Uma ferramenta para auxiliar o teste de programas Java

Nelson Rodrigo Lombardi Bassetto

Monografia apresentada à
Universidade Cidade de São Paulo
para a obtenção da graduação em
Ciência da Computação

Área de concentração: Engenharia de Software
Orientador: Prof. Msc. Leandro César Prudente

São Paulo, Novembro de 2005

Folha de aprovação

Conceito obtido

Leandro César Prudente

Nome e assinatura do participante (1) da banca julgadora

Nome e assinatura do participante (2) da banca julgadora

Data : ____/____/______

À minha família.

Agradecimentos

Agradeço ao Prof. Msc. Leandro César Prudente por sua competente e atenciosa orientação, a qual foi essencial no desenvolvimento deste trabalho e da ferramenta GetUseDef;

Ao Paulo Roberto de A. F. Nunes, por sua atenção e apoio nas dúvidas quanto ao Recoder;

A Universidade Cidade de São Paulo por me proporcionar a oportunidade de desenvolver um trabalho com tamanha significância ao meu desenvolvimento;

Aos meus amigos pelo apoio e compreensão nos momentos de ausência;

Aos meus familiares por me fornecerem bases sólidas que servem de apoio ao meu desenvolvimento.

Resumo

Este trabalho apresenta uma ferramenta desenvolvida para o auxílio em atividades de teste de programas Java. Para tanto, a ferramenta baseia-se em técnica de teste estrutural e no critério baseado em fluxo de objetos, todos os pares definição-uso. O objetivo é automatizar a aplicação do critério todos os pares definição-uso, que, quando aplicado manualmente, pode ser uma atividade dificultosa, demorada e sujeita a erros. A ferramenta desenvolvida automatiza o primeiro passo deste critério, identificar as definições e usos sobre os objetos. Para isto, recebe como entrada arquivos fonte de um programa Java e com base nestes gera uma lista de definições e usos sobre objetos. Com base nesta lista, a pessoa responsável pela atividade de teste poderá dar continuidade à sua atividade de teste com mais confiança e finalizá-la em menor tempo.

Abstract

The present work explores a tool developed to assist in Java program testing activities. The tool is based on structural testing techniques and the object-flow criterion, covering all definition-use pairs. The goal is to automate the application of the all definition-use pairs criterion, which can be a challenging, time-consuming, and error-prone manual activity. The developed tool automates the first step of this criterion, identifying the definitions and uses of objects. It takes Java program source files as input and generates a list of object definitions and uses. Based on this list, the tester can continue their testing activity with more confidence, taking less time for completion.

Conteúdo

1. Introdução 9
   1.1. Contexto 9
   1.2. Motivação e Objetivos 11
   1.3. Organização 11
2. Teste de Software 12
   2.1. Idéia Básica 13
   2.2. Objetivos do Teste 14
   2.3. Técnicas e Critérios de Teste 15
      2.3.1. Critério de teste para sistemas orientados a objetos 18
      2.3.2. Critérios baseados em fluxo de objetos 19
      Critério de todos os pares de definição-uso 21
3. Uma ferramenta para auxiliar o teste de programas Java 24
   3.1. Visão Inicial 24
   3.2. Arquitetura da ferramenta GetUseDef 25
      Código fonte do programa Java 26
      (UI) *User Interface* (interface com o usuário) 26
      Recoder 29
      Analisador Java (fase 1) 29
      Conjunto parcial definições-usos 29
      Analisador Java (fase 2) 29
      Conjunto completo definições-usos 29
      Extrator resultados 30
      Arquivo com resultados formatados 30
      3.2.1. Relacionamento entre os componentes da ferramenta 31
   3.3. Um exemplo de uso da ferramenta GetUseDef 34
4. Conclusões e trabalhos futuros 37
   4.1. Conclusão 37
      4.1.1. Resultados obtidos 37
   4.2. Trabalhos Futuros 39
      Aprimoramentos nos algoritmos da ferramenta 39

Alteração no *lay-out* do arquivo de saída da ferramenta 39
Implementação da próxima fase do critério de todos os pares du............. 40
Estudos para metrificar os benefícios do uso da ferramenta GetUseDef ... 40

