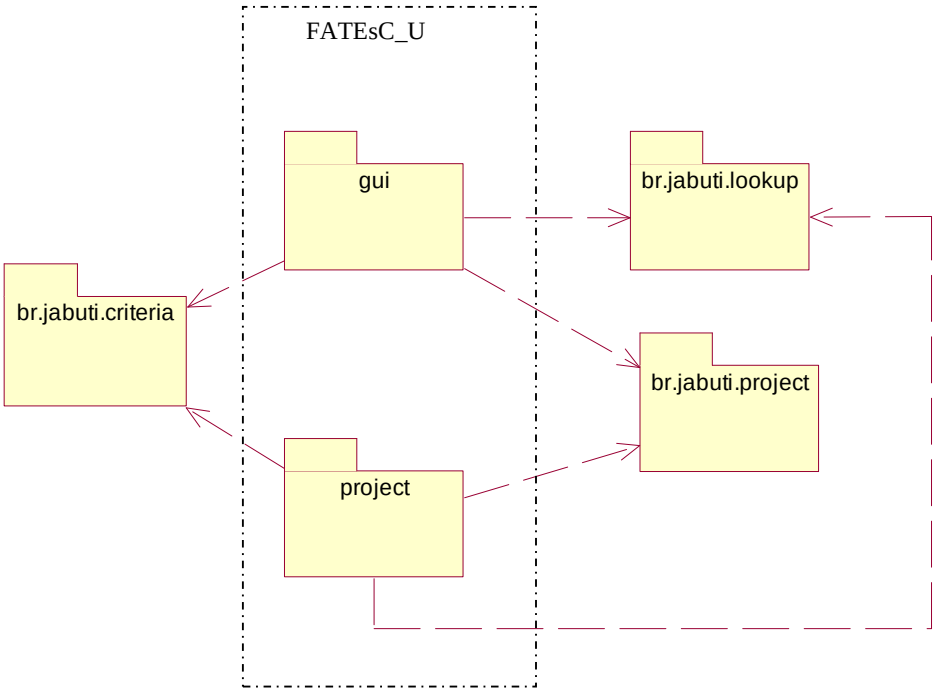


Figura 4.6 – Representação da relação de dependência entre os pacotes da ferramenta FATEsC_U em relação aos pacotes da ferramenta JaBUTi

A FATEsC_D, que é a visão utilizada pelo desenvolvedor do componente, é composta de quatro pacotes: ***gui***, ***test***, ***project*** e ***summary***. O pacote ***gui*** contém as classes responsáveis pela interface gráfica da ferramenta. O pacote ***test*** contém as classes responsáveis por funcionalidades ligadas aos casos de teste, como gerar e compilar a classe com os casos de teste. O pacote ***project*** contém as classes responsáveis por funcionalidades ligadas ao projeto, como salvar projeto e capturar das informações de cobertura na ferramenta JaBUTi. O pacote ***summary*** contém as classes responsáveis por apresentar o resumo das informações de cobertura capturadas após a execução dos casos de teste na ferramenta JaBUTi.

A FATEsC_U, que é a visão utilizada pelo usuário do componente, é composta de três pacotes: ***compare***, ***gui***, ***project***. O pacote ***compare*** contém as classes responsáveis pela apresentação e comparação das coberturas dos requisitos obtidas pelos casos de teste do desenvolvedor e usuário do componente. O pacote ***gui*** contém a classe responsável pela interface gráfica de parte do menu da ferramenta. O pacote ***project*** contém a classe responsável pela captura das informações de cobertura alcançadas pelos casos de teste da aplicação do usuário do componente na ferramenta JaBUTi.

A seguir serão apresentadas as funcionalidades, restrições, considerações e alguns detalhes relevantes da FATEsC, por meio de um exemplo simples de forma que facilite o entendimento dessas informações.

4.3.1 Funcionalidade na visão do desenvolvedor do componente

Para apresentar as funcionalidades da FATEsC_D propõe-se um projeto exemplo, bastante simples, considerando como componente a classe **`Fat`** que faz o cálculo do fatorial de um número, cujo código fonte pode ser visto na Figura 4.7.

Também para o exemplo, considera-se o critério Todos-Arcos que gera 12 requisitos de teste para o método **`fat`** numerados de 0 a 11, conforme mostrado no grafo na Figura 4.8, gerado pela ferramenta JaBUTi e modificado com a numeração. A seqüência dos números dos arcos está de acordo com a seqüência gerada pela ferramenta JaBUTi, no arquivo XML que será comentado adiante.

```
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 4.7 – Código do componente proposto para o exemplo

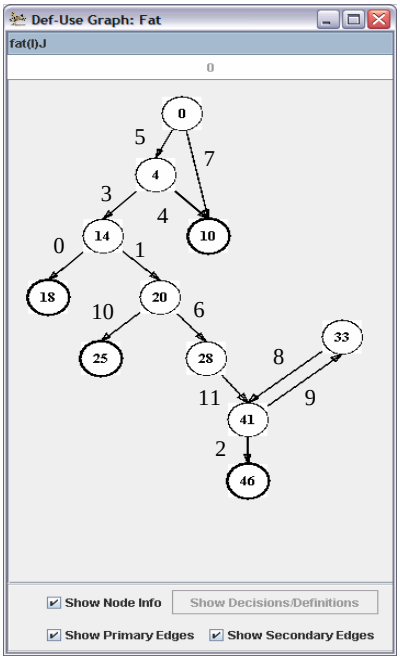


Figura 4.8 - Grafo do método `fat` do componente gerado pela ferramenta JaBUTi

Na Tabela 4.5 apresentam-se o conjunto de teste e as descrições dos casos de teste propostos para o componente.

Tabela 4.5 – Conjunto de teste e descrição informal dos testes propostos para o componente

Conjunto de teste	Descrição informal dos testes
testFat1	Foi passado o valor -1 como argumento para o método Fat, pois esse trata de modo particular valores de entrada negativos.
testFat2	Foi 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	Foi passado o valor 0 como argumento para o método Fat, pois esse trata de modo particular esse valor.
testFat4	Foi 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	Foi passado o valor 15 como argumento para o método Fat para efetuar o cálculo do seu fatorial.

Na Figura 4.9 tem-se uma visão dos menus disponíveis na ferramenta para as atividades do desenvolvedor do componente.

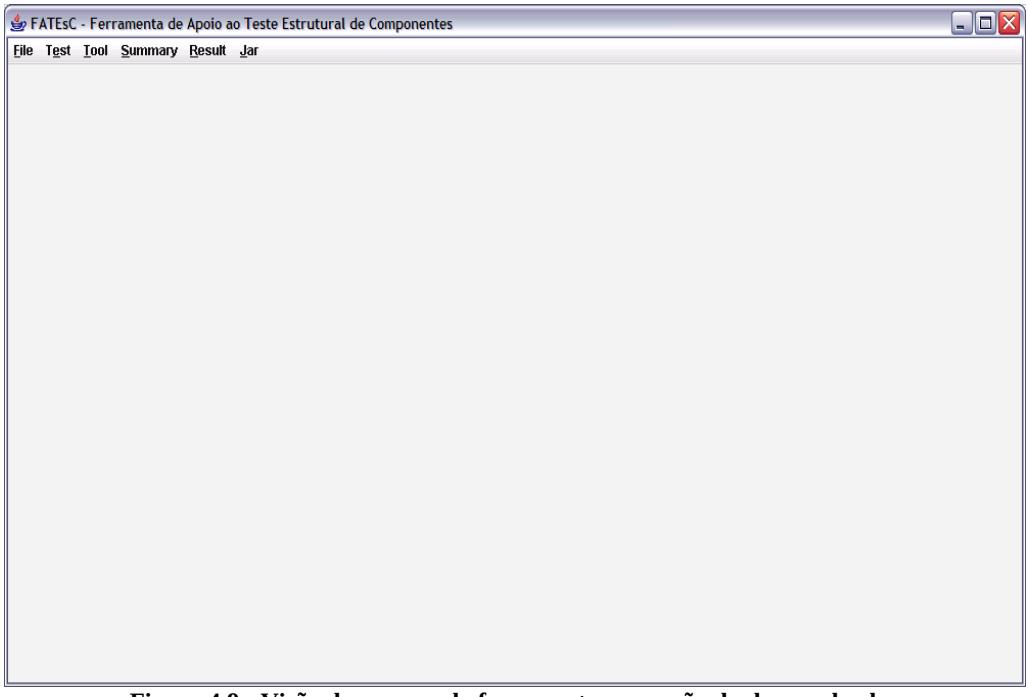


Figura 4.9 - Visão dos menus da ferramenta na versão do desenvolvedor

No menu ***File*** estão as funcionalidades para gerenciar o projeto, no menu ***Test*** estão a funcionalidade para gerenciar os casos de teste, no menu ***Tool*** estão as funcionalidades para gerenciar o agrupamento dos casos de teste em uma classe única, o menu ***Summary*** apresenta

um resumo das coberturas obtidas pelos testes realizados, no menu ***Result*** são importados os dados de cobertura obtidos após a execução dos casos de teste na Ferramenta JaBUTi e o menu **Jar** possibilita acrescentar os metadados ao componente no arquivo .jar.

Na Figura 4.10 tem-se uma visão dos submenus do menu ***File***, que agrupa funcionalidades referentes ao projeto, como: iniciar novo projeto (*Open Jar*), abrir um projeto existente (*Open Project*), fechar um projeto aberto (*Close Project*), salvar um projeto (*Save Project*) e sair da aplicação (*Exit*) . A seguir apresentam-se essas funcionalidades para o exemplo proposto.

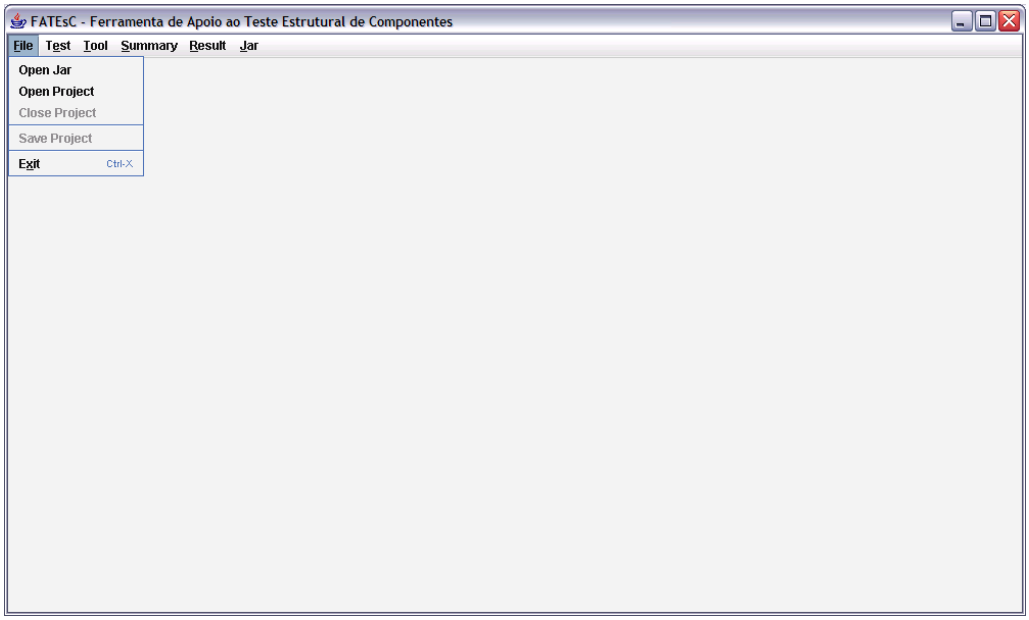


Figura 4.10 - Visão dos submenus do menu file.

O submenu ***Open Jar*** é utilizado para criar um projeto na FATEsC e, é necessário que o componente `Fat` esteja em um arquivo .jar, que é selecionado em uma caixa de diálogo. Após a seleção do arquivo **Fat.jar** é apresentada uma nova tela conforme mostra a Figura 4.11. Do lado esquerdo é apresentada uma estrutura em árvore contendo o nó raiz `User Packages` que tem como filhos: 1) os pacotes do componente; 2) o nó `general` e; 3) o nó `import`.

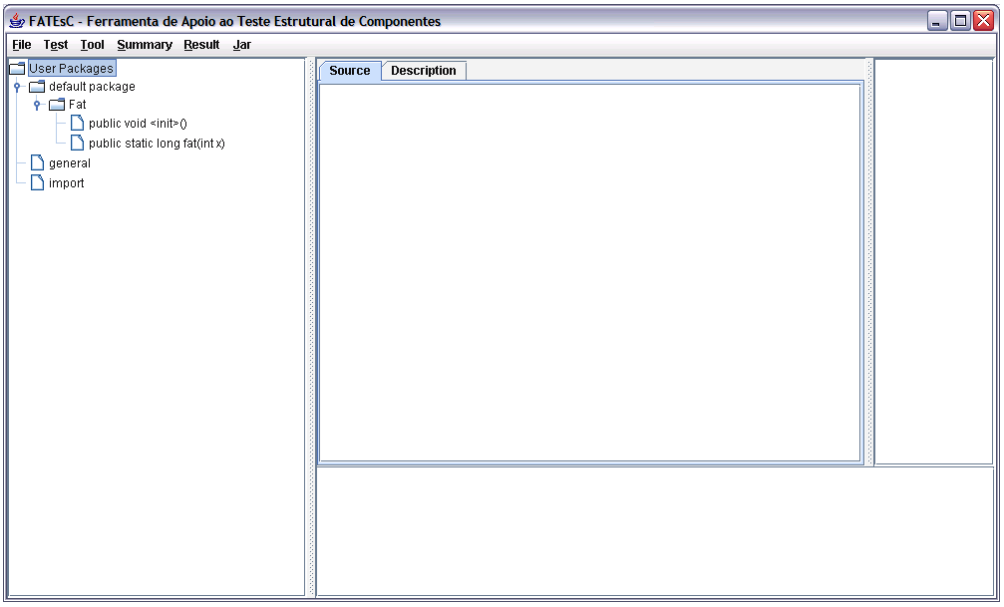


Figura 4.11 - Apresentação em forma de árvore do seu conteúdo do componente.

Os nós **`general`** e **`import`** são criados pela ferramenta e tem o conteúdo *default* necessários para a posterior criação da classe Java contendo os casos de teste. O conteúdo desses nós podem ser alterados conforme a necessidade do desenvolvedor.

No nó `general`, conforme mostra a Figura 4.12, há a estrutura inicial da classe ***FatTC***, cujo nome foi sugerido pela ferramenta com base no nome do arquivo **Fat.jar** selecionado, mas que poderá ser alterado a qualquer momento pelo desenvolvedor. Essa classe agrupa todos os métodos contendo os casos de teste, que serão utilizados para testar o componente. Os casos de teste estão no formato JUnit, que é um *framework* padrão para o desenvolvimento de testes de unidade em Java.

O conteúdo do nó `general` pode ser visto na parte central da tela na aba ***source***, após ter sido selecionado com o *mouse.* A classe ***FatTC*** herda as características da classe ***TestCase*** do JUnit e o método construtor da classe é uma exigência do JUnit para versões anteriores a 3.8, que é utilizada pela Ferramenta JaBUTi. O outro método, *tearDown()*, é necessário para a realização dos testes na Ferramenta JaBUTi.

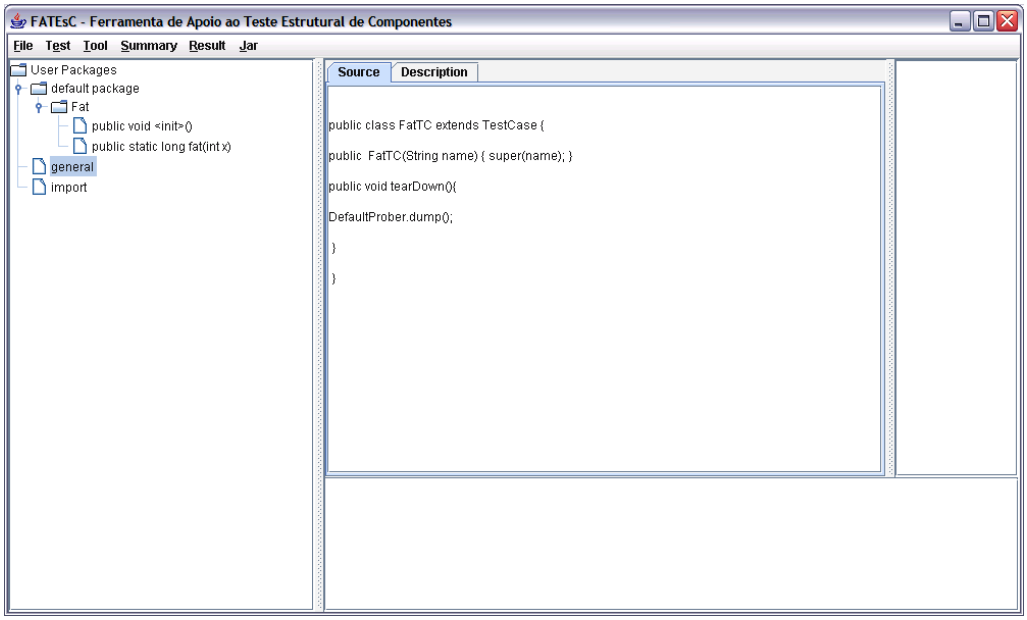


Figura 4.12 - Apresentação do código default do nó general na aba source.

O nó `import`, conforme mostra a Figura 4.13, contém as declarações *import* utilizadas pela classe ***FatTC*** e podem ser vistos na parte central da tela, na aba ***source***, após ter sido selecionado com o *mouse*.

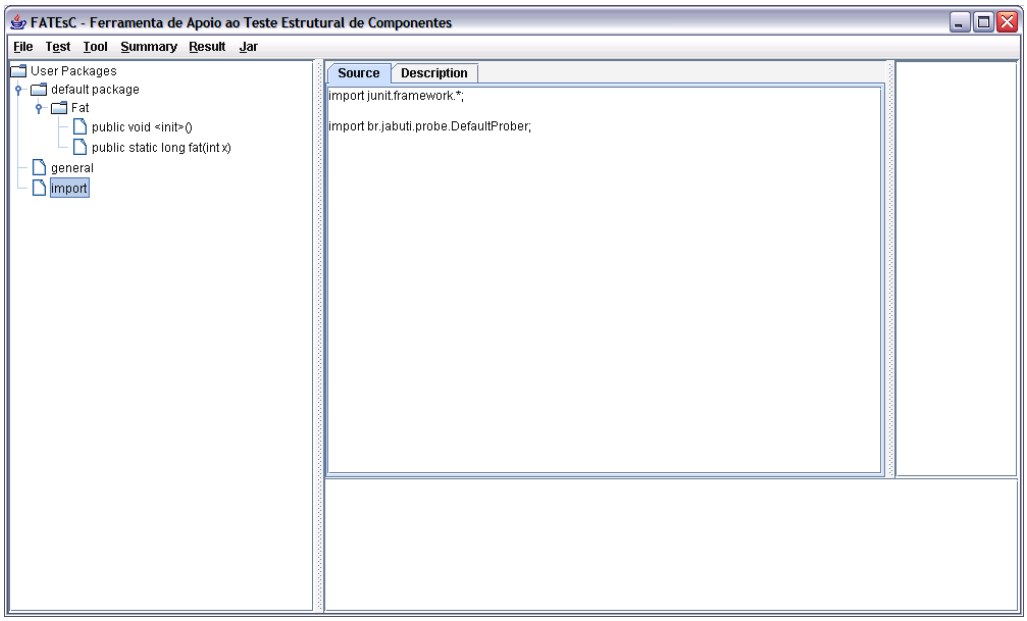


Figura 4.13 - Apresentação do código default do nó *import* na aba *source*.

O submenu ***SaveProject*** salva o projeto `Fat`. Se for a primeira vez que o projeto é salvo deve-se escolher em uma caixa de diálogo, o nome do projeto e o diretório onde será salvo, caso contrário, o diálogo será omitido. A ferramenta FATEsC gera um arquivo para armazenar os dados do projeto `Fat` que tem a extensão .exj e está no formato XML. Conforme mostra a Figura 4.14, foi armazenado o *path* do arquivo `Fat.jar`, o código dos nós `general` e `import` e o *path* do projeto (`Fat.exj`) em destaque. Os demais dados que são armazenados no arquivo `Fat.exj` são discutidos adiante.

O submenu ***Close Project*** fecha um projeto `Fat.exj`, perguntando ao usuário se o projeto deve ser salvo antes de ser fechado. Após ter fechado `Fat.exj`, a tela é limpa e retorna ao estado inicial, para que outra ação seja tomada, como por exemplo, abrir ou criar um outro projeto na ferramenta.

A função do submenu ***Open Project*** é abrir um projeto já existente. Para abrir o projeto `Fat.exj` basta escolhê-lo na caixa de diálogo e todos os dados salvos estão disponíveis para o desenvolvedor do componente. O submenu ***exit*** fecha a aplicação, solicitando a confirmação da ação. Se um projeto estiver carregado, o mesmo não será salvo antes da execução da ação.

A seguir serão criados os casos de teste para o componente `Fat`, apresentando as operações de criar, salvar e remover casos de teste disponíveis no menu ***Test***. Na Figura 4.15 tem-se uma visão dos submenus do menu ***Test***, que será detalhado a seguir.

Um caso de teste está associado a um único método, mas um método pode ter vários casos de teste associado a ele, portanto, para criar o ***testFat1*** e sua descrição é necessário que se selecione o método `fat` na estrutura do componente apresentado à esquerda e escolha-se a opção ***New*** no submenu do menu ***Test*** ou pressionando o botão direito do *mouse* no menu suspenso. Na caixa de entrada, apresentada na Figura 4.16, digita-se o nome do caso de teste (testFat1) que obrigatoriamente deverá iniciar com a palavra **test** por exigência do JUnit e esse nome é único, não podendo ser utilizado para outro caso de teste dentro do mesmo projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_06" class="java.beans.XMLDecoder">
 <object class="project.FatescProject">
  <void property="fileJar">
   <string>C:\eclipse\workspace\Fat\Fat.jar</string>
  </void>
  <void property="metCont">
   <object class="java.util.Hashtable">
    <void method="put">
     <string>general</string>
     <array class="java.lang.String" length="4">
      <void index="0">
       <string></string>
      </void>
      <void index="1">
       <string></string>
      </void>
      <void index="2">
       <string>

public class FatTC extends TestCase {

public  FatTC(String name) { super(name); }

public void tearDown(){

DefaultProber.dump();

 }

 }</string>
      </void>
      <void index="3">
       <string></string>
      </void>
     </array>
    </void>
    <void method="put">
     <string>import</string>
     <array class="java.lang.String" length="4">
      <void index="0">
       <string></string>
      </void>
      <void index="1">
       <string></string>
      </void>
      <void index="2">
       <string>import junit.framework.*;

import br.jabuti.probe.DefaultProber;</string>
      </void>
      <void index="3">
       <string></string>
      </void>
     </array>
    </void>
   </object>
  </void>
  <void property="projName">
   <string>C:\eclipse\workspace\Fat\Fat.exj</string>
  </void>
  </object>
</java>
```

Figura 4.14 – Arquivo XML do projeto `Fat.exj`

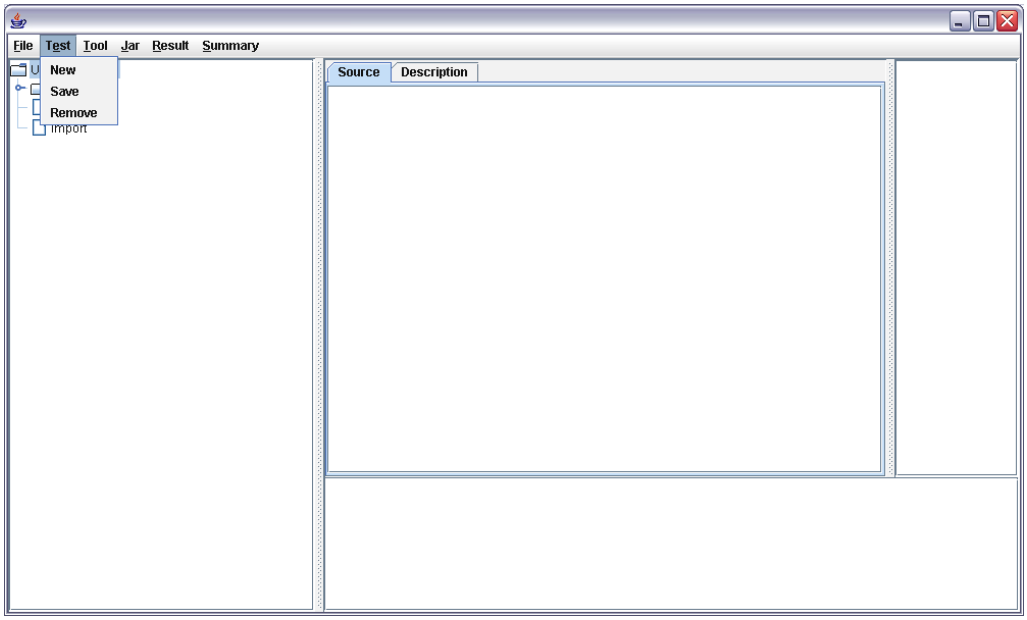


Figura 4.15 - Visão dos submenus do menu *Test.*

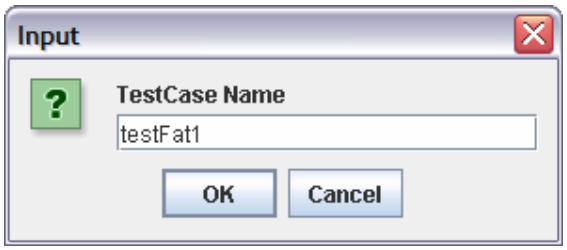


Figura 4.16 – Diálogo para criação do nome do caso de teste.

Ao selecionar com o *mouse* o botão **OK** na caixa de entrada o caso de teste é exibido em uma lista na parte mais à direita da tela. Para a codificação do caso de teste ***testFat1*** utiliza-se o espaço na aba ***Source*** e para fazer a sua descrição na aba ***Description***, conforme apresentam as Figuras 4.17 e 4.18. Esse mesmo processo deve ser repetido para a criação dos demais casos de teste (testFat2, testFat3, testFat4 e testFat5). As descrições dos testes devem conter informações que ajudem o usuário do componente a entender o objetivo do caso de teste e que dê indicações para serem reproduzidos para a aplicação, se necessário, já que as descrições de cada caso de teste serão anexadas ao componente como forma de metadado. As informações podem ser: argumentos utilizados, retorno obtido, exceções, etc.

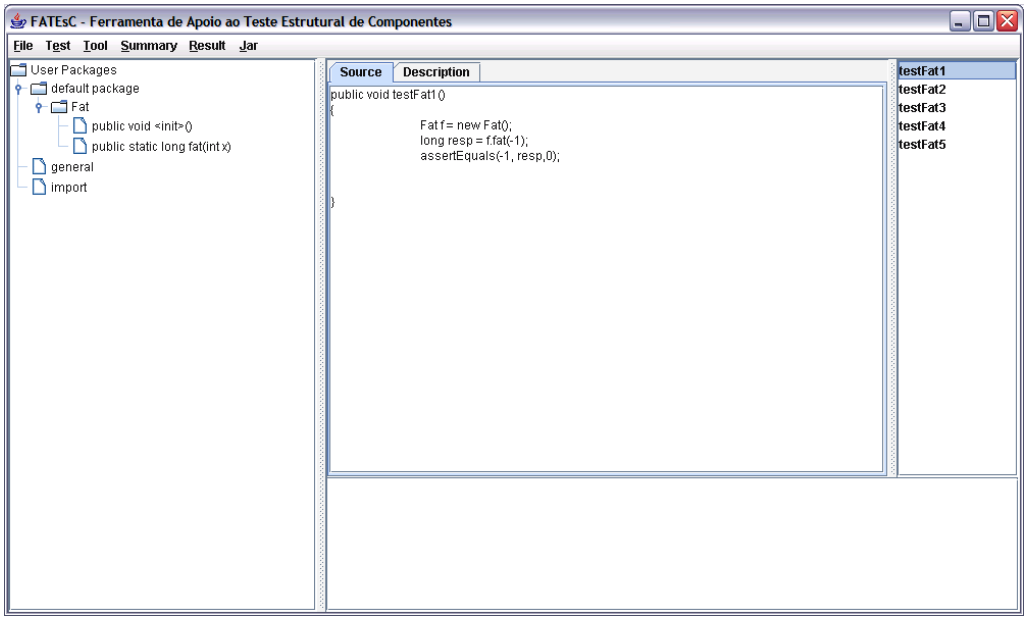


Figura 4.17 – Apresentação do caso de teste testFat1 na aba *Source*

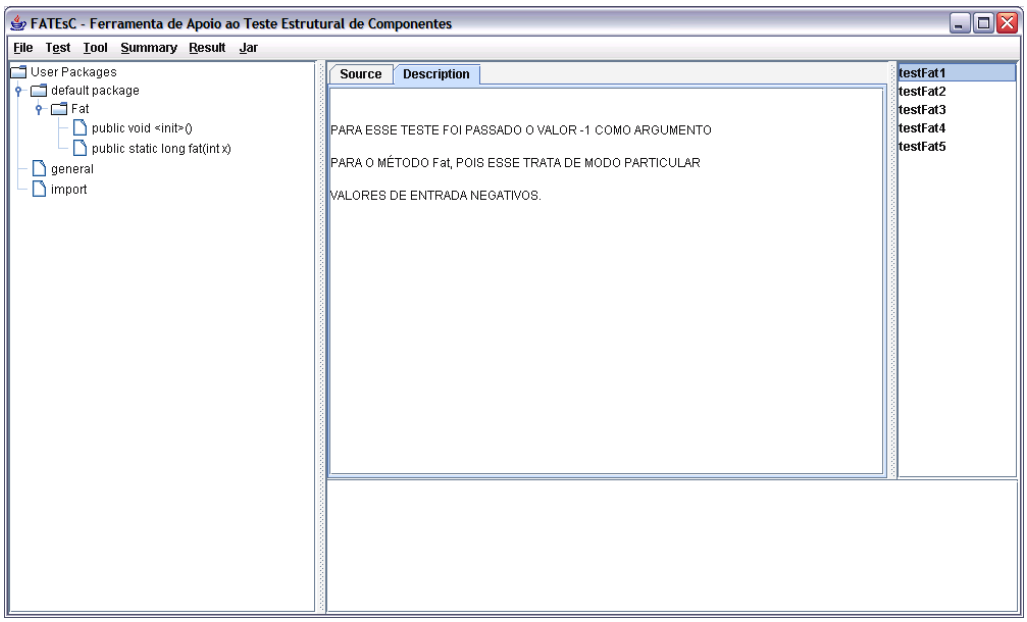


Figura 4.18 - Apresentação do descritivo para o caso de teste testFat1 na aba *Description*.

Para salvar o caso de teste ***testFat1*** utiliza-se o submenu ***Save*** ou o menu suspenso, e caso seja necessário fazer qualquer alteração nos casos de teste salvos, basta selecioná-lo na lista de casos de teste, a direita da tela, fazer as alterações e salvá-las novamente. Esse submenu ***Save*** ou o menu suspenso também é utilizado para salvar alterações feitas no código dos nós `general` e `import`.

Se por algum motivo tem-se a necessidade de remover um caso de teste salvo, seleciona-se o caso de teste na lista e utiliza-se o submenu ***Remove*** ou o menu suspenso**.** A fim de facilitar a identificação dos casos de teste associados a um determinado nó da estrutura do componente, basta selecionar o nó desejado e só serão listados os casos de teste a ele associado.

Observe que se o componente for modificado, por exemplo, um método ao qual existam casos de teste associados deixe de existir, a FATEsC apresentará uma mensagem informando que existem casos de teste sem vinculo e que deverão ser removidos.

Os dados referentes a um caso de teste que são salvos no projeto `Fat.exj` são: o nome do caso de teste, a assinatura do método ao qual ele está associado em duas formas, uma completa e outra resumida, o código do caso de teste e sua descrição, apresentados em destaque no trecho XML do projeto `Fat.exj` mostrados na Figura 4.19.

```
<void method="put">
     <string>testFat1</string>
     <array class="java.lang.String" length="4">
      <void index="0">
       <string>default package.Fat.public static long fat(int x)</string>
      </void>
      <void index="1">
       <string>Fat.fat(I)J</string>
      </void>
      <void index="2">
       <string>public void testFat1()
{
        Fat f = new Fat();
        long resp = f.fat(-1);
        assertEquals(-1, resp,0);


}</string>
      </void>
      <void index="3">
       <string>

PARA ESSE TESTE FOI PASSADO O VALOR -1 COMO ARGUMENTO

PARA O MÃ‰TODO Fat, POIS ESSE TRATA DE MODO PARTICULAR

VALORES DE ENTRADA NEGATIVOS.

</string>
      </void>
     </array>
    </void>
```

Figura 4.19 – Dados do testFat1 salvos no projeto `Fat.exj`.

A seguir, os casos de teste são agrupados em uma única classe, cuja assinatura foi criada no nó *general*, em um arquivo de extensão .java para o componente `Fat`, apresentando as operações de criar, salvar e remover casos de teste disponíveis no menu ***Test***. Na Figura 4.20 tem-se uma visão dos submenus do menu ***Tool***, que será detalhado a seguir

Após a criação dos casos de testes gera-se a classe com o conteúdo do nó import, general e todos os casos de testes. Ao selecionar o submenu Generate Class, será aberto um diálogo para informar o nome e lugar onde o arquivo .java será salvo, que para o exemplo é FatTC.java. O código da classe gerada pode ser visto ao centro, na aba ***source.***

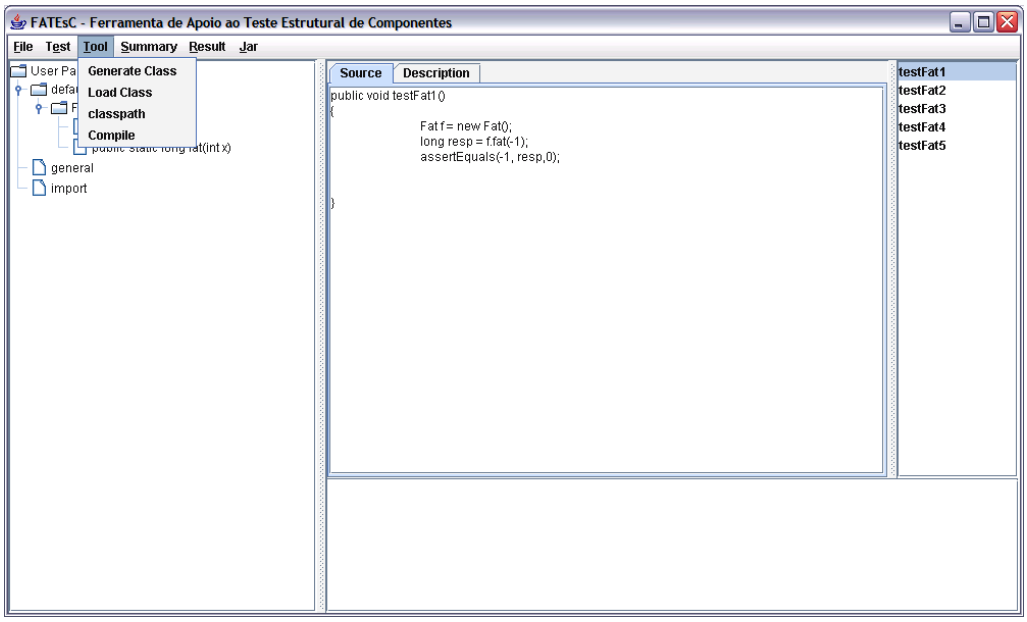


Figura 4.20 – Visão dos submenus do menu *Tool.*

Também a qualquer momento, pode-se ver o código `FatTC.java`, na aba *source*, utilizando-se o submenu ***Load Class***. Qualquer alteração necessária no código do arquivo `FatTC.java` não pode ser feita diretamente nesse arquivo, mas sim nas partes que o compõe, por exemplo, se for necessário a inclusão de um novo *import*, deve-se incluí-lo no nó `import` e gerar novamente o arquivo FatTC.java no submenu ***Generate Class***. Essa sistemática de alteração é uma forma de garantir que as partes (nó `import` e `general` e casos de teste) estejam de acordo com o todo (`FatTC.java`).

Na FATEsC utilizando-se o submenu ***classpath***, conforme mostrado na Figura 4.21, pode-se configurar a *classpath*, que é uma variável de ambiente que especifica onde estão os arquivos e bibliotecas necessários para fazer a compilação dos casos de teste. Acionando o botão ***find*** abre-se um diálogo para a seleção dos arquivos e bibliotecas que são necessários. O código do arquivo `FatTC.java` é compilado gerando seu *bytecode* (`FatTC.class`) utilizando-se o submenu ***Compile***. O resultado da compilação é apresentado na seção inferior da Ferramenta, conforme mostrado na Figura 4.22. A mensagem ***Process Completed*** indica

que a compilação foi realizada com sucesso. Para limpar a área de compilação deve-se utilizar o menu suspenso *clear*.

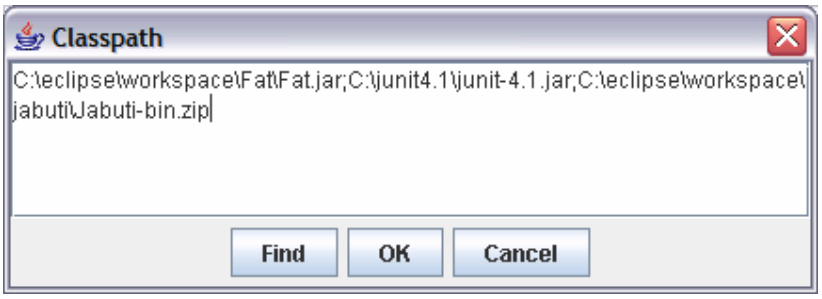


Figura 4.21 – Definição da *classpath* para compilar o arquivo `FatTC.java`

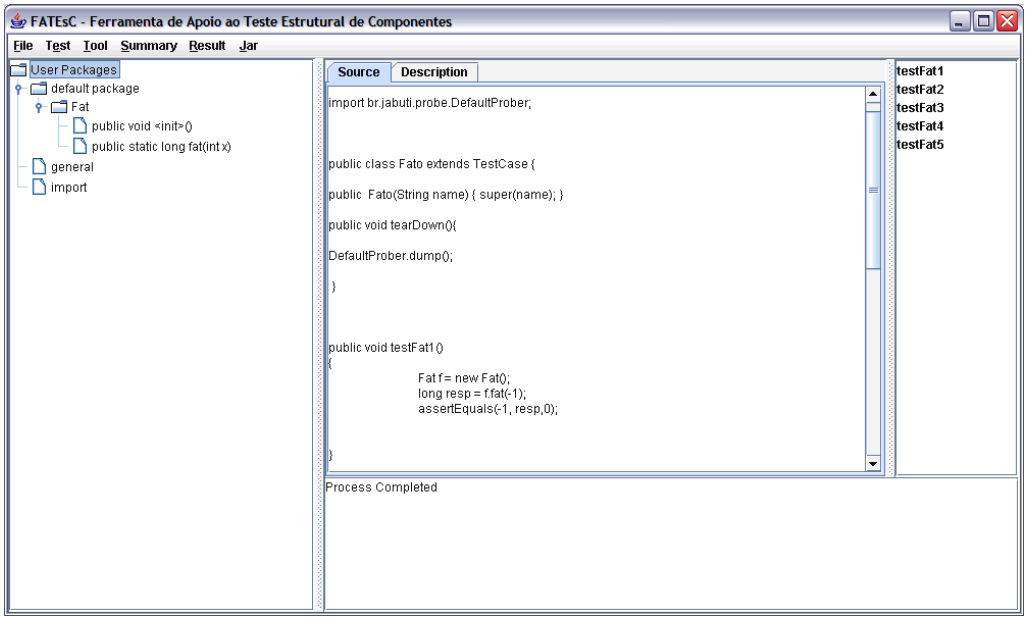


Figura 4.22 – Resultado da compilação do arquivo `FatTC.java`

Conforme comentado anteriormente, a ferramenta JaBUTi automatiza algumas fases de teste utilizando a técnica de teste estrutural e o próximo passo é executar os casos de teste utilizando o arquivo `FatTC.class` que foi criado na ferramenta FATEsC, a fim de obter os dados de cobertura dos testes para todos os critérios implementados por ela. Cria-se um projeto `Fat.jbt` para o componente `Fat` na ferramenta JaBUTi, conforme ilustrado na Figura 4.23, e utilizando o submenu ***Importing from JUnit*** do menu ***Test Case***, seleciona-se

`FatTC.class` na caixa de diálogo e os casos de teste são listados, conforme apresentado na Figura 4.24. Ao acionar o botão ***Import Selected*** executam-se os casos de teste selecionados.