Índice de Figuras

Figura 2-1 - Diagrama de Fluxo de Dados do Processo de Teste. Fonte: [9] 14
Figura 2-2 - Classe de exemplo do critério Todos pares du (1)............................. 22
Figura 2-3 - Classe de exemplo do critério Todos pares du (2)............................. 22
Figura 3-1 – Arquitetura da ferramenta *GetUseDef*.. **Erro! Indicador não definido.**
Figura 3-2 - Tela Inicial da ferramenta *GetUseDef*... 26
Figura 3-3 - Seleção do projeto a ser analisado.. 27
Figura 3-4 - Seleção do arquivo de saída 28
Figura 3-5 - Projeto analisado com sucesso. 28
Figura 3-6 - Arquivo de saída com resultados... 30
Figura 3-7 – *Lay-out* do arquivo de saída... 31
Figura 3-8 – *Lay-out* do arquivo de saída, uma classe sem pacote 31
Figura 3-9 – Classes analisadas pela ferramenta GetUseDef............................... 34
Figura 3-10 – Arquivo resultado da ferramenta GetUseDef 35

1. Introdução

1.1. Contexto

Com o avanço constante da tecnologia, os computadores estão mais acessíveis e como conseqüência, aparecem cada vez mais em maior número nas organizações. As principais funções de um computador em uma organização são: otimização, aumento de desempenho e produtividade dos processos organizacionais. Com este aumento de demanda por *hardware* também surge o aumento de demanda por *softwares* cada vez mais robustos e confiáveis. Como grande parte do desenvolvimento de um *software* é realizado por processo humano, o resultado final está sujeito a falhas[1], as quais, dependendo do caso podem gerar grandes prejuízos.

Com a finalidade de se aumentar a confiabilidade e a qualidade de um *software,* existe uma área da Engenharia de *Software*, o teste. Teste de *software* tem por objetivo identificar e corrigir erros de programação[2], especificação e projeto antes que o *software* seja posto em ambiente de produção, consumindo custos de tempo e de esforço mínimos. Segundo Pressman [1] “A atividade de teste não pode mostrar a ausência de defeitos[3], ela só pode mostrar se defeitos de *software* estão presentes”. Conforme a definição de Myers [2] “Teste é o processo de executar o programa com o objetivo de encontrar defeitos”.

Com o objetivo de reduzir custo despendido na atividade de teste são utilizadas técnicas de teste. As técnicas de teste dão uma abordagem sistemática ao teste e aumentam a probabilidade de descobrir erros no *software*. Conforme define Leandro César Prudente [8] “um dos objetivos das técnicas sistemáticas é

---

[1] Uma falha (“*failture*”) acontece quando uma saída incorreta é produzida com relação à especificação do programa [7].

[2] Um erro de programação (“*error*”) é uma diferença existente entre o valor correto e o valor esperado para alguma variável na execução do programa [7].

[3] Defeito (“*bug*”) é uma deficiência no programa que pode provocar uma saída incorreta para algum dado de entrada [7].

garantir que aspectos relevantes da especificação e da implementação do sistema sejam executados (exercitados) pelo menos uma vez por algum caso de teste”. Este trabalho aborda, dentre outras técnicas existentes, a técnica estrutural, também conhecida como teste de caixa-branca. Esta técnica de teste utiliza o código fonte para identificar requisitos a serem cumpridos pelos casos de teste [8].

Testar todas as combinações possíveis de valores de entrada para um programa, mesmo sendo um programa pequeno, é na maioria das vezes inviável ou impossível, pois o número de combinações de valores de entrada possível é muito grande. Para ajudar o *testador*[4] a selecionar casos de teste e a analisar o nível de cobertura[5] dos casos de teste existentes são elaborados os critérios de teste [8]. Um critério de teste pode ser entendido como um método sistemático de se testar um programa.