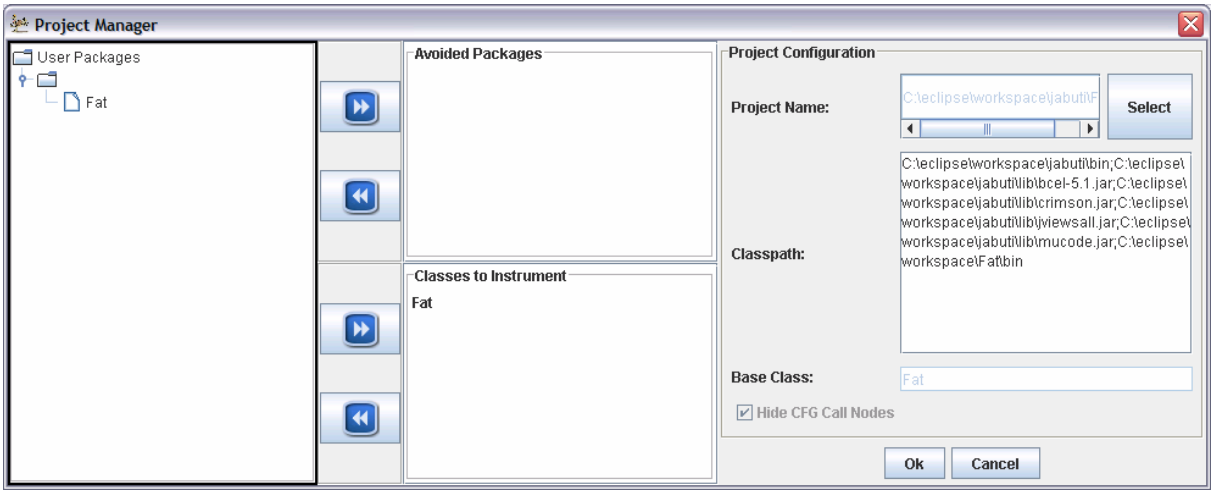


Figura 4.23 – Tela da ferramenta JaBUTi para a criação do projeto `FaT.jbt`

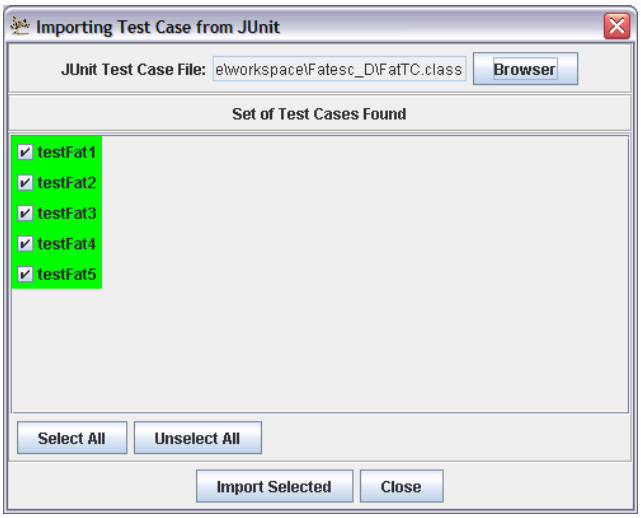


Figura 4.24 – Tela da ferramenta JaBUTi para a importação dos casos de teste para o componente

A ferramenta JaBUTi salva os dados referentes ao projeto Fat.jbt no formato XML, e dentre os dados estão a relação dos casos de teste e quais foram os requisitos cobertos por cada caso de teste, separados por métodos do componente e critérios implementados na ferramenta. Essas informações de cobertura dos requisitos de teste são capturadas pela ferramenta FATEsC.

Na Figura 4.25 tem-se uma visão do submenu ***Importing JaBUTi*** do menu ***Result***, que aciona a importação dos dados de cobertura obtidos com a execução dos casos de teste na

ferramenta JaBUTi. Uma caixa de diálogo é aberta, na primeira vez que essa ação é executada, para que se escolha o projeto que contém as informações que serão importadas. No caso escolhe-se o arquivo `Fat.jbt`.

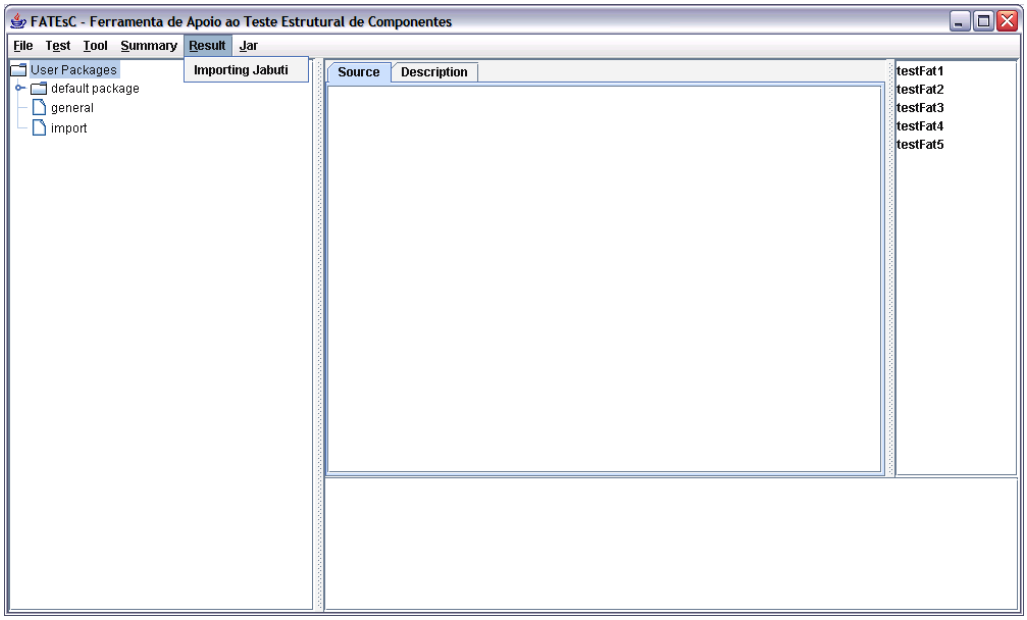


Figura 4.25 – Visão do submenu do menu Result.

Antes de iniciar a descrição da forma como são capturadas as informações, faz-se necessário a seguinte consideração: para a importação das informações de cobertura dos requisitos de teste, considera-se para cada caso de teste apenas os requisitos cobertos por ele nos métodos para o qual ele foi elaborado. Por exemplo, na Figura 4.26 tem-se um componente com dois métodos públicos (MC_1 e MC_2) e os casos de teste t_1 e t_2 foram elaborados para testar o método MC_1 e o caso de teste t_3 para testar o método MC_2. Na execução do caso de teste t_2, tem-se a chamada do método MC_2 (seta pontilhada), nesse caso considera-se para t_2 apenas as coberturas obtidas por ele para o método MC_1, desconsiderando-se a cobertura obtida para o método MC_2. Essa estratégia foi adotada para diminuir a complexidade no momento de comparar as coberturas obtidas pelos casos de teste do desenvolvedor e usuário do componente. A comparação das coberturas será discutida mais adiante.

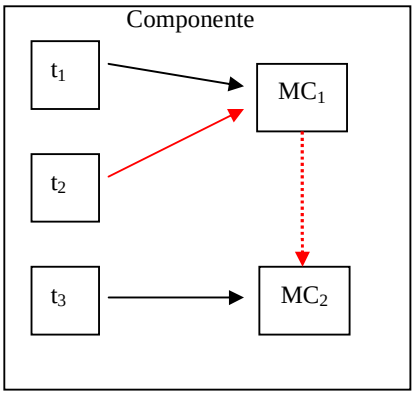


Figura 4.26 - Representação do critério utilizado para a captura das informações de cobertura do componente

A partir da importação dos dados de cobertura pode-se chegar ao resumo apresentado na Tabela 4.6, para cada caso de teste quais foram os requisitos de teste cobertos para o método `fat`.

Tabela 4.6 – Requisitos cobertos por cada caso de teste para o método `fat`

Nome dos testes	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

A Figura 4.27 mostra um trecho do XML do projeto `Fat.exj`, onde se observa a forma que a FATEsC salva as informações de cobertura obtidas pelos casos de teste. Na *tag* <string> tem-se o caso de teste (`testFat5`), o método para o qual o caso de teste foi elaborado (`Fat.fat(I)J`) e a qual critério pertence esse resultado (2). Adota-se 0 para o critério Todos-Nós independentes de exceção (*All-Nodes-ei*), 1 para o critério Todos-Nós dependentes de exceção (*All-Nodes-ed*), 2 para o critério Todos-Arcos independentes de exceção (*All-Edges-ei*), 3 para o critério Todos-Arcos dependentes de exceção (*All-Edges-ed*), 4 para o critério Todos-Usos independentes de exceção (*All-Uses-ei*), 5 para o critério Todos-Usos dependentes de exceção (*All-Uses-ed*), 6 para o critério Todos-Potenciais-Usos independentes de exceção (*All-Pot-Uses-ei*) e 7 para o critério Todos-Potenciais-Usos dependentes de exceção (*All-Pot-Uses-ed*).

```
<void method="put">
  <string>testFat5/Fat.fat(I)J/2</string>
  <array class="java.lang.String" length="14">
    <void index="0">
      <string>F</string>
    </void>
    <void index="1">
      <string>T</string>
   </void>
   <void index="2">
     <string>T</string>
  </void>
  <void index="3">
     <string>T</string>
  </void>
  <void index="4">
     <string>F</string>
  </void>
  <void index="5">
     <string>T</string>
  </void>
  <void index="6">
    <string>T</string>
  </void>
  <void index="7">
    <string>F</string>
  </void>
  <void index="8">
    <string>T</string>
  </void>
  <void index="9">
    <string>T</string>
  </void>
  <void index="10">
    <string>F</string>
  </void>
  <void index="11">
    <string>T</string>
  </void>
 </array>
</void>
```

Figura 4.27 – Parte do XML gerado para o projeto `Fat.exj`

Também na Figura 4.27 pode-se ver os requisitos de teste nas *tags void index* de 0 a 11 com o conteúdo T (*True*) quando o requisito está coberto pelo caso de teste (`testFat5`) ou F (*False*) para os requisitos que não estão cobertos. Essas informações se repetem no arquivo XML, do projeto `Fat.exj`, para todos os métodos em relação aos casos de teste e em relação aos critérios citados acima.

Na Figura 4.28 tem-se uma visão do submenu ***by method*** do menu ***Summary***, que apresenta os resultados de cobertura dos requisitos pelos casos de teste.

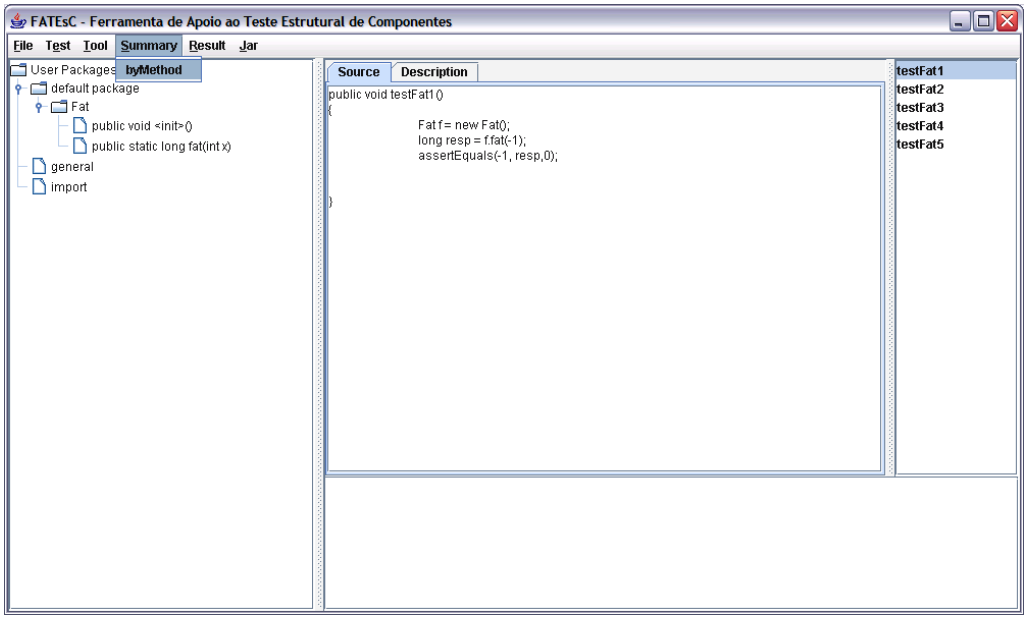
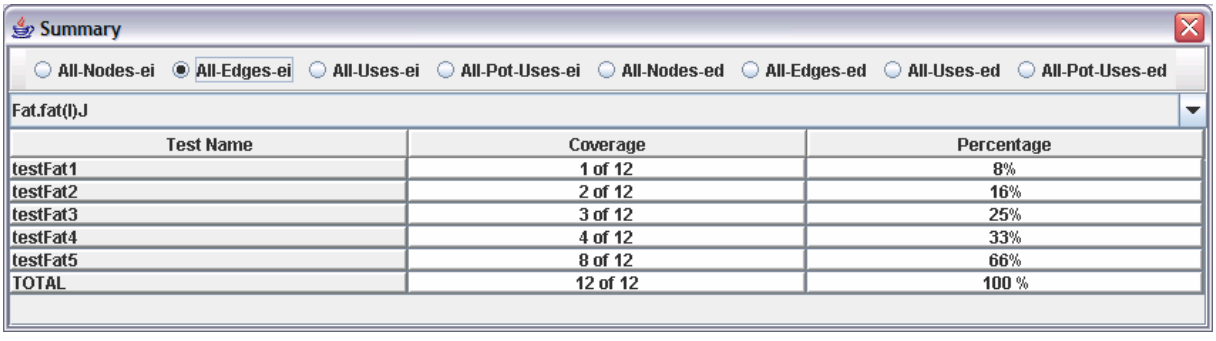


Figura 4.28 – Visão do submenu *Summary*.

Na Figura 4.29 é apresentada em forma de tabela, para o critério Todos-Arcos independentes de exceção (*All-Edges-ei*) e o método `fat`, na primeira coluna os casos de teste, na segunda coluna a quantidade de requisitos cobertos pelos casos de teste em relação ao total de requisitos e na terceira coluna a porcentagem de cobertura. Por exemplo, o caso de teste testFat1 cobriu 1 de um total de 12 requisitos, portanto um percentual de cobertura de 8%. Na última linha da tabela é apresentado o total geral, ou seja, o total de requisitos cobertos pelo conjunto de teste em relação ao total de requisitos.



Test Name	Coverage	Percentage
testFat1	1 of 12	8%
testFat2	2 of 12	16%
testFat3	3 of 12	25%
testFat4	4 of 12	33%
testFat5	8 of 12	66%
TOTAL	12 of 12	100 %

Figura 4.29 – Apresentação do resultado dos testes por método do componente.

Na Figura 4.30 tem-se uma visão do submenu ***Add Metadado*** do menu ***Jar***. Essa funcionalidade permite acrescentar, ao componente, as informações de cobertura obtidas pelo

conjunto de teste elaborado pelo desenvolvedor e as descrições para cada caso de teste. Por exemplo, pode-se acrescentar o arquivo `Fat.exj` (projeto FATEsC), que contém as informações que ficarão disponíveis para o usuário do componente, ao `Fat.jar` (componente) e para isso, será aberta uma caixa de diálogo para que se escolha o arquivo que será acrescentado.

Atualmente, todas as informações contidas no arquivo de projeto são adicionadas ao componente. Pode-se aprimorar essa funcionalidade adicionando-se apenas as informações relevantes, ou permitindo que o desenvolvedor selecione os dados a serem disponibilizados.

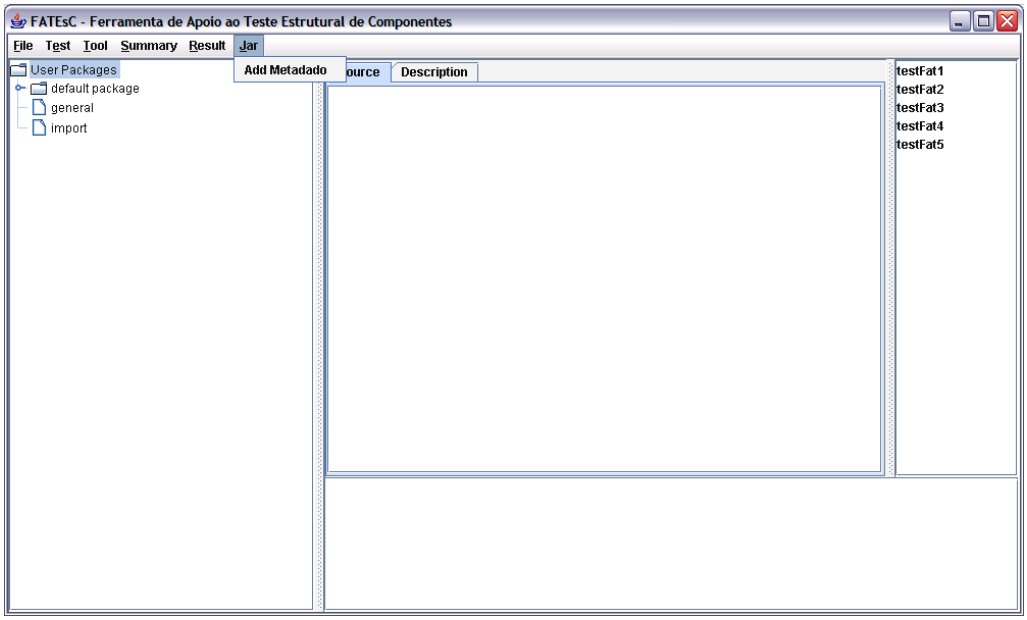


Figura 4.30 - Visão do submenu do menu Jar.

Na próxima seção são descritas as funcionalidades disponíveis para o usuário do componente. Basicamente, todas as funcionalidades descritas nessa seção estão disponíveis para o usuário do componente, exceto a funcionalidade de acrescentar informações ao componente. Portanto essas funcionalidades em comum não serão abordadas tão detalhadamente a seguir.

4.3.2 Funcionalidade na visão do usuário do componente

Para demonstrar as funcionalidades da FATEsC_U propõe-se um projeto exemplo, bastante simples, considerando como uma aplicação a classe `Agrupamentos` que faz o cálculo de combinação simples utilizando a classe **`Fat`** como componente. O código fonte da aplicação pode ser visto na Figura 4.31. Na Tabela 4.7 apresenta-se o conjunto de teste e dados de teste propostos para a aplicação.

Segue-se com o exemplo proposto, criando-se o projeto `Agrupamentos.exj`, a partir do arquivo `Agrupamentos.jar`, e os casos de teste para a aplicação, conforme mostrado na Figura 4.32.

```
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 4.31 – Código da aplicação proposta para o exemplo

Tabela 4.7 – Conjunto de teste e dados de teste propostos para a aplicação

Conjunto de teste	Dados de teste
testAgrupamentos1	k = 0, n = 1
testAgrupamentos2	k = 1, n = 0
testAgrupamentos3	k = 5, n = 7
testAgrupamentos4	k = 10, n = 3

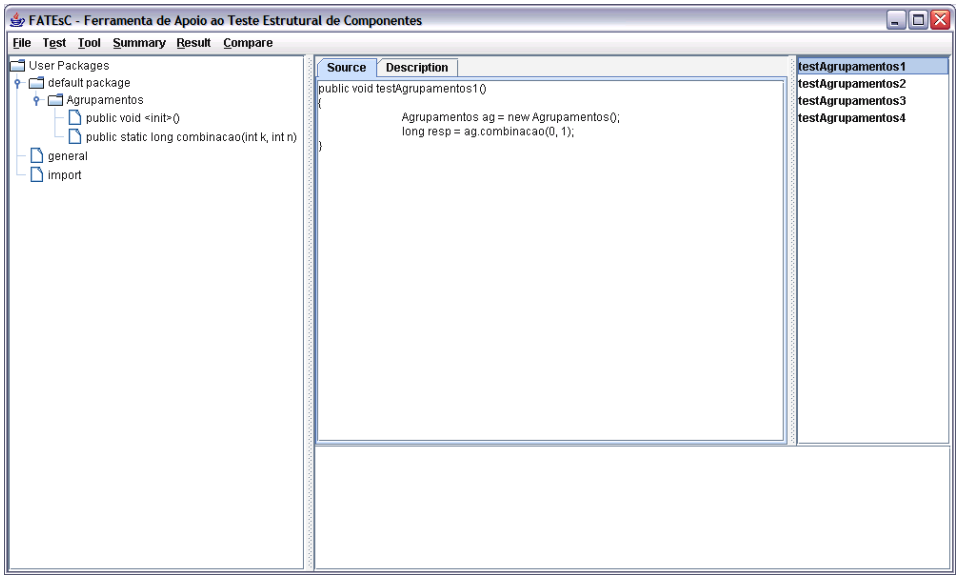


Figura 4.32 – Tela da FATEsC mostrando casos de teste gerados para a aplicação `Agrupamentos.exj`

Também é gerada a classe `AgrupamentosTC.class`, que contém o conjunto de teste elaborado para testar a aplicação, no formato JUnit, e que é utilizada na execução dos testes na ferramenta JaBUTi.

Para o projeto `Agrupamentos.jbt`, na ferramenta JaBUTi, foram instrumentadas as classes `Agrupamentos` (aplicação) e `Fat` (componente), conforme mostrado na Figura 4.33. Desta forma obtêm-se as informações de cobertura da aplicação e do componente. O próximo passo é importar os casos de teste para a execução, conforme mostrado na Figura 4.34.

Após a execução dos testes na ferramenta JaBUTi, deve-se capturar as informações de cobertura obtidas pelos casos de teste utilizando o submenu ***Importing JaBUTi*** do menu ***Result*** na FATEsC. O processo é o mesmo descrito anteriormente para o componente, uma caixa de diálogo é aberta, na primeira vez que essa ação é executada, para que se escolha o projeto que contém as informações que serão importadas. No caso escolhe-se o arquivo `Agrupamentos.jbt`.

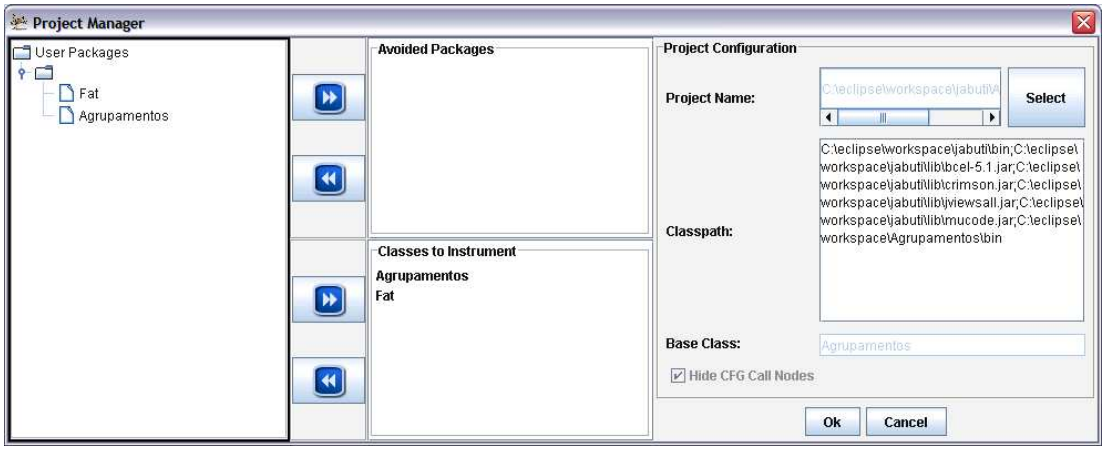


Figura 4.33 - Tela para a criação do projeto Agrupamentos.jbt

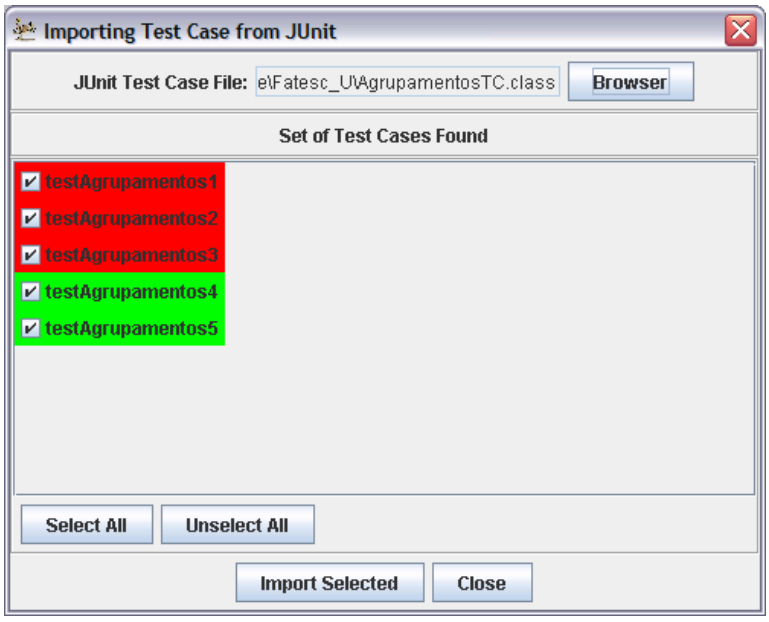


Figura 4.34 – Tela para importar os casos de teste da aplicação na Ferramenta JaBUTi

A Figura 4.35 apresenta um trecho de XML referente ao projeto `Agrupamentos.jbt`, criado na ferramenta JaBUTi. Na Tabela 4.8 é apresentado um resumo das coberturas obtidas pelos casos de teste, para o método `fat` e critério Todos-Arcos independentes de exceção (*All-Edges-ei*).

```
<INST_CLASS  name="Fat" size="430" checksum="46-0">
 <EXTEND name="java.lang.Object" level="1"/>
 <SOURCE name=""/>
 <METHOD id="1" name="fat(I)J">
  <All-Edges-ei id="2" totreq="12" act="12" inf="0">
    <EDGE id="0" from="14" to="18" active="Y" covered="N" infeasible="N" effectivetcs=""/>
    <EDGE id="1" from="14" to="20" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
    <EDGE id="2" from="41" to="46" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
    <EDGE id="3" from="4" to="14" active="Y" covered="Y" infeasible="N" effectivetcs="0007 "/>
    <EDGE id="4" from="4" to="10" active="Y" covered="N" infeasible="N" effectivetcs=""/>
    <EDGE id="5" from="0" to="4" active="Y" covered="Y" infeasible="N" effectivetcs="0007 "/>
    <EDGE id="6" from="20" to="28" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
    <EDGE id="7" from="0" to="10" active="Y" covered="N" infeasible="N" effectivetcs=""/>
    <EDGE id="8" from="33" to="41" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
    <EDGE id="9" from="41" to="33" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
    <EDGE id="10" from="20" to="25" active="Y" covered="N" infeasible="N" effectivetcs=""/>
    <EDGE id="11" from="28" to="41" active="Y" covered="Y" infeasible="N" effectivetcs="0007
"/>
  </All-Edges-ei>
 </METHOD>
<TEST_SET last_id="8">
   <TEST_CASE label="0001" host="localhost" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0002" host="null" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0003" host="localhost" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0004" host="null" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0005" host="localhost" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0006" host="null" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0007" host="localhost" active="Y" success="U" fail="U"></TEST_CASE>
   <TEST_CASE label="0008" host="null" active="Y" success="U" fail="U"></TEST_CASE>
</TEST_SET>
```

Figura 4.35 – Parte do XML gerado para o projeto `Agrupamentos.jbt`

Tabela 4.8 – Resumo dos requisitos cobertos pelos casos de teste da aplicação

Conjunto de teste	Requisitos cobertos para o método fat
testAgrupamentos1	-
testAgrupamentos2	-
testAgrupamentos3	-
testAgrupamentos4	1, 2, 3, 5, 6, 8, 9 e 11

O trecho de XML, na Figura 4.36, é parte do arquivo `Agrupamentos.exj` após a captura das informações na ferramenta JaBUTi. Observa-se que o `testAgrupamentos4` cobriu para o método `fat`, segundo o critério Todos-Arcos os requisitos com T (*True*) em destaque e estão de acordo com a Tabela 4.8.