Existem diversos critérios de teste, dentre eles, os critérios baseados no fluxo de objetos são desenvolvidos para o teste de programas orientados a objetos e tem sido bastante utilizado pelos *testadores* [8]. Dentre os critérios baseados no fluxo de objetos, o critério todos os pares definição-uso (du), proposto por Chen e Kao [3], será aplicado neste trabalho. No critério de todos os pares du, todas as definições de objetos (situação que altera o estado do objeto) e usos (situação em que o objeto ou um de seus atributos é referenciado) são apurados, com base nestes são gerados os pares du que devem ser exercitados por algum caso de teste.

Com o objetivo de automatizar o primeiro processo do critério de todos os pares du, onde são apurados os objetos e as definições e usos sobre eles, foi desenvolvida a ferramenta *GetUseDef*, cujo nome significa obter usos e definições. Esta ferramenta analisa programas escritos na linguagem Java, trazendo como resultado uma listagem de definições e usos de objetos do programa. Com base nestes resultados o *testador* poderá identificar todos os pares du e testá-los.

---

[4] Pessoa que executa a atividade de teste.

[5] Porcentagem de requisitos que são satisfeitos por um critério.

1.2. Motivação e Objetivos

O processo de teste de *software* tende a aumentar a qualidade e confiabilidade do *software*, no entanto, por demandar tempo de desenvolvimento e necessitar de recursos humanos capacitados para aplicá-lo, pode causar impacto no custo final e no prazo de entrega do *software*. Segundo Pressman [1] "a atividade de teste consome tipicamente entre 40% e 50% do esforço total no desenvolvimento de *software"*. A rigor, o teste de *software* que envolve vidas (p. ex., controle de vôo, monitoramento de reatores nucleares) pode custar de três a cinco vezes mais do que todos os outros passos de engenharia de *software* combinados [5]. Para reduzir os impactos deste problema é essencial o uso de ferramentas que auxiliem o *testador* em sua atividade de teste.

Com base nestas considerações foram definidos os objetivos deste trabalho:

- Realizar estudos sobre a automação do teste de *software*;
- Definir os requisitos, desenvolver e implementar uma ferramenta que auxilie o *testador* a identificar todos os pares de definição-uso (du), em sua tarefa de teste do *software*;
- Validar a ferramenta através de experimentos e análise de casos

1.3. Organização

O **Capítulo 2** discute conceitos relacionados ao teste de *software*. São enfatizados técnicas e critérios de teste para sistemas orientados a objetos. O critério todos os pares definição-uso, usado como base na ferramenta, é explicado e exemplificado.

O **Capítulo 3** descreve a solução desenvolvida. De início, é apresentada uma visão inicial da ferramenta. Em seguida a arquitetura de componentes é discutida e detalhada. Por fim, é apresentado um exemplo de uso da ferramenta desenvolvida.

O **Capítulo 4** apresenta as conclusões deste trabalho e indica possíveis extensões.

2. Teste de Software

Como grande parte do desenvolvimento de um *software* é realizado por processo humano, o resultado final está sujeito à falhas. Neste mesmo sentido, Pressman [5] cita Deutsch [16]:

> "O desenvolvimento de sistema de *software* envolve uma série de atividades de produção em que as oportunidades para injetar a falibilidade humana são enormes. Erros podem vir a ocorrer até no início do processo, onde os objetivos podem ser errônea ou imperfeitamente especificados, bem como [em] estágios posteriores de projeto e desenvolvimento... Por causa da inabilidade humana de se realizar e de se comunicar com perfeição, o desenvolvimento é acompanhado por uma atividade de garantia de qualidade."

O Teste de *Software* é um dos processos aplicados no desenvolvimento de um *software*, visa garantir a qualidade do *software,* revisando a especificação, projeto e código [5]. Conforme define Pressman [1] "o Teste de *Software* é um procedimento crítico para a qualidade do *software*, representando a última atividade na qual é possível revelar erros na especificação, no projeto e na codificação do *software"*. Uma vez gerado o código fonte [5], o software deve ser testado para descobrir (e corrigir) tantos erros quanto possível antes de ser entregue ao seu cliente.

O crescente aumento da demanda por *softwares,* onde o custo de atendimento associado a uma falha pode ser enorme, em algumas situações com prejuízos irreversíveis, são forças motivadoras para uma atividade de teste rigorosa e bem planejada [5].