```
<void method="put">
     <string>testAgrupamentos4/Fat.fat(I)J/2</string>
     <array class="java.lang.String" length="14">
      <void index="0">
       <string>F</string>
      </void>
      <void index="1">
       <string>T</string>
      </void>
      <void index="2">
       <string>T</string>
      </void>
      <void index="3">
       <string>T</string>
      </void>
      <void index="4">
       <string>F</string>
      </void>
      <void index="5">
       <string>T</string>
      </void>
      <void index="6">
       <string>T</string>
      </void>
      <void index="7">
       <string>F</string>
      </void>
      <void index="8">
       <string>T</string>
      </void>
      <void index="9">
       <string>T</string>
      </void>
      <void index="10">
       <string>F</string>
      </void>
      <void index="11">
       <string>T</string>
      </void>
     </array>
   </void>
```

Figura 4.36 – Parte do XML gerado pela FATEsC para o projeto `Agrupamentos.exj`

Foi comentada anteriormente, nas funcionalidades do desenvolvedor do componente, a estratégia utilizada para a captura das informações de cobertura para o componente, na qual considera-se para cada caso de teste apenas os requisitos cobertos por ele nos métodos para o qual ele foi elaborado. O mesmo ocorre para a aplicação, como exemplo, na Figura 4.37 tem-se o caso de teste T_5 elaborado para testar o método MA_2 da aplicação, que por sua vez invoca o método do componente MC_1 que invoca o método do componente MC_2 (seta pontilhada), são consideradas apenas as coberturas dos requisitos de MC_1.

Essa estratégia se complica quando utilizada para a aplicação do usuário do componente. Embora os casos de teste sejam associados aos métodos da aplicação, e não ao componente, a ferramenta analisa o código da aplicação para descobrir quais são os métodos do componente invocados e, assim, efetuar a correta comparação de cobertura, em relação a esse método. Por exemplo, ao analisar o método da aplicação MA_1 encontra-se a invocação do método MC_1 do componente, portanto, os resultados de cobertura obtidos pelo caso de teste t_4 serão considerados para a comparação do método MC_1.

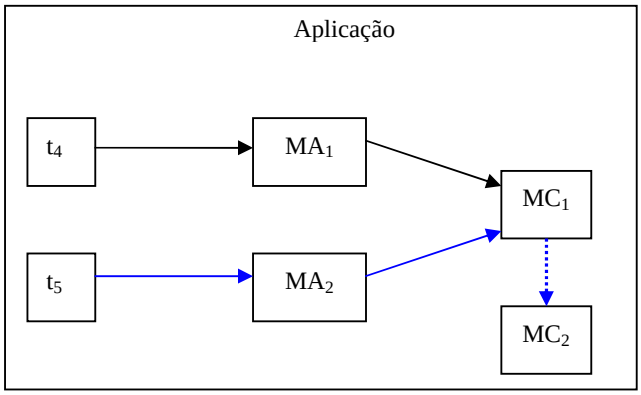


Figura 4.37 – Representação do critério utilizado para a captura das informações de cobertura da aplicação

Feitas essas considerações, pode-se descrever a funcionalidade de comparação. Na Figura 4.38 tem-se uma visão do submenu ***by Method*** do menu ***Compare***, cuja funcionalidade está disponível apenas na visão do usuário do componente. Essa funcionalidade possibilita comparar as informações de cobertura obtidas para o conjunto de teste do usuário, com as informações de cobertura obtidas pelo desenvolvedor do componente, disponibilizadas ao usuário junto com o componente.

Conforme comentando anteriormente, o usuário do componente utiliza os metadados para avaliar os casos de testes gerados para o teste da sua aplicação. Antes de iniciar a comparação dos resultados é necessário extrair o arquivo `Fat.exj` anexado ao componente (`Fat.jar`). Para a comparação dos resultados é aberta uma caixa de diálogo, onde seleciona-se o arquivo `Fat.exj` contendo os metadados.

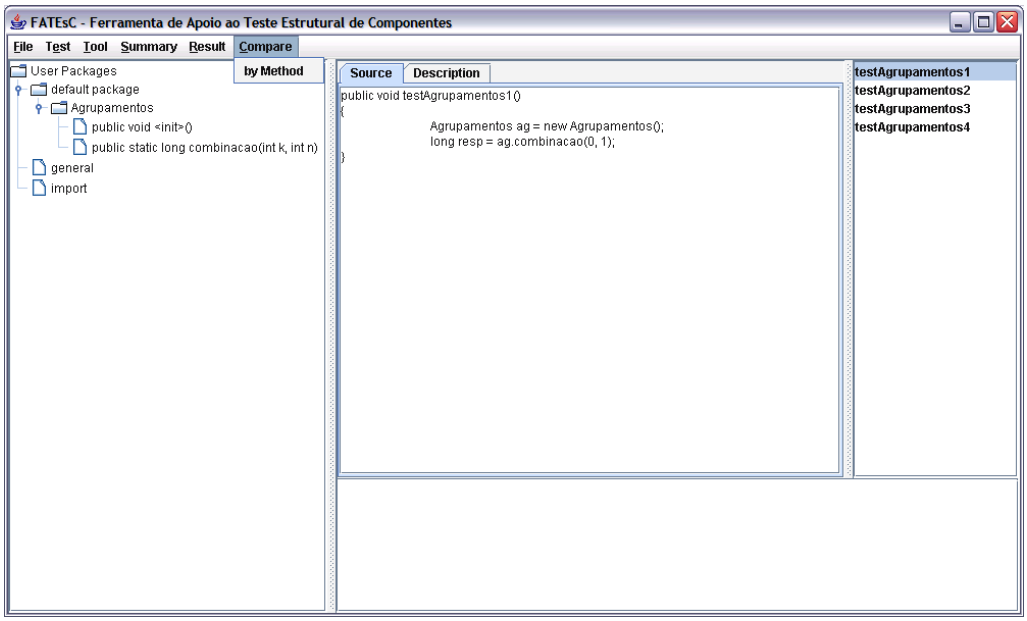


Figura 4.38 – Visão do submenu do menu *Compare*.

Na Figura 4.39 apresenta-se, em forma de tabela, os resultados das comparações feitas pela FATEsC para o método `fat` do componente utilizado pela aplicação `Agrupamento` do usuário do componente. 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. Por exemplo, na Tabela 4.9, feita a partir das Tabelas 4.6 e 4.8 apresentadas anteriormente, tem-se um resumo das informações de cobertura para o testFat4, onde observa-se que o testFat4 cobriu um total de 4 requisitos (1,3,5 e 10) e dentre todos os casos de teste da aplicação o requisito 10 não foi coberto. Portanto, os casos de teste da aplicação cobriram apenas 3 dos 4 requisitos cobertos pelo testFat4. Na quarta e última coluna o cálculo da porcentagem de cobertura obtida pelo usuário do componente em relação à cobertura obtida pelo desenvolvedor do componente.

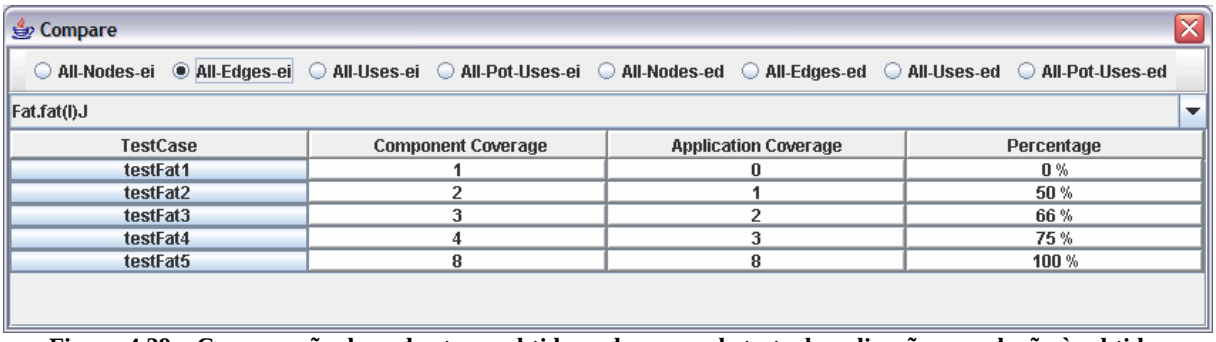


Figura 4.39 – Comparação das coberturas obtidas pelos casos de teste da aplicação em relação às obtidas por cada caso de teste do componente

Tabela 4.9 – Resumo dos requisitos cobertos pelo testFat4 e os requisitos cobertos pelos casos de teste da aplicação

Nome do teste	Requisitos cobertos pelo testFat4 para o método fat	Requisitos cobertos por todos os casos de teste da Aplicação para o método fat
testFat4	1, 3, 5 e 10	1, 2, 3, 5, 6, 8, 9 e 11

De posse das informações resultantes da comparação das informações de cobertura, o usuário do componente deve analisar cada caso onde a cobertura não foi a mesma obtida pelo desenvolvedor. 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 4.40, a descrição diz que foi passado um argumento negativo para o método fat.

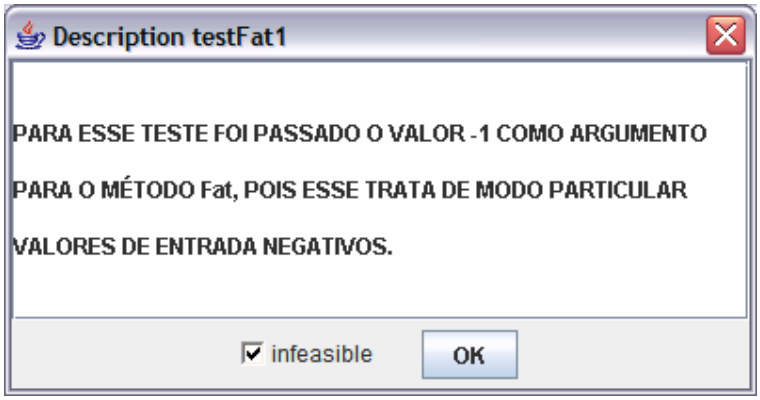


Figura 4.40 - Apresentação da descrição do caso de teste testFat1

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 4.41.

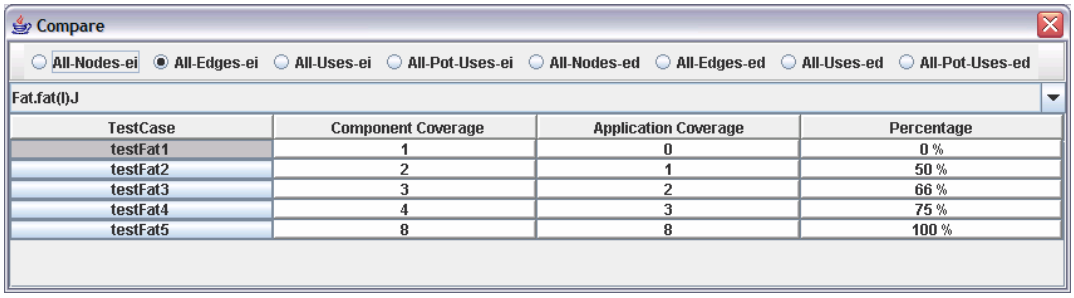


Figura 4.41 - Aparência de um caso de teste (testFat1) marcado como "*infeasible*"

Ao analisar o caso de teste testFat2, que também não foi coberto totalmente, verificando a descrição do caso de teste apresentado na Figura 4.42, o usuário do componente percebe que o cálculo do fatorial de 25 não é -1. Então poderá se lembrar que o componente calcula o fatorial até o valor 20 e que esta limitação deverá ser tratada em sua aplicação, isso mostra como a estratégia adotada por esse trabalho ajuda a revelar defeitos. A aplicação possui um defeito, pois permite-se invocar o método `fat` com qualquer valor inteiro, mesmo os acima de 20.

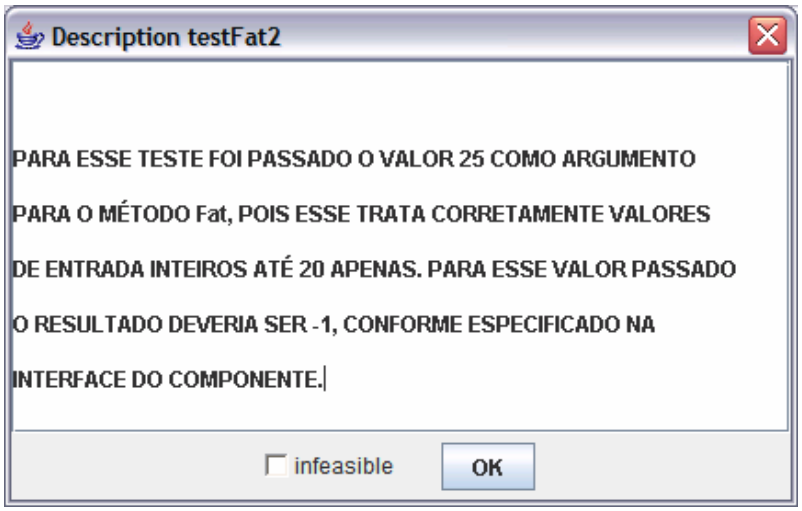


Figura 4.42 - Apresentação da descrição do caso de teste testFat2

Ao analisar o caso de teste testFat3, também não coberto, verificando a descrição do caso de teste apresentado na Figura 4.43, o usuário do componente perceberá que o valor

passado como argumento foi 0 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, marcando-o como "*infeasible*".

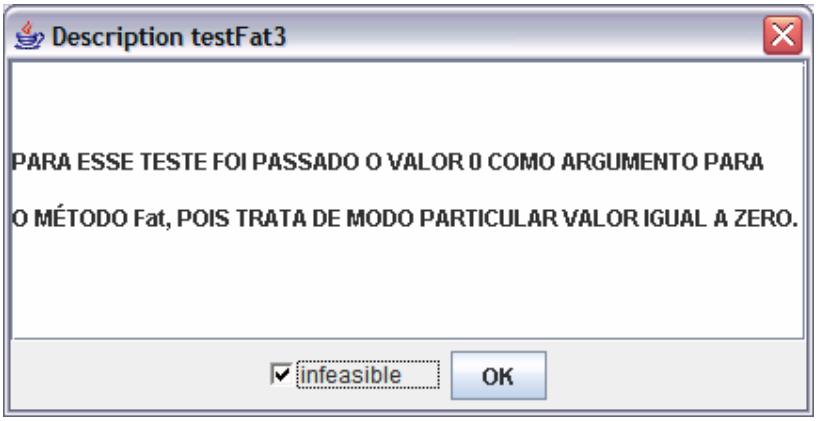


Figura 4.43 - Apresentação da descrição do caso de teste testFat3

Ao analisar o caso de teste testFat4, verificando a descrição do caso de teste apresentado na Figura 4.44, 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, por exemplo, k = 10 e n = 2. Isso faria com que os requisitos relativos ao testFat4 ainda não cobertos, fossem cobertos.

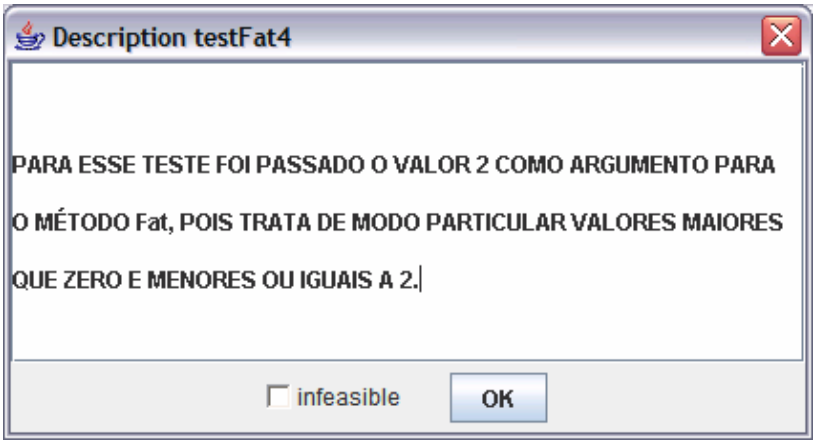


Figura 4.44 - Apresentação da descrição do caso de teste testFat4

O usuário do componente deve alterar sua aplicação para aceitar apenas valores que respeitem limite do cálculo de fatoriais até o valor 20. Deve também acrescentar o teste com argumentos k = 10 e n = 2. Com isso, a comparação das coberturas fica conforme mostrado na

Figura 4.45, ou seja, com três casos de teste que não podem ser reproduzido para a aplicação (cinza escuro) e dois casos de teste com coberturas iguais ao do desenvolvedor do componente. Conclui-se, portanto, que os casos de teste do usuário do componente estão adequados para exercitar a integração do componente em sua aplicação.

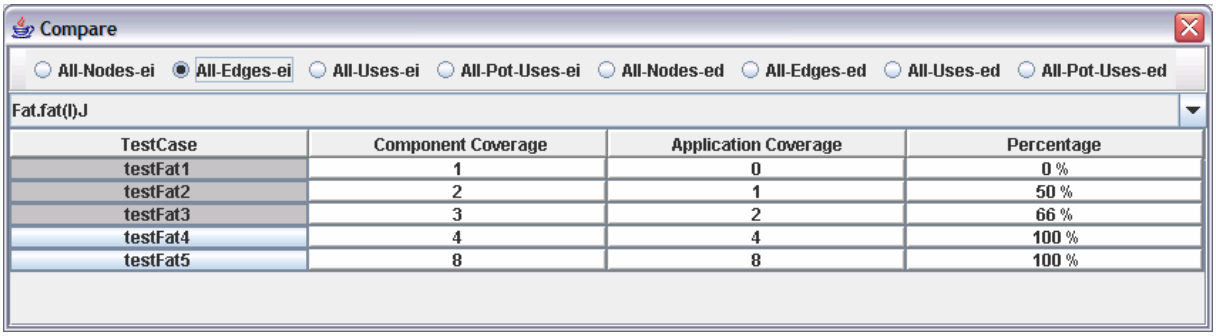


Figura 4.45 – Resultado da comparação das informações de cobertura após as análises

4.4 Considerações finais

Neste capitulo apresentaram-se o embasamento teórico da proposta deste trabalho, a proposta de uma estratégia de teste para desenvolvimento baseado em componentes, detalhes do desenvolvimento da FATEsC e a apresentação de suas funcionalidades por meio de um exemplo simples. No exemplo utilizou-se a classe `Fat`, que faz o cálculo de fatorial, como componente e foram elaborados os casos de teste segundo o critério Todos-Arcos e descrições de cada caso de teste. Os casos de teste elaborados para o componente cobriram 100% dos requisitos de teste.

Também no exemplo, foi utilizada como aplicação a classe Agrupamentos que faz o cálculo de combinação simples utilizando a classe **`Fat`** (componente). Foram elaborados casos de teste segundo o critério Todos-Arcos para o teste da aplicação. Na comparação das coberturas obtidas pelos casos de teste da aplicação em relação às coberturas dos casos de teste do componente constatou-se, com a ajuda das descrições dos casos de teste do componente, que três casos de teste não puderam ser reproduzidos para a aplicação em virtude da aplicação não acessar o componente com determinados valores. Outra constatação foi que

a aplicação não tratava a restrição do componente em não calcular o fatorial para valores maiores do que 20 no que mostrou um grande benefício.

A seguir será apresentado um estudo de caso utilizando como componente o pacote `br.jabuti.lookup` da ferramenta JaBUTi.

5. ESTUDO DE CASO

Como estudo de caso, utilizou-se o pacote `br.jabuti.lookup` como componente e desenvolveu-se um aplicativo que utiliza algumas funcionalidades do pacote. Esse pacote faz parte da ferramenta JaBUTi e optou-se por ele, pois não há muita dependência de outros pacotes da ferramenta, ficando assim o mais próximo do que se deve esperar de um componente.

O pacote `br.jabuti.lookup`, conforme apresentado na Figura 5.1, é composto de quatro classes: ***Program***, ***RClass***, ***RClassCode*** e ***ClassClosure***.

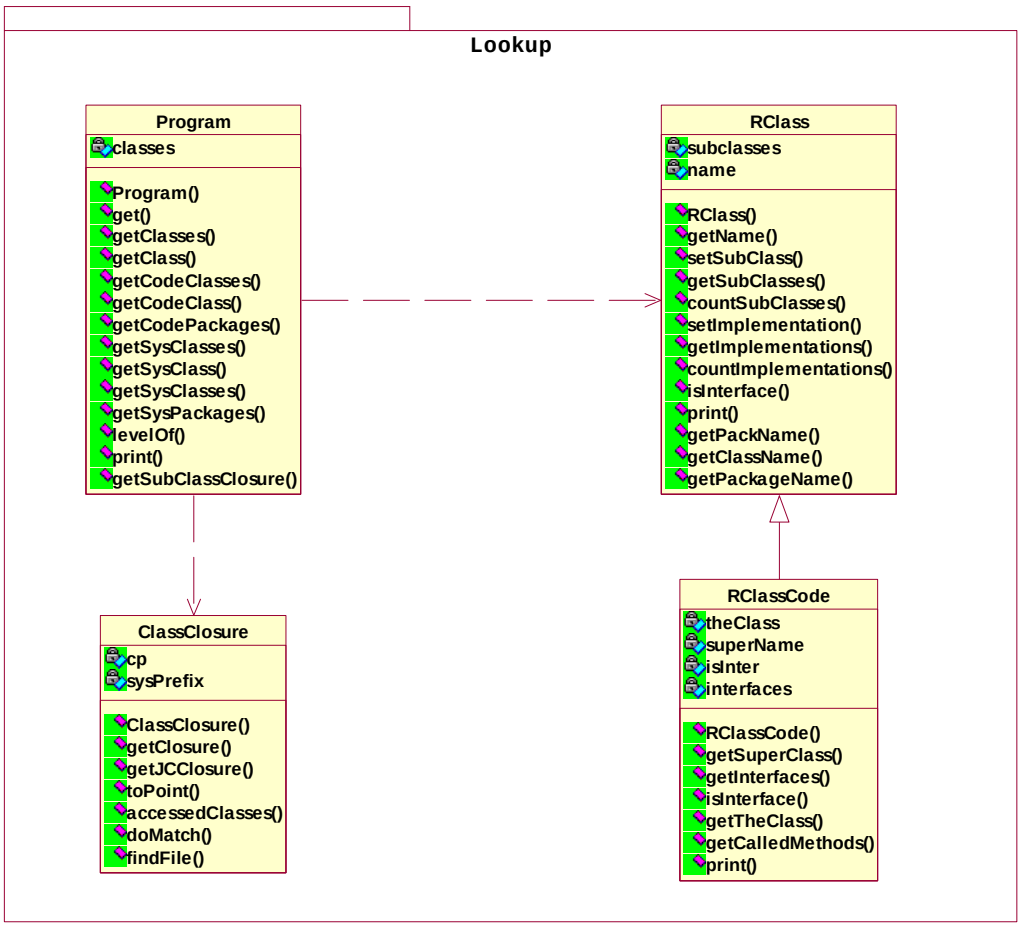


Figura 5.1 – Diagrama de classes do pacote Lookup

A classe ***Program*** é uma abstração de um programa de vários arquivos .class. Os arquivos .class podem ter subclasses e implementações. Em um objeto ***Program***, delimitam-se dois tipos de arquivos .class: aqueles que são de interesse para a definição do programa e aqueles que não são. Isso permite, por exemplo, que sejam excluídas classes como as da API Java, que são utilizadas em um programa, mas que na realidade não fazem parte da sua estrutura.

As classes de interesse são representadas pelo objeto **RClassCode** e classes periféricas aparecem como objetos **RClass**. A classe ***RClass*** é usada para armazenar informações sobre uma determinada classe periférica do programa. Por exemplo, um objeto **RClass** armazena informações sobre quais subclasses herdam as características de uma determinada classe **X** ou quais as classes a implementam, se a classe **X** for uma interface. A classe ***RClassCode*** é usada no contexto de um programa para representar as classes desse programa. Por exemplo, um objeto **RClassCode** herda as características da classe ***RClass***, portanto, além de armazenar o que já foi descrito anteriormente para o objeto **RClass**, também armazena as informações completas sobre a própria classe. A classe ***ClassClosure*** é usada para explorar a estrutura do programa e tem alguns métodos para procurar as dependências de classes do programa. O programador pode especificar, nas buscas, se quer ou não incluir classes de sistema e pode também determinar quais pacotes podem ser ignorados na busca. No Apêndice C é apresentada uma documentação, no formato *Javadoc*, dessas classes do pacote `lookup`.

O objetivo deste estudo de caso é validar, com dados mais substanciais, as funcionalidades da ferramenta FATEsC e verificar se a estratégia de teste apresentada por esse trabalho apresenta benefícios. Um ponto fraco deste estudo é que 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 Figura 5.2 e na Tabela 5.1.

Também foram feitas as descrições de cada caso de teste com informações que apoiassem as análises das coberturas e que dessem indicações de como os casos de teste foram elaborados. Para o estudo de caso foram consumidas 40 horas, considerando todas as atividades envolvidas: desenvolvimento da aplicação, elaboração e execução dos casos de teste e análise dos resultados.

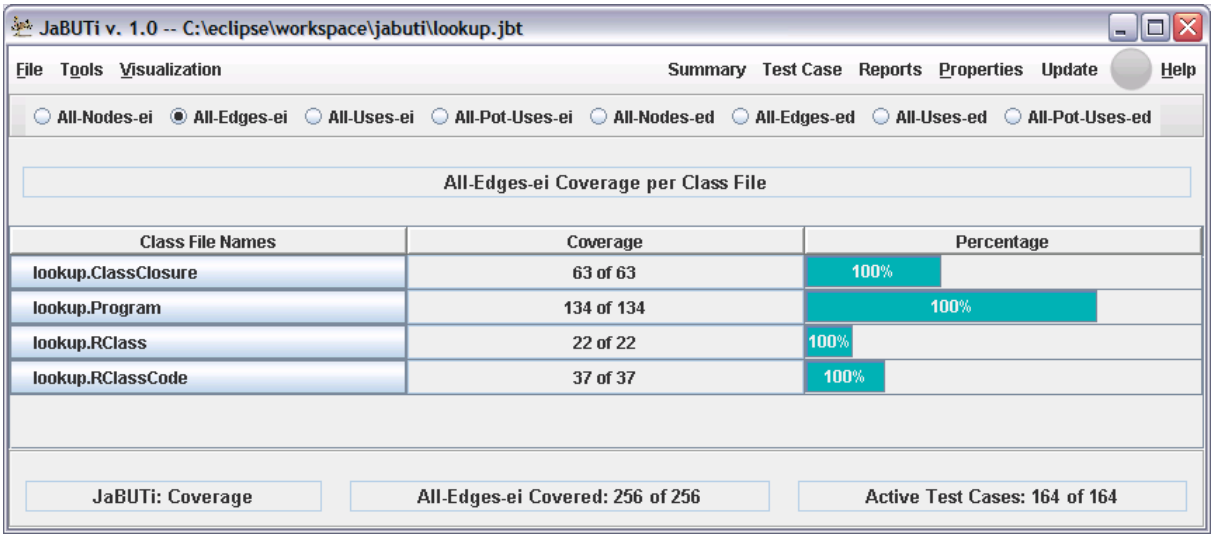


Figura 5.2 – Sumário por classe apresentado pela Ferramenta JaBUTi

Tabela 5.1 – Resumo geral por classes do componente

Classes do componente	Quantidade métodos públicos	Quantidade de requisitos	Quantidade de requisitos cobertos	Porcentagem de cobertura	Quantidade de 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

Na Tabela 5.2 apresenta-se um resumo das coberturas obtidas pelos casos de teste do componente `Lookup`, para os métodos que foram utilizados pela aplicação `UseLookup`. No Apêndice A pode-se ver os sumários gerados na ferramenta FATEsC, os quais foram utilizados para construir a Tabela 5.2.

Tabela 5.2 – Resumo por métodos do componente utilizados pela aplicação `UseLookup`

Métodos públicos do componente utilizados pela Aplicação	Casos de teste	Quantidade de requisitos	Quantidade de requisitos cobertos	Porcentagem de cobertura
Program(ZipFile zippedFile, boolean noSys, String toAvoid)	testProgram2	13	13	100%
	testProgram32		13	
Program(JarFile jarFile, boolean noSys, String toAvoid)	testProgram1	13	13	100%
	testProgram31		13	
RClass get(String s)	testProgram5	0	-	-
String[] getCodeClasses(String packName)	testProgram10	8	7	100%
	testProgram27		6	
String[] getCodePackages()	testProgram11	4	4	100%
String getClassName(String x)	testRClass14	2	1	100%
	testRClass15		1	
String[] getCalledMethods(String assinatura)	testRClassCode5	15	13	100%
	testRClassCode7		6	

A aplicação `UseLookup`, que usa o componente `Lookup` e está 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. Na Figura 5.3 é apresentado o diagrama de classes da aplicação `UseLookup`.

Essa aplicação permite, a partir de um arquivo .zip ou .jar mostrar a estrutura desse arquivo em forma de árvore conforme mostrado na Figura 5.4, e ao selecionar um método da estrutura e ir ao menu ***Find Calls*** e selecionar ***Find***, são listados todos os métodos invocados pelo método escolhido à direita da estrutura em árvore. Outra forma de executar essa funcionalidade é selecionar ***Find Out*** no menu ***Find Calls*** e escolher um novo arquivo .jar na caixa de dialogo. Após a escolha do arquivo, é possível digitar o nome do método, do qual deseja-se saber os métodos invocados, na caixa de *input* conforme apresentado na Figura 5.5.

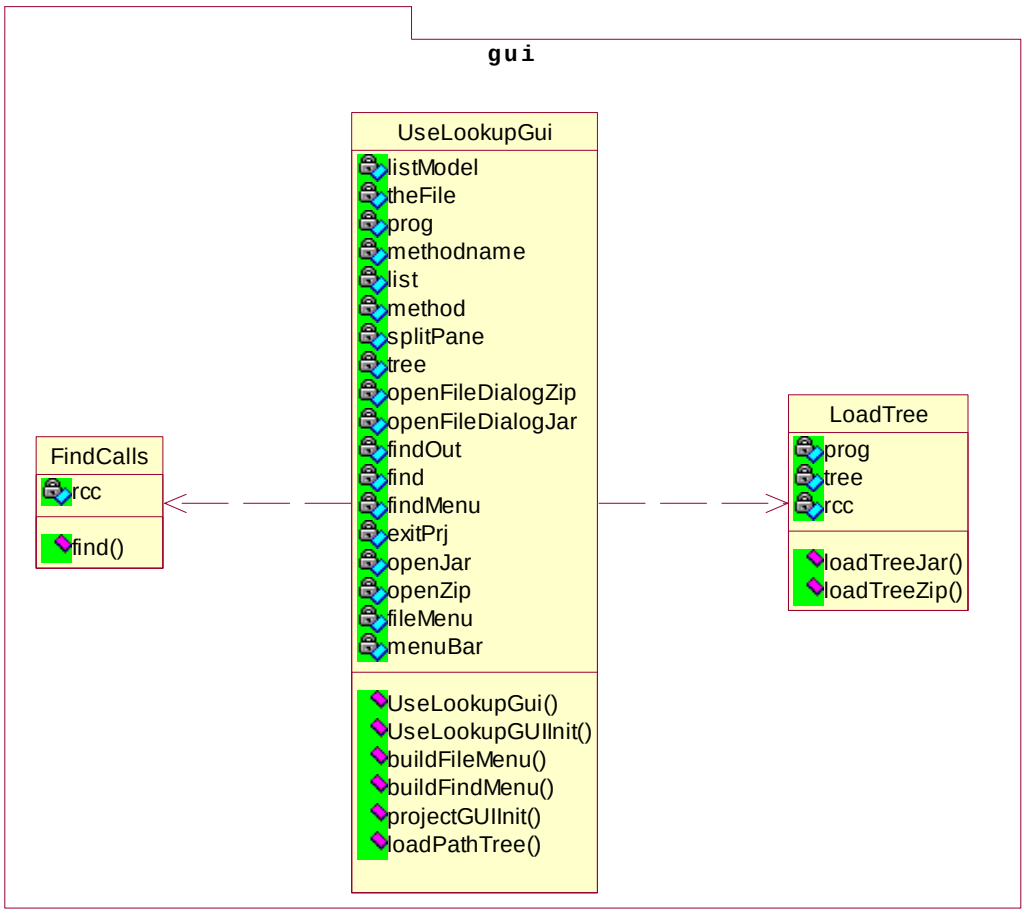


Figura 5.3 – Diagrama de classes da aplicação

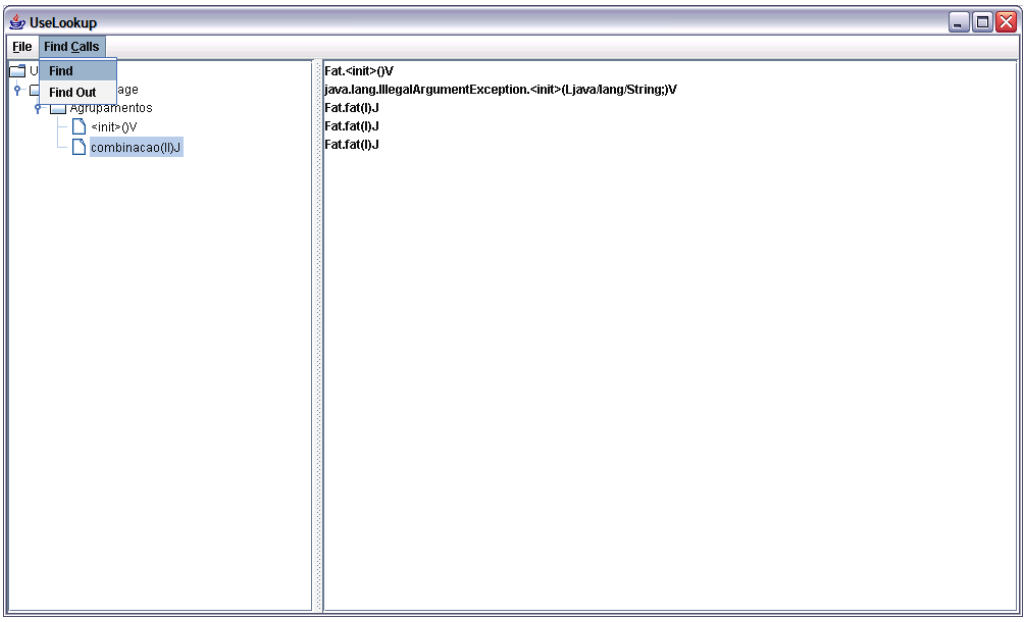


Figura 5.4 – Menu *Find Calls* da aplicação `UseLookup`

Figura 5.5 – Dialogo para a digitação da assinatura do método

Foram elaborados três casos de teste para os métodos `find`, `loadTreeJar` e `loadTreeZip` da aplicação `UseLookup` e são esses métodos que invocam os 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 da aplicação e para os métodos do componente utilizados pela aplicação. Na Tabela 5.3 é apresentado um resumo das coberturas obtidas pelos casos de teste da aplicação em relação aos métodos do componente.

Tabela 5.3 – Resumo por métodos do componente utilizados pela aplicação `UseLookup`

Métodos públicos do componente utilizados pela Aplicação	Casos de teste da Aplicação que cobriram os métodos do componente	Quantidade de requisitos	Quantidade de requisitos cobertos	Porcentagem de cobertura
Program(ZipFile zippedFile, boolean noSys, String toAvoid)	testLoadTreeZip1	13	13	100%
Program(JarFile jarFile, boolean noSys, String toAvoid)	testLoadTreeJar1	13	13	100%
RClass get(String s)	-	0	-	-
String[] getCodeClasses(String packName)	testLoadTreeZip1	8	7	87%
	testLoadTreeJar1		7	
String[] getCodePackages()	testLoadTreeZip1	4	4	100%
	testLoadTreeJar1			
String getClassName(String x)	testLoadTreeZip1	2	1	50%
	testLoadTreeJar1		1	
String[] getCalledMethods(String assinatura)	testFindCalls1	15	13	86%

A comparação das coberturas obtidas pelos casos de teste do componente em relação aos casos de teste da aplicação estão resumidos na Tabela 5.4, que foi elaborada a partir dos resultados apresentados pela ferramenta FATEsC e que podem ser vistos no Apêndice B.

Observa-se, na Tabela 5.4 que em três casos as coberturas foram diferentes, nos casos de teste `testProgram27`, `testRclass14` e `testRClassCode7` necessitando de análise para verificar a adequação dos casos de teste elaborados para a aplicação.

Tabela 5.4 – 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%

Para auxiliar na análise das diferenças de cobertura obtidas pelos casos de teste elaborados para o componente com as obtidas pelos casos de teste elaborados para a aplicação, foram utilizadas as descrições dos casos de teste `testProgram27`, `testRClass14` e `testRClassCode7`. Para o `testProgram27`, como pode ser visto na Figura 5.6, buscaram-se as classes para um pacote que não faz parte da estrutura principal das classes existentes em um arquivo em questão. Como para a aplicação somente são buscadas as classes para os pacotes que fazem parte da estrutura principal das classes de um determinado arquivo selecionado, conclui-se que esse caso de teste não pode ser reproduzido para a aplicação e, portanto, pode-se marcá-lo como "*infeasible*". O caso de teste

`testProgram27` passa a ser apresentado em cor cinza escuro para que represente a análise feita, conforme apresentado na Figura 5.7.

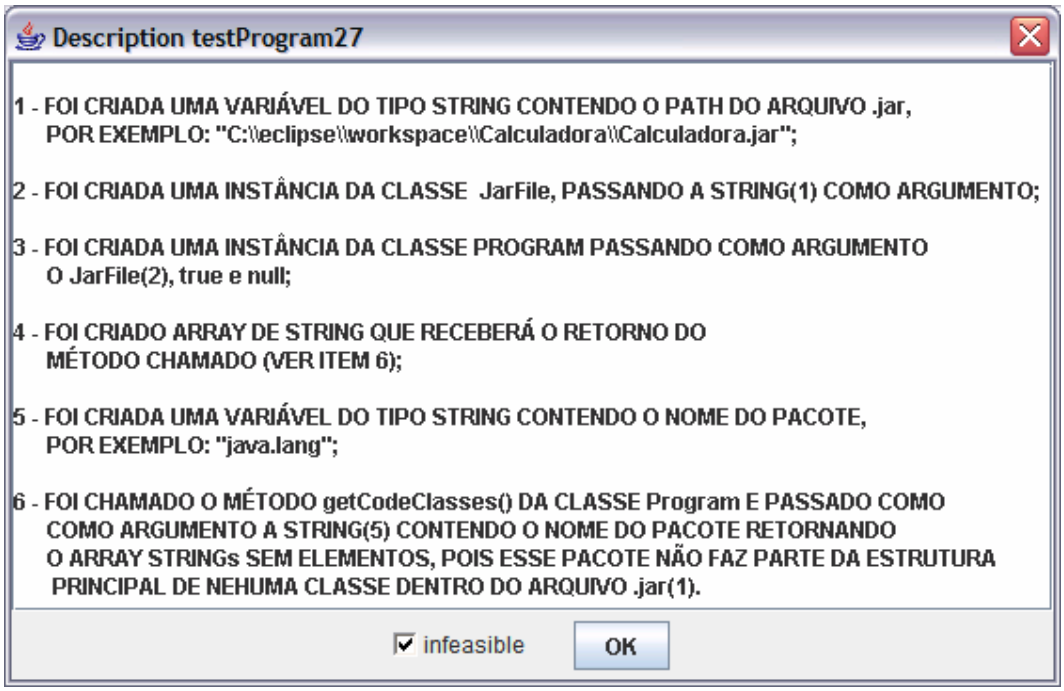


Figura 5.6 – Descrição do caso de teste `testProgram27`

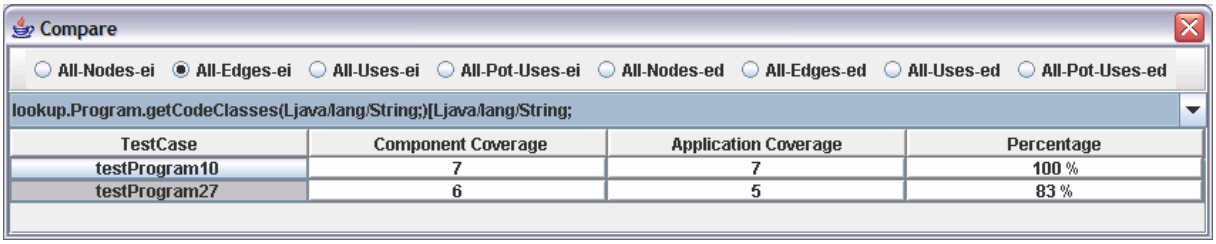


Figura 5.7 – Aparência do caso de teste `testProgram27` após a marcação de "*infeasible*"

Para `testRclass14`, como pode ser visto na Figura 5.8, a descrição indica que ao passar um nome de uma classe que não esteja em um pacote, retorna se o próprio nome da classe. Ao verificar os casos de teste (testLoadTreeZip1 e testLoadTreeJar1) elaborados para a aplicação que cobriram a parte em que o método `getClassName()`do componente é chamado, nota-se que a classe utilizada como argumento está em um pacote e portanto não foi testada a situação em que se utilizasse uma classe que não estivesse em um pacote. Para testar

essa situação, basta criar um novo caso de teste utilizando um arquivo que não contenha classes dentro de pacotes, pois essa é uma situação que pode ocorrer para a aplicação.

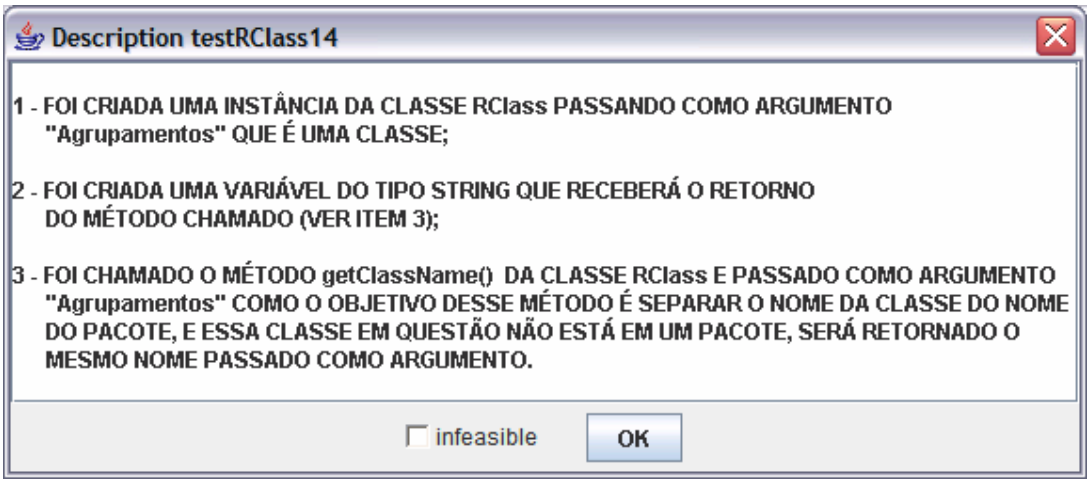


Figura 5.8 – Descrição do caso de teste testRClass14

Para `testRclassCode7`, como pode ser visto na Figura 5.9, a descrição indica que o argumento utilizado na chamada do método do componente `getCalledMethods`() não está nas classes do arquivo selecionado, portanto o retorno é null. Isso faz com que se perceba que essa situação não foi tratada para a aplicação e que pode ser um problema ao utilizar-se a funcionalidade ***Find Out***, pois o argumento é digitado em uma caixa de diálogo.

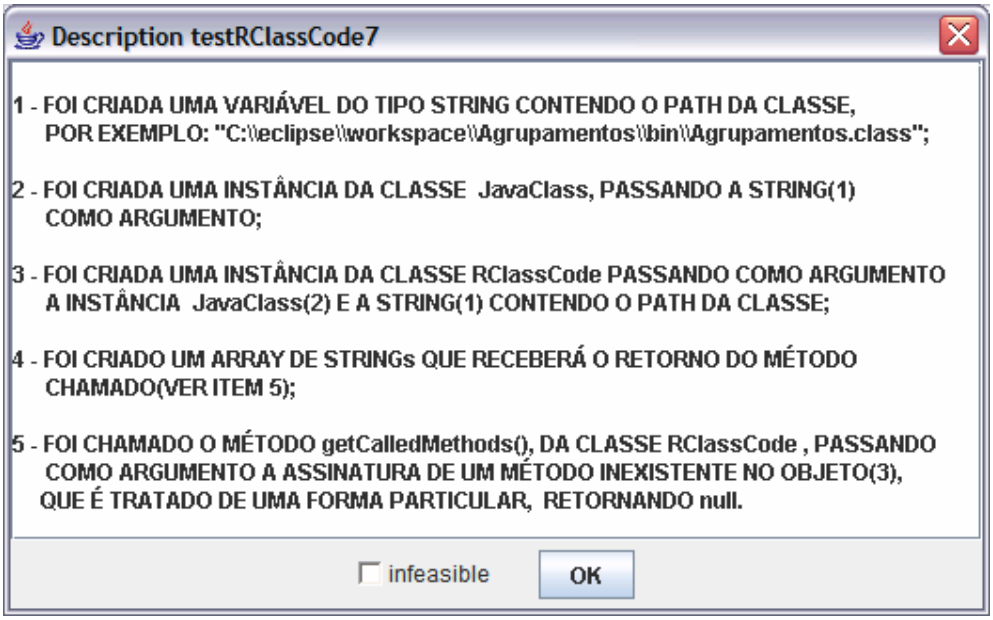


Figura 5.9 – Descrição do caso de teste `testRClass7`

Após alterar a aplicação e elaborar mais dois casos de teste refazendo todo o processo de execução e captura das informações de cobertura usando as ferramentas FATEsC e JaBUTi, conforme mostram as Figuras 5.10 e 5.11, chegou-se à mesma cobertura obtida pelos casos de teste `testRClass14` e `testRClassCode7`, elaborados para o componente.

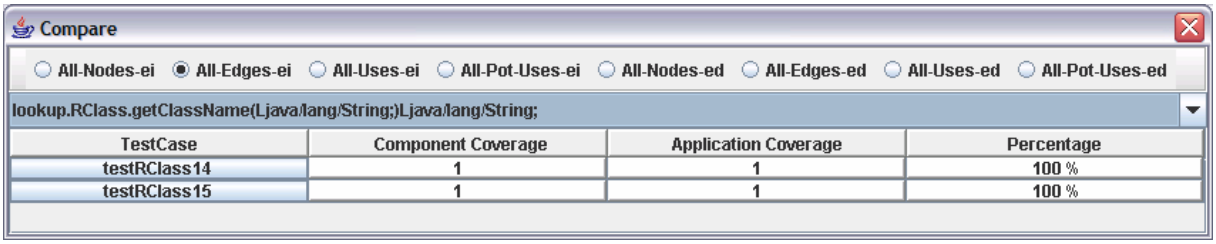


Figura 5.10 – Cobertura obtida pelos casos de teste da aplicação com relação aos casos de teste testRClass14

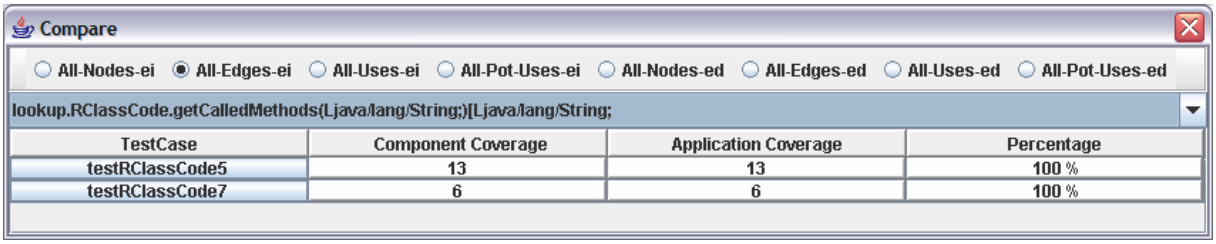


Figura 5.11 - Cobertura obtida pelos casos de teste da aplicação com relação aos casos de teste `testRClass7`

Para o estudo de caso utilizou-se como componente o pacote lookup e foram elaborados 82 casos de teste que cobriram 100% dos requisitos para o critério Todos-Arcos. Foi elaborada uma aplicação que utiliza sete métodos públicos do componente. Para testar a aplicação foram elaborados, primeiramente, 3 casos de teste para os métodos da aplicação que invocam métodos do componente. Após a execução e captura dos dados de cobertura do componente e aplicação, foram comparadas as coberturas obtidas pelos casos de teste do componente em relação aos casos de teste da aplicação.

Na comparação observou-se que para apenas 3 casos de teste as coberturas diferiram. Com o apoio das descrições dos casos de teste foram analisadas as situações que envolviam as diferenças e chegou-se à conclusão que um dos casos de teste não poderia ser reproduzido

para a aplicação e, portanto, foi marcado como "*infeasible*". O segundo caso também foi analisado e verificou-se que a situação envolvendo esse caso de teste deveria ser testada para a aplicação também e assim foi criado um caso de teste que a validasse. Para o terceiro caso, percebeu-se que havia uma situação em que ao invocar o método o retorno poderia ser null, o que não havia sido tratado na aplicação. A aplicação foi alterada e um novo caso de teste foi elaborado para testar essa nova situação.

Desta forma, foram elaborados ao todo 5 casos de teste para a aplicação e, exceto pelo caso de teste marcado como "*infeasible*", os demais tiveram suas coberturas equiparadas às coberturas obtidas pelos casos de teste do componente.

Durante o estudo de caso, até mesmo pela maior utilização da ferramenta FATEsC, percebeu-se a possibilidade de melhorias, principalmente na apresentação das informações e essas melhorias foram promovidas. Passou-se a apresentar, tanto a sumarização das coberturas alcançadas pelos casos de teste, como a comparação do entre os resultados de cobertura alcançados pelos casos de teste do desenvolvedor e usuário, por método.

Também houveram algumas dificuldades na execução desse estudo de caso. A primeira foi em associar os casos de teste gerados na FATEsC com os *labels* gerados pela JaBUTi no momento de capturar as informações de cobertura, pois a JaBUTi não leva em consideração os nomes dos casos de teste e cria uma seqüência numérica de acordo com a execução dos casos de teste. Para evitar uma associação errônea, seria interessante que a Ferramenta JaBUTi incluísse como identificador os nomes dos casos de teste. A segunda dificuldade, também relacionada à primeira, é que a cada alteração de um caso de teste que já fora executado na JaBUTi, é necessário que se crie um novo projeto e se refaça todo o processo, pois como não há uma associação pelo nome do caso de teste, a JaBUTi acrescenta esse caso de teste alterado no final da lista dos casos de teste executados anteriormente. A terceira dificuldade foi elaborar os casos de teste para cobrirem 100% dos requisitos do componente, em virtude da falta de conhecimento no componente utilizado no estudo de caso.

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.

5.1 Considerações finais

Apesar de o estudo de caso não ter sido realizado nas condições ideais, ou seja, em um ambiente de desenvolvimento de software baseado em componente, revelou que pode ajudar o usuário do componente na avaliação da adequação dos seus casos de teste em exercitar as partes do componente que são utilizadas por sua aplicação. Outro ponto positivo vem no sentido de que, por meio das descrições dos casos de teste elaboradas pelo desenvolvedor do componente, pode-se ter indicações de possíveis problemas não previstos pelo usuário do componente, ficando claro também, que para se obter esses benefícios depende-se da qualidade das informações contidas nas descrições dos casos de teste.

6. CONCLUSÃO

Neste trabalho foi apresentada uma estratégia de teste para desenvolvimento baseado em componente, utilizando a técnica estrutural de teste. Para isso, foram apresentados os conceitos básicos envolvendo o paradigma de desenvolvimento com componentes. Além disso, também foram apresentadas metodologias para desenvolvimentos baseado em componente como a de Cheesman e Daniels (2000) e a de Pressman (2002) e uma revisão sobre testes, abordando as fases de teste e algumas técnicas de teste.

A partir do estudo realizado sobre o problema com a falta de informação e também com base nas propostas apresentada na literatura que visam promover a troca de informação entre o desenvolvedor e usuário do componente, especificamente as que propõem fornecer metadados, percebeu-se que essas propostas não tratam objetivamente quais seriam as informações trocada entre eles.

Diante desse contexto, desenvolveu-se a ferramenta FATEsC que associada à ferramenta JaBUTi, apóia a estratégia de teste proposta por esse trabalho. Por meio de um exemplo, foi descrita a estratégia proposta nesse trabalho que consiste em o desenvolvedor do componente disponibilizar metadados de teste anexados ao componente para que possa apoiar o usuário do componente na atividade de teste.

Foi realizado um estudo de caso que mostrou que a estratégia ajuda a revelar defeitos e também o quão adequado estão os casos de teste do usuário do componente para testar a integração do componente na aplicação, por meio das informações disponibilizadas na forma de metadados.

6.1 Contribuição

A principal contribuição desse trabalho para testes de software baseado em componentes foi a estratégia proposta, que possibilita a troca de informações entre desenvolvedor e usuário do componente, apresentando uma solução objetiva para o problema. Na estratégia proposta, o desenvolvedor do componente utiliza a FATEsC para criar os casos

de teste, no formato JUnit, utilizados para testar o componente e também faz uma descrição para cada caso de teste. Os testes são realizados na ferramenta JaBUTi, que gera as informações de cobertura para os requisitos de teste de alguns critérios da técnica estrutural, para os métodos do componente. A FATEsC captura as informações de cobertura de teste e anexa-as juntamente com as descrições dos casos de teste ao componente.

O usuário do componente, por sua vez, também cria casos de teste para testar a sua aplicação que utiliza o componente, executa os testes e captura os dados de cobertura obtidos por seus casos de teste. Utilizando a FATEsC o usuário do componente pode comparar a cobertura de teste obtida por seus casos de teste e a cobertura de teste disponibilizada nos metadados. Para as análises das coberturas de teste o usuário do componente pode ver as descrições dos casos de teste também fornecidas nos metadados. Resultados semelhantes indicam que os casos de teste do usuário do componente estão adequados para testar a integração do componente em sua aplicação. Caso contrário, pode-se verificar nas descrições dos casos de teste se eles podem ser reproduzidos na aplicação, a fim de obter a mesma cobertura de teste obtida pelo desenvolvedor do componente.

Fica bastante claro que o sucesso dessa estratégia está diretamente ligado à qualidade das informações disponibilizadas nos metadados.

6.2 Trabalhos futuros

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.

É importante separar os metadados dos dados do projeto gerado pela FATEsC, pois atualmente são acrescentados dados ao componente que não serão utilizados. Também poderia ser interessante permitir ao desenvolvedor do componente decidir quais as informações de cobertura devem ser geradas. Atualmente são geradas todas as informações de cobertura dos requisitos de teste para todos os critérios implementados na ferramenta JaBUTi. Poderia ser interessante que se fornecessem apenas as informações de cobertura dos requisitos de um critério em específico.

Incorporar na ferramenta FATEsC recursos que facilitam a execução de algumas atividades visando produtividade e qualidade. Por exemplo, na criação dos casos de teste poder-se-ia incorporar um analisador de sintaxe em tempo real e na criação das descrições dos casos de teste, poder-se-ia incorporar algumas funcionalidades de edição de texto (cor, fonte, etc.).

REFERÊNCIAS

BARROCA, E.; **GIMENES**, I.; **HUZITA**, E.: **Desenvolvimento baseado em componentes**. Editora Ciência Moderna, 2005.

BELLOIR, N.; **BRUEL**, J.; **BARBIER**, F.: **Whole-Part Relationships for Component Combination**. In EUROMICRO Conference "New Waves in System Architecture", IEEE Computer Society, 2003, p. 86-91.

BEYDEDA, S.; **GRUHN**, V.: **State of the art in testing components**. In: International Conference on Quality Software (QSIC), IEEE Computer Society Press, 2003, p. 146-153.

BHOR, A.: **Software Component Testing Strategies**. Techinical Report UCI-ICS-02-06, University of California, Irvine, 2001.

BUNDELL, G. A.; **LEE**, G.; **MORRIS**, J.; **PARKER**, K.; **LAM**, P.: **A software component verification tool** . In International Conference on Software Methods and Tools(SMT), IEEE Computer Society Press, 2000, p. 137-146.

BURNSTEIN, I.: **Pratical Software Testing**, Springer-Verlag, 2002

CHEESMAN, J.; **DANIELS**, j.: **UML components: A simple process for specifying component-based software**. Addison Wesley, 2000.

CRNKOVIC, I.; **LARSSON**, M.: Component-**Based Software Engineering – New Paradigm of Software Development**. In: Euromicro Workshop on Component-Based Software Engineering, 2001, p. 127-133.

DELAMARO, M.; **MALDONADO**, J.: **Uma visão sobre a aplicação da análise de mutantes**. Technical Report 133, ICMC/USP, São Carlos, 1993.

DEMILLO, R. A.: **Mutation Analysis as a Tool for Software Quality Assurance**. Proc. of COMPSAC 80, Chicago - IL, 1980.

EDWARDS, S.: **Toward reflective metadata wrappers for formally specified software components**. In OOPSLA Workshop Specification and Verification of Component-Based Systems, 2001, p. 14-21.

FREEDMAN, R. S.: **Testability of Software Components**. IEEE Transactions on Software Engineering, 1991, p. 553-564.

GAO, J.; **GUPTA**, K.; **SHIM**, S.: **On building testable software components**. In: COTS-Based Software Systems (ICCBCC), Springer Verlag, 2002.

HARROLD, M. J.; **LIANG**, D.; **SINHA**, S.: **An approach to analyzing and testing component-based-systems**. In International ICSE Workshop Testing Distributed Component-Based Systems, 1999.

HÖRNSTEIN, J.; **EDLER**, H.: **Test reuse in cbse using built-in tests**. In Workshop on Component-based Software Engineering, Composing systems from components, 2002, p. 4.

LEWIS, T. **The next $10{,}000_2$ years**, part II. IEEE Computer, 1996, p.78-80

LIU, C.; **RICHARDSON**, D.: **Software component with retrospectors**. In International Workshop on the Role of Software Architecture in Testing and Analysis, 1998, p. 63-68.

MA, Y.; **OH**, S.; **BAE**, D.; **KWON**, Y.: **Framework for third Party Testing of Component Software.** In Asia-Pacific Software Engineering Conference (APSEC' 01), 2001.

MALDONADO, J. C.; **BARBOSA**, E. F.; **VINCENZI**, A. M. R.; **DELAMARO**, M. E.; **SOUZA**, S. R. S.; **JINO**, M.: **Introdução ao Teste de Software**, Notas Didáticas do ICMC, São Carlos 2004.

MASSOL, V. e **HUSTED**, T. **"JUnit Em Ação"**. Editora Ciência Moderna, 2005.

ORSO, A.; **HARROLD**, M.; **ROSENBLUM**, D.: **Component metadata for software engineering task**. In International Workshop on Engineering Distributed Object (EDO), Springer-Verlag, 2000, p. 129-144.