O objetivo deste capítulo é introduzir conceitos relacionados à atividade de teste de sistemas orientados a objetos dentro do contexto do trabalho desenvolvido.

2.1. Idéia Básica

Conforme a definição de Paulo Marcos Bueno [6] a idéia básica do teste é:

> "executar um programa, fornecendo dados de entrada (referidos como dados de teste) e comparar os valores de saída produzidos com o resultado esperado, segundo a especificação do programa. Caso a saída alcançada seja diferente do resultado esperado tem-se uma falha do programa. A causa desta discrepância deve ser identificada e corrigida no processo de depuração[6]."

Por definição, um dado de teste é um conjunto de dados de entrada planejados para o programa em teste, já um caso de teste refere-se a um dado de teste associado a um resultado esperado após determinado processamento. Podemos entender um dado de teste como os valores lançados em um programa com o objetivo de testá-lo. Um caso de teste pode ser entendido como um procedimento onde são conhecidos os dados de teste e o resultado que o programa deve obter após a execução de um determinado processamento.

A Figura 2-1 é um diagrama de fluxo de dados do processo de teste proposto por Ian Sommerville [9], este diagrama ilustra a idéia básica da atividade de teste onde: casos de testes são projetados, com base nestes casos de testes são preparados os dados de testes, em seguida o programa é executado com estes dados de testes onde é verificada a existência de erros comparando-se os resultados obtidos pelo programa com os resultados esperados nos casos de testes, com base nestes resultados, relatórios podem ser gerados e consequentemente os erros podem ser corrigidos.

---

[6] "A depuração não é teste. Na depuração busca-se relacionar sintomas observados no teste (saídas incorretas do programa) com as suas causas (defeitos no programa). O objetivo é a identificação e correção destes defeitos" [6]

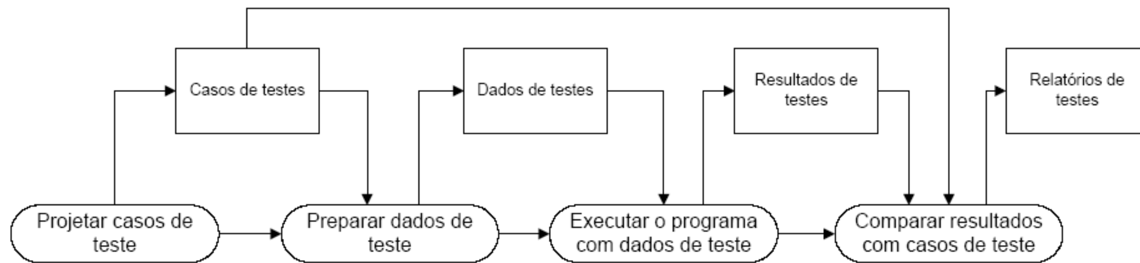


Figura 2-1 - Diagrama de Fluxo de Dados do Processo de Teste. Fonte: [9]

2.2. Objetivos do Teste

Segundo Pressman [1] "a atividade de teste consome tipicamente entre 40% e 50% do esforço total no desenvolvimento de *software"*. Este dado pode ser considerado relativo ao se ponderar que as práticas adotadas pelo *desenvolvedor*[7] ou organização que desenvolve o *software* variam de acordo com suas políticas adotadas e seus objetivos. Pode se objetivar um menor tempo e custo possível no desenvolvimento do *software,* deixando o processo de teste em segundo plano ou mesmo de lado. No entanto, para garantir a qualidade e confiabilidade de um *software* é indispensável que o processo de teste seja executado de maneira sistemática e eficaz. Segundo Hetzel [10] teste é "o processo de estabelecimento de confiança de que um programa ou sistema faz o que sopõe-se que ele faça".

Como foi visto, existem casos em que responsáveis pelo desenvolvimento de um *software* atribuem pouca ou nenhuma prioridade para a atividade de teste, no entanto existem sistemas em que a atividade de teste é essencial e indispensável, por exemplo, sistemas que envolvem vidas (controle de vôo, monitoramento de reatores nucleares e outros do gênero). Pressman [5] afirma que o teste deste tipo de sistema pode custar de três a cinco vezes mais do que todos os outros passos de engenharia de *software* combinados.

---

[7] Desenvolvedor é a pessoa que desenvolve e codifica o software, também conhecido como programador.

Com base nestes dados podemos dividir os objetivos do teste em dois segmentos: o teste de *software* com o objetivo de se garantir a qualidade do *software* e o teste de *software* com o objetivo de se eliminar a maior quantidade de erros possíveis tentando garantir a integridade do *software*. Neste mesmo sentido Paulo Marcos Bueno [6] afirma que:

> "Existem dois enfoques essenciais e complementares: o teste como uma atividade de validação das funcionalidades que o *software* deve apresentar, procurando demonstrar o comportamento correto – chamado de "*clean test*" [11]; e o teste feito com o propósito de "quebrar" o *software*, chamado de "*dirty test"*. Segundo Myers [2] e Graham [12], considerar o primeiro enfoque apenas tente a resultar em um teste de baixa qualidade. Isto é, além de verificar se o *software* faz o que é esperado que ele faça é necessário também verificar se este *software* não faz – erroneamente – o que não é esperado que ele faça."

Pressman [5] cita em seu livro algumas regras enumeradas por Glen Myers [2] que considera servirem como objetivos do teste:

1. "Teste é um processo de execução de um programa com a finalidade de encontrar um erro."
2. "Um bom caso de teste é aquele que tem alta probabilidade de encontrar um erro ainda não descoberto".
3. "Um teste bem-sucedido é aquele que descobre um erro ainda não descoberto".

2.3. Técnicas e Critérios de Teste

Conforme foi visto, os principais objetivos do teste de *software* são garantir a qualidade e integridade do software utilizando-se de custos de tempo e esforços mínimos. Tentar testar [6] todas as combinações possíveis de valores de entrada para um programa é praticamente impossível, pois o número de combinações tende a ser exorbitante e consumiria-se muito tempo para serem apurados e testados. No mesmo sentido, tentar testar todos os possíveis caminhos lógicos de

um programa também é praticamente impossível. Pressman [5] cita um exemplo desta impossibilidade:

> "(...) considere um programa de 100 linhas em linguagem C. Depois de algumas declarações básicas de dados, o programa contém dois ciclos aninhados, que executam de 1 a 20 vezes cada um, dependendo das condições especificadas na entrada. Dentro do ciclo interior, quatro construções se-então-senão são necessárias. Há aproximadamente 10^{14} caminhos possíveis que podem ser executados neste programa!
>
> Para colocar este número em perspectiva, consideramos que um processador de teste mágico ("mágico" porque não existe tal processador) tenha sido desenvolvido para teste completo. O processador pode desenvolver um caso de teste, executá-lo e avaliar os resultados em um milissegundo. Trabalhando 24 horas por dia, 365 dias por ano, o processador iria trabalhar durante 3.170 anos para testar o programa. Isto iria, inegavelmente, tumultuar a maioria dos cronogramas de desenvolvimento. Teste completo é impossível para grandes sistemas de *software*."

Por este motivo surge a necessidade de utilizar técnicas que orientem uma forma mais produtiva e eficaz na atividade de teste. As técnicas de teste de *software* visam oferecer uma abordagem sistemática na atividade de teste, tornando-o mais seguro, produtivo e com melhor qualidade. Conforme a definição de Leandro César Prudente [8] "Um dos objetivos das técnicas sistemáticas é garantir que aspectos relevantes da especificação e da implementação do sistema sejam executados (exercitados) pelo menos uma vez por algum caso de teste". Existem diversas técnicas de teste, dentre elas, duas são bastante utilizadas [8]:

1. Teste funcional ou comportamental (teste caixa-preta): examina-se o programa sem a necessidade de se saber como este foi implementado. O programa é uma espécie de "caixa-preta" onde entram informações e saem resultados que são validados de acordo com as especificações do programa. Segundo Pressman [5] "(...) o teste de caixa-preta permite ao engenheiro de *software* derivar conjuntos de condições de entrada que vão exercitar plenamente todos os requisitos funcionais de um programa.". Este teste tem por objetivo identificar erros das seguintes categorias [5]: funções incorretas ou omitidas, erros de interface, erros

de estrutura de dados ou de acesso a base de dados externa, erros de comportamento ou desempenho e erros de iniciação e término.