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

PRESSMAN, R.: **Engenharia de Software**. Editora McGraw-Hill, 1995.

ROCHA, C. R.; **MARTINS**, E.: **Um Modelo para Construção de Componentes Testáveis**. V Workshop Brasileiro de Testes e Tolerância a Falhas (WTF 2004) – XXII SBRC'2004, vol. 1, pp. 1-10, Gramado, RS, 2004.

SZYPERSKI, C.: **Component Oriented Programming**. Editora Addison-Wesley, 1997.

SOMMERVILLE, I.: **Engenharia de Software**. Editora Addison-Wesley, 2003.

VINCENZI, A. M. R.; **MALDONADO**, J. C.; **DELAMARO**, M. E.; **SPOTO**, E. S.; **WONG**, E.: **Software baseado em componente: uma revisão sobre teste**. In: Desenvolvimento baseado em componente. Editora Ciência Moderna, 2005.

VINCENZI, A.M.R.; **DELAMARO**, M.E.; **WONG**, E.; **MALDONADO**, J.C. **Establishing Structural Testing Criteria for Java Bytecode.** Software, Practice & Experience, v. 36, p. 1513-1541, 2006.

VOAS, J.: **COTS software: The economical choice?** IEEE software, 1998, p.16-19.

VOAS, J.: **Certifying Off-the-Shelf Software Components**. IEEE software, 1998, p.53-59.

WANG, Y.; **KING**, G.; **WICKBURG**, H.: **A method for built-in tests in component-based software maintenance**. In European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society Press, 1999, p. 186.

WEYUKER, E.: **Testing component-based software: A cautionary tale**. IEEE software, 1998, p. 54-59.

WU, Y.; **PAN**, D.; **CHEN**, M.: **Techniques for Testing Component-Based Software**. In International Conference on Engineering of Complex Computer System (ICECCS'01), 2001, p. 222-232.

APÊNDICE A

As figuras apresentadas neste apêndice foram geradas pela FATEsC e foram utilizadas para elaborar a Tabela 5.2. Essas figuras apresentam os resultados de cobertura obtidos por cada caso de teste gerado para uma interface pública do componente segundo o critério Todos-Arcos.

Na primeira coluna são listados os casos de teste. Na segunda coluna a quantidade de requisitos cobertos em relação ao total de requisitos. Na última coluna a porcentagem obtida pela cobertura. Na última linha são apresentados os totais de cobertura e porcentagem alcançados por todos os casos de teste listados na primeira coluna.

Na Figura A1 o resumo das coberturas obtidas pelos casos de teste `testProgram2` e `testProgram32` elaborados para testar o método `Program(ZipFile zippedFile, boolean noSys, String toAvoid).` Na Figura A2 é mostrado o resumo das coberturas obtidas pelos casos de teste `testProgram1` e `testProgram31` elaborados para testar o método `Program(JarFile jarFile, boolean noSys, String toAvoid).` Na Figura A3, nada foi listado, pois o método `RClass get(String s)` não tem requisitos para o critério selecionado. Na Figura A4 é mostrado o resumo das coberturas obtidas pelos casos de teste `testProgram10` e `testProgram27` elaborados para testar o método `String[] getCodeClasses(String packName`). Na Figura A5 é mostrado o resumo das coberturas obtidas pelo caso de teste `testProgram11` elaborado para testar o método `String[] getCodePackages()`. Na Figura A6 é mostrado o resumo das coberturas obtidas pelos casos de `testRClass14` e `testRClass15` elaborados para testar o método `String getClassName(String x)`. Na Figura A7 é mostrado o resumo das coberturas obtidas pelos casos de `testRClassCode5` e `testRClassCode7` elaborados para testar o método `String[] getCalledMethods(String assinatura).`

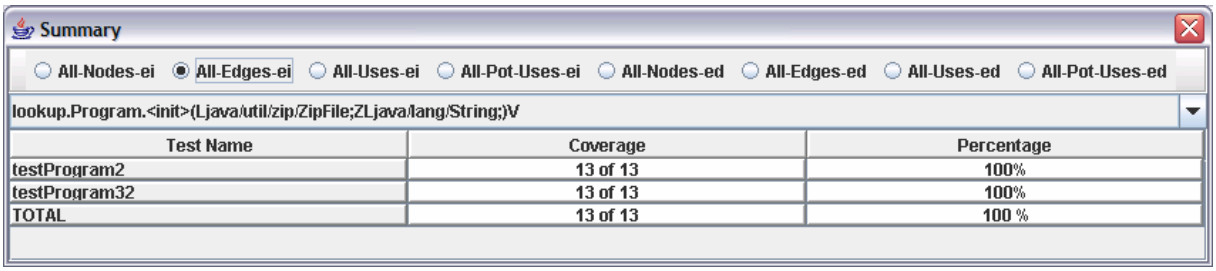


Figura A 1 – Resumo de cobertura do método Program(ZipFile zippedFile, boolean noSys, String toAvoid)

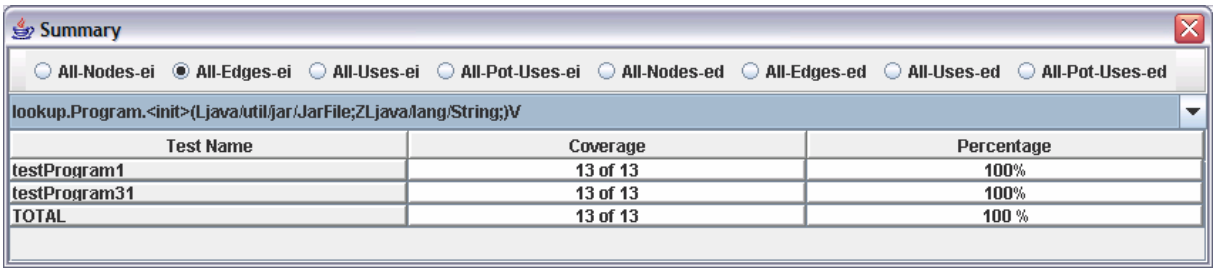


Figura A 2 - Resumo de cobertura do método `Program(JarFile jarFile, boolean noSys, String toAvoid)`

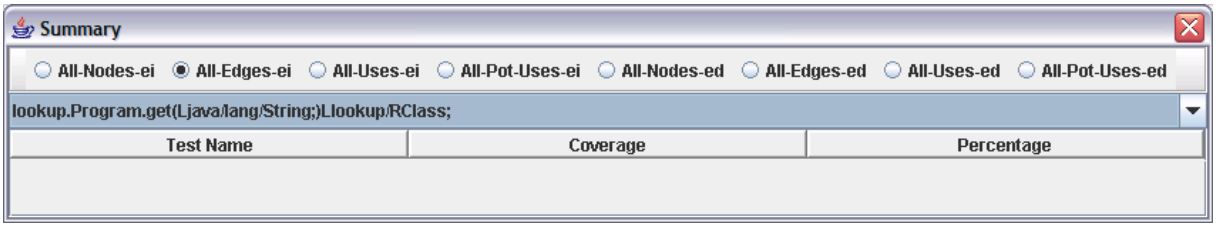


Figura A 3 - Resumo de cobertura do método `RClass get(String s)`

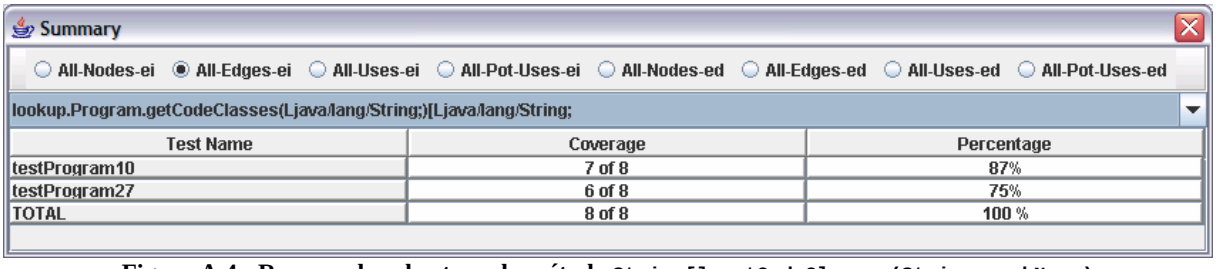


Figura A 4 - Resumo de cobertura do método `String[] getCodeClasses(String packName)`

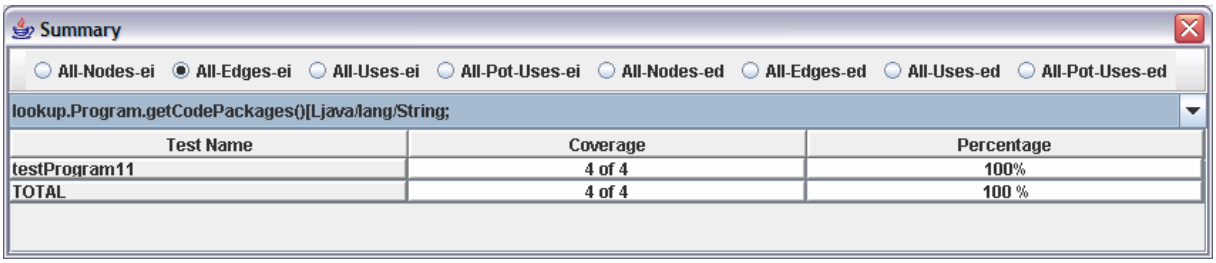


Figura A 5 - Resumo de cobertura do método `String[] getCodePackages()`

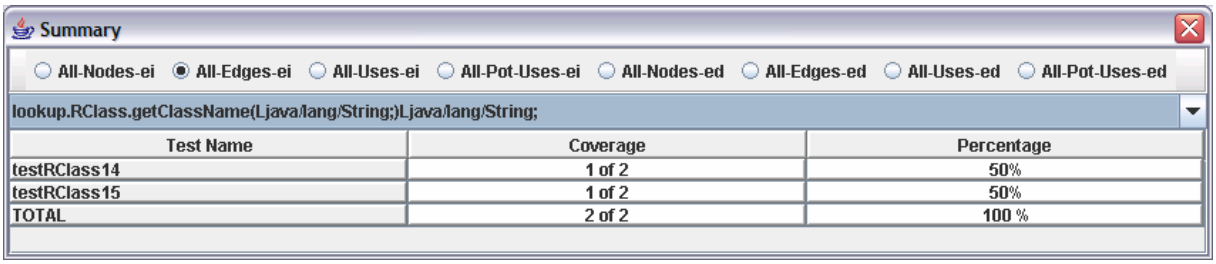


Figura A 6 - Resumo de cobertura do método `String getClassName(String x)`

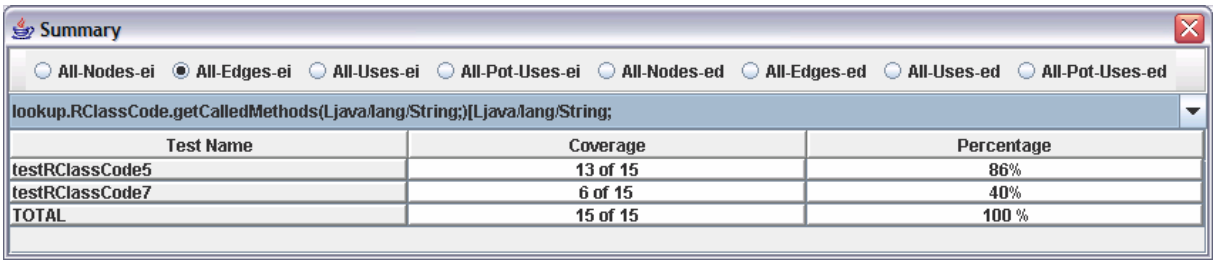


Figura A 7 - Resumo de cobertura do método `String[] getCalledMethods(String assinatura)`

APÊNDICE B

As figuras apresentadas neste apêndice foram geradas pela FATEsC e foram utilizadas para elaborar a Tabela 5.3. Essas figuras apresentam a comparação entre os resultados de cobertura obtidos , por cada caso de teste gerado para uma interface pública do componente segundo o critério Todos-Arcos, pelo desenvolvedor e usuário do componente.

Na primeira coluna são listados os casos de teste elaborados para testar os métodos públicos do componente. Na segunda coluna a quantidade de requisitos cobertos pelos testes. Na terceira coluna a quantidade de requisitos cobertos em relação à segunda coluna pelos casos de teste elaborados para a aplicação. Na última coluna a porcentagem da relação entre as colunas 2 e 3.

Na Figura B1 é mostrado a comparação das coberturas obtidas pelos casos de teste `testProgram2` e `testProgram32` elaborados para testar o método `Program(ZipFile zippedFile, boolean noSys, String toAvoid)`, com as coberturas obtidas pelos casos de teste da aplicação. Na Figura B2 é mostrado a comparação das coberturas obtidas pelos casos de teste `testProgram1` e `testProgram31` elaborados para testar o método `Program(JarFile jarFile, boolean noSys, String toAvoid)`, com as coberturas obtidas pelos casos de teste da aplicação. Na Figura B3 nada foi listados, pois esse método não tem requisitos para o critério selecionado. Na Figura B4 é mostrado o a comparação das coberturas obtidas pelos casos de teste `testProgram10` e `testProgram27` elaborados para testar o método `String[] getCodeClasses(String packName)`, com as coberturas obtidas pelos casos de teste da aplicação. Na Figura B5 é mostrado a comparação das coberturas obtidas pelo caso de teste `testProgram11` elaborado para testar o método `String[] getCodePackages()`, com as coberturas obtidas pelos casos de teste da aplicação. Na Figura B6 é mostrado a comparação das coberturas obtidas pelos casos de `testRClass14` e `testRClass15` elaborados para testar o método `String getClassName(String x)`, com as coberturas obtidas pelos casos de teste da aplicação. Na Figura B7 é mostrado a comparação das coberturas obtidas pelos casos de `testRClassCode5` e `testRClassCode7` elaborados para testar o método `String[]`

`getCalledMethods(String assinatura)`, com as coberturas obtidas pelos casos de teste da aplicação.

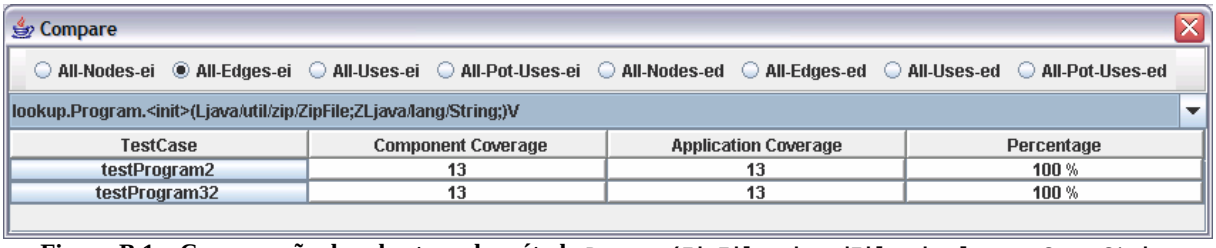


Figura B 1 – Comparação de cobertura do método `Program(ZipFile zippedFile, boolean noSys, String toAvoid)`

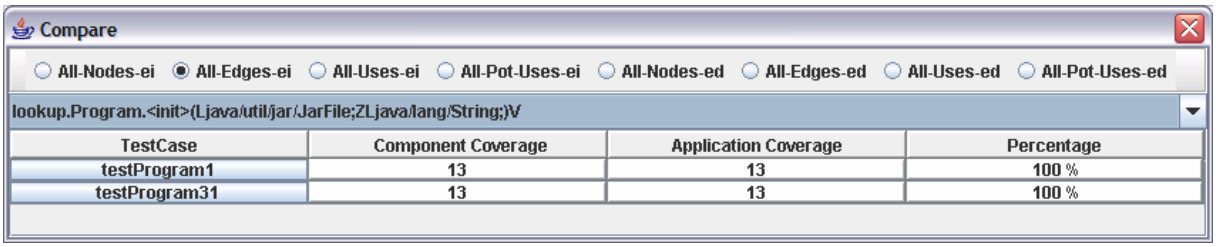


Figura B 2 - Comparação de cobertura do método Program(JarFile jarFile, boolean noSys, String toAvoid)

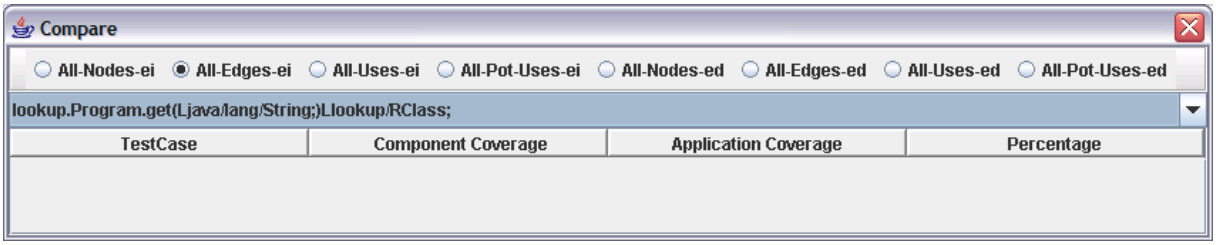


Figura B 3 - Comparação de cobertura do método `RClass get(String s)`

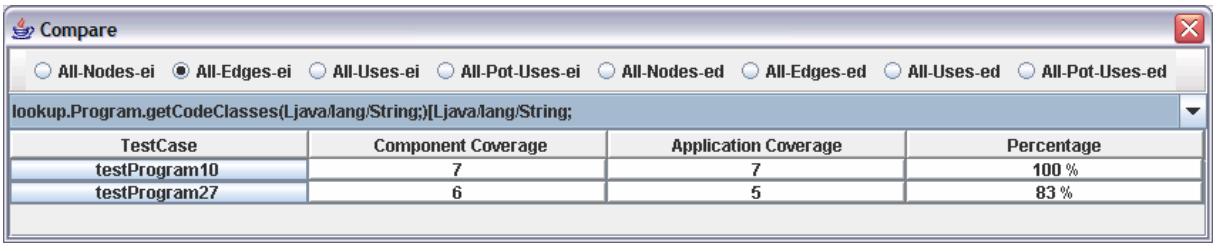


Figura B 4 - Comparação de cobertura do método `String[] getCodeClasses(String packName)`

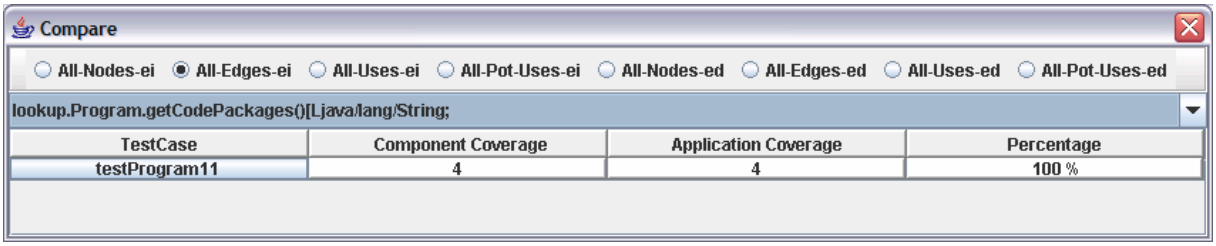


Figura B 5 - Comparação de cobertura do método String[] getCodePackages()

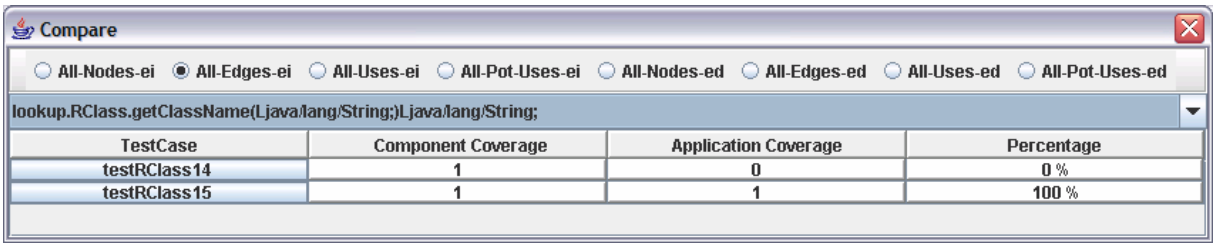


Figura B 6 - Comparação de cobertura do método String getClassName(String x)

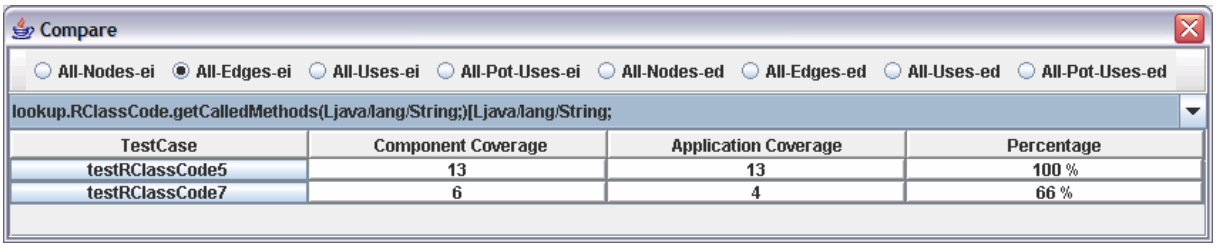


Figura B 7 - Comparação de cobertura do método String[] getCalledMethods(String assinatura)

APÊNDICE C

Nesse apêndice é apresentado o javadoc gerado para o componente utilizado no estudo de caso, o pacote `lookup`. Nesse documento obtém-se informações sobre as classes, métodos, construtores, atributos, argumentos, etc.

Package lookup

Class Summary

ClassClosure	
Program	This class is an abstraction of a program, composed of several .class files.
RClass	This class is used to store information about a given class in a program.
RClassCode	This class is used to store information about a given class in a program.

Class ClassClosure
java.lang.Object
└ **lookup.ClassClosure**

---

public class **ClassClosure**
extends java.lang.Object

Constructor Summary

`ClassClosure()`
`ClassClosure(java.lang.String classPath)`

Method Summary

`java.lang.String[]`	`accessedClasses(JavaClass javaClass)`
`boolean`	`doMatch(java.lang.String x, boolean noSys, java.lang.String toAvoid)`
`java.lang.String`	`findFile(java.lang.String x)`
`java.lang.String[]`	`getClosure(java.lang.String className)`
`java.lang.String[]`	`getClosure(java.lang.String className, boolean noSys, java.lang.String toAvoid)`

	Method responsable to parse the code of a given class name and found a list of all classes related with this one.
`java.lang.String[]`	**`getJCClosure`**`(JavaClass classFile, boolean noSys, java.lang.String toAvoid)`
`java.lang.String`	**`toPoint`**`(java.lang.String s)`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

ClassClosure
public **ClassClosure**(java.lang.String classPath)

ClassClosure
public **ClassClosure**()

Method Detail

getClosure
public java.lang.String[] **getClosure**(java.lang.String className)

---

getClosure
public java.lang.String[] **getClosure**(java.lang.String className,
boolean noSys,
java.lang.String toAvoid)
Method responsable to parse the code of a given class name and found a list of all classes related with this one. In this list will be included neither system class if the variable `noSys` is true nor the ones spaecified in `toAvoid`.
Parameters:
`className` - The name of the starting class. From it all the referenced classes are found and included in the structure. The class should be found in the classpath
`noSys` - This param tells whether "system" classes should be part of the main program structure or just as peripheral classes. If `true`, classes with the following prefix are kept out of the main structure:
java.
javax.lang
org.omg

In addition, any referenced class for wich the code (a .class) file can not be found is considered out of the main structure.
`toAvoid` - This is a string that indicates other classes that should be avoided in the main structure of the program. For example, if the program uses library packages `org.dummy` and `br.din.foo` the use of "org.dummy br.din.foo" as the third argument will keep the classes in these packages out of the program structure, even if their class files can be found in the classpath

---

getJCClosure
public java.lang.String[] **getJCClosure**(JavaClass classFile,
boolean noSys,
java.lang.String toAvoid)

---

toPoint
public java.lang.String **toPoint**(java.lang.String s)

---

accessedClasses

public java.lang.String[] **accessedClasses**(JavaClass javaClass)

---

doMatch
public boolean **doMatch**(java.lang.String x,
boolean noSys,
java.lang.String toAvoid)

---

findFile
public java.lang.String **findFile**(java.lang.String x)

Class Program
java.lang.Object
└ **lookup.Program**
All Implemented Interfaces:
java.io.Serializable

---

public class **Program**
extends java.lang.Object
implements java.io.Serializable
This class is an abstraction of a program, composed of several .class files. The .class files may have subclassing and implementation relationship between them and this is represented in this structure.
On the othar hand, it is considered that other classes like the Java API or other libraries are not of direct interest to this abstraction. So they are represented in the "borders" of the program if one of the classes in the program has something to do with them. For example if the class `MyClass1` extends the `Vector` class, both should appear in the program structure but `MyClass1` appears as a first class object and `Vector` only as an auxiliary object.
The classes of interest are represented by `RClassCode` objects. In opposition, peripheral classes appears as `RClass` objects. For the first a complete description of the class is avaiable, like superclass, interfaces, subclasses, code, fields, etc. For the second only the essential information can be obtained, i.e., the classes that extend or implements it.
See Also:
Serialized Form

Constructor Summary

`Program``(java.util.jar.JarFile jarFile, boolean noSys, java.lang.String toAvoid)` The same of `Program(String, boolean, String, String)` but all the classes in the `JarFile` are included in the structure of the program, as well as the classes they reference..
`Program``(java.lang.String className)` The same of `Program( className, true, null )`
`Program``(java.lang.String className, boolean noSys, java.lang.String toAvoid, java.lang.String classPath)` Creates the structure os a program.
`Program``(java.util.zip.ZipFile zippedFile, boolean noSys, java.lang.String toAvoid)` The same of `Program(String, boolean, String, String)` but all the classes in the `ZipFile` are included in the structure of the program, as well as the classes they reference..

Method Summary

`RClass`	**`get`**`(java.lang.String s)` Gets the `RClass` object for a given class name
`java.lang.String`	**`getClass`**`(int k)`

	Gets the name of the kth class returned by `getClasses()`
`java.lang.String[]`	**`getClasses`**`()` Gets the complete list with the names of the classes
`java.lang.String`	**`getCodeClass`**`(int k)` Gets the name of the kth class returned by `getCodeClasses()`
`java.lang.String[]`	**`getCodeClasses`**`()` Gets the list with the names of the classes in the main structure of this program.
`java.lang.String[]`	**`getCodeClasses`**`(java.lang.String packName)` Gets the list with the names of the classes in the main structure of this program.
`java.lang.String[]`	**`getCodePackages`**`()` Gets the list with the names of the packages in the main structure of this program.
`java.lang.String[]`	**`getSubClassClosure`**`(java.lang.String cl)` Gets the list of subclasses of this one and their subclasses....
`java.lang.String`	**`getSysClass`**`(int k)` Gets the name of the kth class returned by `getSysClasses()`
`java.lang.String[]`	**`getSysClasses`**`()` The opposite of `getCodeClasses()`.
`java.lang.String[]`	**`getSysClasses`**`(java.lang.String packName)` The opposite of `getCodeClasses()`.
`java.lang.String[]`	**`getSysPackages`**`()` Gets the list with the names of the packages not in the main structure of this program.
`int`	**`levelOf`**`(java.lang.String s)` Computes the level (depth) of a class in the hierarchical structure of the program.
`void`	**`print`**`()` Send to the standard output a few informations about this program like: The list of classes and their level The number of code classes The number of "system"classes The classe whith highest depth The classe with highest number of subclasses The interface with highest number of implementations

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Program
public **Program**(java.lang.String className,
boolean noSys,
java.lang.String toAvoid,
java.lang.String classPath)
throws java.lang.ClassNotFoundException,
java.io.FileNotFoundException,
java.io.IOException
Creates the structure os a program. Begining at a given class name, this constructor calculates all the referenced classes and includes in the structure. See the description of the parameters for a more complete explanation