2. Teste estrutural (teste caixa-branca ou caixa de vidro): o código fonte em si é analisado com o objetivo de se identificar requisitos a serem cumpridos pelos casos de teste [8]. Conforme Pressman [5], usando teste de caixa-branca o engenheiro de *software* pode derivar casos de teste que: garantam que todos os caminhos independentes, de um módulo, tenham sido exercitados pelo menos uma vez, exercitam todas as decisões lógicas em seus lados verdadeiro e falso, executam todos os ciclos nos seus limites e dentro de seus intervalos operacionais, e exercitam as estruturas de dados internas para garantir sua validade.

Conhecendo-se estas duas técnicas de teste de *software* pode surgir a seguinte questão: "Por que gastar tempo se preocupando com os detalhes de implementação fazendo o teste estrutural ao invés de utilizar-se somente do teste funcional garantindo que os requisitos do programa foram satisfeitos?" Uma das respostas para esta questão segundo Pressman [5] é a seguinte:

> "Erros lógicos e pressupostos incorretos são inversamente proporcionais à probabilidade de que um caminho de programa vai ser executado. Os erros tentem a penetrar no nosso trabalho quando projetamos e implementamos funções, condições ou controle que estão fora da função principal. Um processamento cotidiano tende a ser bem entendido (e bem examinado), enquanto um processamento 'de casos especiais' tende a cair pelas frestas."

Com base nestas informações podemos concluir que tanto o teste funcional quanto o teste estrutural são essenciais para garantir a qualidade e a integridade do *software*. Reforçando esta conclusão, Paulo Marcos Bueno [6] afirma que:

> "As diversas técnicas não são excludentes. Isto é, não se deve utilizar uma ou outra técnica: elas são complementares, pois cada uma delas testa o *software* sob uma perspectiva diferente. Geralmente, uma técnica proporciona a descoberta de classes de defeitos diferentes das que as outras proporcionam."

Associados às técnicas, *Critérios de Teste* têm sido elaborados com o objetivo de se aumentar a probabilidade de revelação de defeitos. Um critério de teste pode ser entendido como um método sistemático de se testar um programa. Este método pode estabelecer passos a serem seguidos e requisitos que devem ser satisfeitos para que se conclua a atividade de teste. Conforme a definição de Paulo Marcos Bueno [6]: "os Critérios de Teste estabelecem requisitos a serem satisfeitos[8] e que podem orientar: a seleção de dados de teste; a avaliação da qualidade do teste e a definição de requisitos de suficiência a serem atingidos para o encerramento desta atividade".

No contexto deste trabalho são de interesse os critérios de teste relacionados à técnica estrutural. Na seção seguinte são apresentadas definições básicas essenciais ao entendimento deste trabalho.

2.3.1. Critério de teste para sistemas orientados a objetos

Segundo Sommerville [9] os sistemas orientados a objetos diferem dos sistemas orientados a funções, de duas maneiras fundamentais:

1. Nos sistemas orientados a funções, existe uma distinção bastante nítida entre as unidades básicas de programas (funções) e os conjuntos dessas unidades de programas (módulos). Em sistemas orientados a objetos, não há tal distinção. Os objetos podem ser simples entidades como uma lista, ou entidades complexas que incluem uma série de outros objetos.

2. Frequentemente, não existe uma hierarquia de objetos bem definida, como é comum nos sistemas orientados a funções.

Devido a estas diferenças critérios de teste estrutural que foram desenvolvidos para sistemas orientados a funções geralmente não são aplicáveis a sistemas orientados a objetos. Segundo Leandro César Prudente [8]:

---

[8] Satisfazer um critério de teste, de um modo genérico, significa satisfazer todos os requisitos de teste estabelecidos pelo critério [6].

> “Tanto no teste procedimental quanto no orientado a objetos, os critérios de teste têm sua importância na atividade de teste. Entretanto, foram observados problemas no teste OO quando aplicados critérios de teste procedimentais: os critérios tradicionais não são suficientes para revelar falhas porque as aplicações OO não são executadas sequencialmente. McDermid, Clack e Kim [13] observam isso nos critérios baseados em fluxo de controle e nos critérios baseados em fluxo de dados aplicados a programas OO.”

Surge então a necessidade de se estudar uma forma de testar apropriadamente sistemas orientados a objetos. Os critérios baseados em fluxo de objetos são propostos para testes em sistemas orientados a objetos e têm apresentado um bom nível de cobertura e descoberta de falhas [8].

2.3.2. Critérios baseados em fluxo de objetos

Foram enumeradas algumas definições que são necessárias para o entendimento do funcionamento dos critérios baseados em fluxo de objetos. Segundo Chen e Kao [3]:

1) **Objeto**: Um objeto é uma instancia de uma classe realizada em tempo de execução. Este objeto tem uma coleção de atributos e métodos;

2) **Estado de um Objeto**: é uma combinação de intervalos de valores de atributos definidos no objeto.

3) **Objeto Definido:** A definição de um objeto ocorre quando seu estado é iniciado ou alterado, isso acontece quando alguma das condições abaixo ocorrem:

   a) o construtor do objeto é invocado;
   b) um atributo é definido, ou seja, o valor do atributo é atribuído explicitamente;
   c) um método que inicia ou modifica os atributos do objeto é invocado.

4) **Objeto usado:** um objeto é usado se uma das seguintes condições ocorrerem:

   a) um de seus atributos é usado numa computação ou num predicado;
   b) um de seus métodos que usa o atributo do objeto é invocado;
   c) o objeto é passado como um parâmetro de um método.

5) **Concretizações:** um objeto pode ser ligado a diferentes classes em tempo de execução numa substituição polimórfica. Em Java, por exemplo, uma classe abstrata “Abs1” pode ser herdada por outras classes “Cl1” e “Cl2”. Um objeto pode ser definido com o tipo “Abs1” e posteriormente ser instanciado tanto como “Cl1” como “Cl2”. Isto é o que caracteriza uma concretização, ou seja, quando o objeto do tipo abstrato é instanciado como um dos possíveis tipos que herdam este tipo abstrato.

Chen e Kao [3] propõem dois critérios baseados no fluxo de objetos, que foram projetados especificamente para sistemas orientados a objetos: “(...) nós desenvolvemos uma estratégia baseada no fluxo de objetos em que duas novas coberturas, critério de todas as concretizações e todos os pares du, são propostas para se manter a par do comportamento dinâmico de programas orientados a objetos.”.

- **Todas as Concretizações (all-binding):** todas as possíveis concretizações de cada objeto devem ser exercitadas pelo menos uma vez quando o objeto for definido ou usado. Se um comando invocar múltiplos objetos então cada combinação de possíveis concretizações precisam ser testadas pelo menos uma vez. Tem como objetivo [8] resolver casos de herança e polimorfismo, podendo ajudar a identificar que métodos herdados que necessitam ser reexaminados.

- **Todos os pares definição-uso (all-du-pairs):** pelo menos um caminho livre de definição de cada objeto definido para cada objeto usado desta definição deve ser exercitado sobre algum teste. É aplicado para [8] monitorar o comportamento de cada um dos objetos durante seu período de vida, guiando o caminho de onde o objeto é definido e onde cada definição é referenciada.

Conforme estudos de caso realizados por Chen e Kao [3], o critério todos os pares definição-uso apresenta um grande percentual de descoberta de falhas. Na seção seguinte são apresentadas definições básicas essenciais ao entendimento deste critério.