Parameters:

`className` - The name of the starting class. From it all the referenced classes are found and included in the structure. The class should be found in the classpath
`noSys` - This param tells whether "system" classes should be part of the main program structure or just as peripheral classes. If `true`, classes with the following prefix are kept out of the main structure:
java.
javax.lang
org.omg

In addition, any referenced class for wich the code (a .class) file can not be found is considered out of the main structure.
`toAvoid` - This is a string that indicates other classes that should be avoided in the main structure of the program. For example, if the program uses library packages `org.dummy` and `br.din.foo` the use of "org.dummy br.din.foo" as the third argument will keep the classes in these packages out of the program structure, even if their class files can be found in the classpath
Throws:
`java.lang.ClassNotFoundException` - If the root class is not found in the classpath
`java.io.FileNotFoundException` - If the root class file is not found
`java.io.IOException` - Error reading root class file

Program
public **Program**(java.lang.String className)
throws java.lang.ClassNotFoundException,
java.io.FileNotFoundException,
java.io.IOException
The same of
`Program( className, true, null )`
Parameters:
`className` - The name of the starting class
Throws:
`java.lang.ClassNotFoundException`
`java.io.FileNotFoundException`
`java.io.IOException`

Program
public **Program**(java.util.zip.ZipFile zippedFile,
boolean noSys,
java.lang.String toAvoid)
throws java.lang.ClassNotFoundException,
java.io.IOException
The same of `Program(String, boolean, String, String)` but all the classes in the `ZipFile` are included in the structure of the program, as well as the classes they reference..
Throws:
`java.lang.ClassNotFoundException`
`java.io.IOException`

Program
public **Program**(java.util.jar.JarFile jarFile,
boolean noSys,
java.lang.String toAvoid)
throws java.lang.ClassNotFoundException,
java.io.IOException
The same of `Program(String, boolean, String, String)` but all the classes in the `JarFile` are included in the structure of the program, as well as the classes they reference..
Throws:
`java.lang.ClassNotFoundException`
`java.io.IOException`

Method Detail

get

public RClass **get**(java.lang.String s)
Gets the `RClass` object for a given class name
Parameters:
`s` - The name of the class for which the information is required

---

getClasses
public java.lang.String[] **getClasses**()
Gets the complete list with the names of the classes
Returns:
An array of strings representing the names of the classes in the program

---

getClass
public java.lang.String **getClass**(int k)
Gets the name of the kth class returned by `getClasses()`
Parameters:
`k` - - the order of the class to be accessed; 0 based
Returns:
A strings representing the name of the kth class returned by a call to `getClasses()`

---

getCodeClasses
public java.lang.String[] **getCodeClasses**()
Gets the list with the names of the classes in the main structure of this program. Only objects of type `RClassCode` are included in the list. The peripheral classes are not.
Returns:
An array of strings representing the names of the classes in the main program structure

---

getCodeClass
public java.lang.String **getCodeClass**(int k)
Gets the name of the kth class returned by `getCodeClasses()`
Parameters:
`k` - - the order of the class to be accessed; 0 based
Returns:
A strings representing the name of the kth class returned by a call to `getCodeClasses()`

---

getCodeClasses
public java.lang.String[] **getCodeClasses**(java.lang.String packName)
Gets the list with the names of the classes in the main structure of this program. Only objects of type `RClassCode` are included in the list. The peripheral classes are not.
Parameters:
`packName` - A package name used as filter to select the classes
Returns:
An array of strings representing the names of the classes in the main program structure

---

getCodePackages
public java.lang.String[] **getCodePackages**()
Gets the list with the names of the packages in the main structure of this program. Only packages corresponding to objects of type `RClassCode` are included in the list. The peripheral classes are not.
Returns:
An array of strings representing the names of the packages in the main program structure

---

getSysClasses
public java.lang.String[] **getSysClasses**()
The opposite of `getCodeClasses()`. Gets the list with the names of the classes not in the main structure of this program. Only objects of type `RClass` are included in the list.
Returns:
An array of strings representing the names of the classes not in the main program structure

---

getSysClass
public java.lang.String **getSysClass**(int k)

Gets the name of the kth class returned by `getSysClasses()`
Parameters:
`k` - - the order of the class to be accessed; 0 based
Returns:
A strings representing the name of the kth class returned by a call to `getSysClasses()`

---

getSysClasses
public java.lang.String[] **getSysClasses**(java.lang.String packName)
The opposite of `getCodeClasses()`. Gets the list with the names of the classes not in the main structure of this program. Only objects of type `RClass` are included in the list.
Parameters:
`packName` - A package name used as filter to select the classes
Returns:
An array of strings representing the names of the classes not in the main program structure

---

getSysPackages
public java.lang.String[] **getSysPackages**()
Gets the list with the names of the packages not in the main structure of this program. Only packages corresponding to objects of type `RClass` are included in the list.
Returns:
An array of strings representing the names of the packages not in the main program structure

---

levelOf
public int **levelOf**(java.lang.String s)
Computes the level (depth) of a class in the hierarchical structure of the program.
Parameters:
`s` - The name of the class
Returns:
The level of the class in the hierarchical structure. If the class is not in the prgram the value is -1. If it is a peripheral class (type `RClass`) the value is 0. Otherwise the value is `1 + levelOf(its superclass)` .

---

print
public void **print**()
Send to the standard output a few informations about this program like:
The list of classes and their level The number of code classes The number of "system"classes The classe whith highest depth The classe with highest number of subclasses The interface with highest number of implementations

---

getSubClassClosure
public java.lang.String[] **getSubClassClosure**(java.lang.String cl)
Gets the list of subclasses of this one and their subclasses....
Returns:
An array of strings that contains the names of the subclasses. It is never `null`. If no subclass, an array of size 0 is returned.

Class RClassCode
java.lang.Object
  └ lookup.RClass
    └ **lookup.RClassCode**

---

public class **RClassCode**
extends RClass
This class is used to store information about a given class in a program. A program is represented by `Program` object. A `RClassCode` object extends `RClass` and so it stores information about which subclasses extends the class and which classes implements it (if it is an interface).
In addition, it stores complete information about the class itself. For that it uses a `de.fub.bytecode.classfile.JavaClass` object

This class is used in the context of a program to represent those classes that are in the scope of interest. For example, if the program is built over a set of 5 classes
MyClass1
MyClass2
MyClass3
MyClass4
MyClassFive

these classes are represented using this RClassCode class and the relationship about them (subclassing and implementation) are also registered.
See Also:
RClass, Program

Field Summary

Fields inherited from class lookup.RClass
DEFAULT_PACKAGE

Constructor Summary

RClassCode(JavaClass y, java.lang.String x) Creates a RClassCode given its name and a JavaClass object

Method Summary

java.lang.String[]	**getCalledMethods**(java.lang.String assinatura) Retorna uma lista de metodos chamados por um dado metodo desta classe.
java.lang.String[]	**getInterfaces**() Return the list of interfaces this class implements.
java.lang.String	**getSuperClass**() Return the name of the superclass.
JavaClass	**getTheClass**() Return the JavaClass object that represents this class
boolean	**isInterface**() Return whether this class is an interface.
void	**print**() Send a few information to the standard output like Interface or class The name The superclass The interfaces it implements Its subclasses Its implementations

Methods inherited from class lookup.RClass
countImplementations, countSubClasses, getClassName, getImplementations, getName, getPackageName, getPackName, getSubClasses, setImplementation, setSubClass

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RClassCode
public **RClassCode**(JavaClass y,
java.lang.String x)
Creates a RClassCode given its name and a `JavaClass` object
Parameters:
`y` - The `JavaClass` object already created for this class
`x` - The complete name of the class

Method Detail

getSuperClass
public java.lang.String **getSuperClass**()
Return the name of the superclass. The name is extracted from the `theClass` field.
Returns:
The complete name of its superclass

---

getInterfaces
public java.lang.String[] **getInterfaces**()
Return the list of interfaces this class implements. The names are extracted from the `theClass` field.
Returns:
The complete names of its interfaces

---

isInterface
public boolean **isInterface**()
Return whether this class is an interface. The information is extracted from the `theClass` field.
Overrides:
`isInterface` in class `RClass`
Returns:
True if this is an interface

---

getTheClass
public JavaClass **getTheClass**()
Return the `JavaClass` object that represents this class
Returns:
the `JavaClass` object that represents this class

---

getCalledMethods
public java.lang.String[] **getCalledMethods**(java.lang.String assinatura)
Retorna uma lista de metodos chamados por um dado metodo desta classe.
Parameters:
`assinatura` - - a assinatura do mÃ©todo que se deseja analisar
Returns:
a lista de metodos chamados pelo metodo passado como argumento. Se o mÃ©todo solicitado nÃ£o Ã© encontrado na classe, retorna null.

---

print
public void **print**()
Send a few information to the standard output like
Interface or class
The name
The superclass
The interfaces it implements
Its subclasses
Its implementations
Overrides:
`print` in class `RClass`

Class RClass
java.lang.Object
└ **lookup.RClass**
Direct Known Subclasses:
RClassCode

---

public class **RClass**
extends java.lang.Object
This class is used to store information about a given class in a program. A program is represented by `Program` object. A `RClass` object stores information about which subclasses extends the class and which classes implements it (if it is an interface).
This class is used in the context of a program to represent those classes that are out of the scope of interest. For example, if the program is built over a set of 5 classes
MyClass1
MyClass2
MyClass3
MyClass4
MyClassFive

and `MyClass1` extends `Vector` and implements `Comparator` the these two classes are represented with `RClass` objects. Both pointing to `MyClass1` one indicating that it is extended by `MyClass1` and the other indicating that it is implemented by `MyClass1`.
On the other hand, the classes that are part of the program are represented by `RClassCode`.
See Also:
`RClassCode`, `Program`

Field Summary

`static java.lang.String`	**`DEFAULT_PACKAGE`** The name used for the default package

Constructor Summary

`RClass``(java.lang.String x)` Creates an object representing a class.

Method Summary

`int`	**`countImplementations`**`()` Gets the size of the list of implementing classes.
`int`	**`countSubClasses`**`()` Gets the size of the list of subclasses.
`static java.lang.String`	**`getClassName`**`(java.lang.String x)`
`java.lang.String[]`	**`getImplementations`**`()` Gets the list of classes that implement this one.
`java.lang.String`	**`getName`**`()` Gets the name of this class.
`java.lang.String`	**`getPackageName`**`()`
`static java.lang.String`	**`getPackName`**`(java.lang.String x)`
`java.lang.String[]`	**`getSubClasses`**`()`

	Gets the list of subclasses of this one.
`boolean`	**`isInterface`**`()` Return whether this class is an interface.
`void`	**`print`**`()` Sends to standard output some information about this class: Its name Its subclasses Its implementations
`void`	**`setImplementation`**`(java.lang.String s)` Adds a class in the list of classes that implements this one.
`void`	**`setSubClass`**`(java.lang.String s)` Adds a class in the list of classes that extend this one.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

DEFAULT_PACKAGE
public static final java.lang.String **DEFAULT_PACKAGE**
The name used for the default package
See Also:
Constant Field Values

Constructor Detail

RClass
public **RClass**(java.lang.String x)
Creates an object representing a class.
Parameters:
`x` - The name of the class. No check is done, for example whether the class can be found in the current classpath.

Method Detail

getName
public java.lang.String **getName**()
Gets the name of this class.
Returns:
The name of the class, assigned on its creation

---

setSubClass
public void **setSubClass**(java.lang.String s)
Adds a class in the list of classes that extend this one. If the class is already there, it is not inserted again.
Parameters:
`s` - The name of the subclass

---

getSubClasses
public java.lang.String[] **getSubClasses**()
Gets the list of subclasses of this one.
Returns:
An array of strings that contains the names of the subclasses. It is never `null`. If no subclass, an array of size 0 is returned.

---

countSubClasses
public int **countSubClasses**()
Gets the size of the list of subclasses.
Returns:
The number of subclasses of this one

---

setImplementation
public void **setImplementation**(java.lang.String s)
Adds a class in the list of classes that implements this one. If the class is already there, it is not inserted again.
Parameters:
`s` - The name of the implementing class

---

getImplementations
public java.lang.String[] **getImplementations**()
Gets the list of classes that implement this one.
Returns:
An array of strings that contains the names of the implementing classes. It is never `null`. If no subclass, an array of size 0 is returned.

---

countImplementations
public int **countImplementations**()
Gets the size of the list of implementing classes.
Returns:
The number of classes that implement this one

---

isInterface
public boolean **isInterface**()
Return whether this class is an interface. The information calculated based on the `countImplementations()`. If it is 0 it is assumed the object does not refer to an interface
Returns:
True if `countImplementations()` > 0.

---

print
public void **print**()
Sends to standard output some information about this class:
Its name
Its subclasses
Its implementations

---

getPackName
public static java.lang.String **getPackName**(java.lang.String x)

---

getClassName
public static java.lang.String **getClassName**(java.lang.String x)

---

getPackageName
public java.lang.String **getPackageName**()