Critério de todos os pares de definição-uso

Conforme foi visto, aplicar um teste estrutural que analise o código fonte do programa em teste por completo, além de inviável, é praticamente impossível. Existem diversas técnicas de teste, no entanto para programas orientados a objetos é necessário que se utilizem técnicas específicas para as características deste paradigma. O critério de todos os pares definição-uso é um critério baseado no fluxo de objetos, ou seja, desenvolvido especificamente para testes em programas orientados a objetos e vem apresentando bons resultados na revelação de falhas nos sistemas [3,8].

Este critério propõe que pares de definições e usos de objetos sejam apurados e que estes sejam testados por algum caso de teste, desta forma trazendo como resultado uma seleção caminhos no código fonte que devem testados.

```
1: public class ClasseBasicaExemplo2 {
2:
3:      public int x;
4:      public int y;
5:
6: }
```

Figura 2-2 - Classe de exemplo do critério Todos pares du (1)

```
1: public class ClasseBasicaExemplo1 {
2:
3:      ClasseBasicaExemplo1() {
4:              int x;
5:              ClasseBasicaExemplo2 exemplo = new ClasseBasicaExemplo2();
6:              .
7:              .
8:              ....Outras Operações....
9:              .
10:             .
11:             x = exemplo.x;
12:             exemplo.x = 250;
13:             .
14:             .
15:             ....Outras Operações....
16:             .
17:             .
18:             x = exemplo.x;
19:             .
20:             .
21:             ....Outras Operações....
22:             .
23:             .
24:     }
25:
26: }
```

Figura 2-3 - Classe de exemplo do critério Todos pares du (2)

As Figuras 2-2 e 2-3 ilustram a implementação de um programa simples, escrito na linguagem de programação Java, com o objetivo de se exemplificar o critério todos os pares definição-uso. A Figura 2-2 ilustra a implementação de uma classe denominada ClasseBasicaExemplo2 onde são definidos dois atributos do tipo inteiro, x e y. Já a Figura 2-3 ilustra a classe denominada ClasseBasicaExemplo1, onde, na implementação do seu construtor, o objeto exemplo do tipo ClasseBasicaExemplo2 é instanciado na linha 5, tem seu atributo x usado na linha 11, definido na linha 12 e, por fim, usado novamente na linha 18. Considerando-se que onde consta a marcação "....Outras Operações...." são realizadas operações quaisquer, que não envolvem o objeto exemplo, temos a seguinte situação:

- O objeto exemplo é definido na linha 5, segundo a definição 3.a;
- O objeto exemplo é definido na linha 12, segundo a definição 3.b;
- O objeto exemplo é usado na linha 11 e 18, segundo a definição 4.a.

Após a identificação das definições e usos pode-se apurar os pares definição-uso. Um par definição-uso pode ser notado pela expressão (n1, n2), onde, n1

representa o número da linha no código fonte onde ocorre uma definição do objeto e n2 representa o número da linha no código fonte onde ocorre um uso do mesmo objeto.

O primeiro passo no processo da apuração dos pares definição-uso é a identificação dos objetos e onde estes são definidos e usados. Uma das possíveis maneiras de se apurar todos os pares definição-uso a partir do conjunto de definições e do conjunto de usos é fazer todas as possíveis combinações entre estes.

Conforme o exemplo citado, temos o objeto “exemplo” e o conjunto de definições Def (5,12) formado pelo número das linhas onde o objeto é definido e do conjunto de usos Use (11,18) formado pelo número das linhas onde o objeto é usado. Para apurar os pares definição-uso deste exemplo podemos fazer todas as possíveis combinações na forma Def [X] Use (Def produto cartesiano Use), tendo como resultado os seguintes pares definição-uso: (5,11), (5,18), (12,11) e (12,18).

A partir de todos os pares, o próximo passo é eliminar os que nunca serão executados. Neste exemplo, devemos eliminar os pares (5,18), pois existe uma definição entre eles, e (12,11), pois o uso está acima da definição. Sobram então os pares (5,11) e (12,18).

Conforme a definição do critério, os pares du apurados devem ser exercitados por algum caso de teste. No exemplo basta a chamada do construtor da classe ClasseBasicaExemplo1 para exercitar os dois pares definição-uso. No entanto, existirão casos em que será necessário definir casos de teste que levem o fluxo de execução do programa ao par definição-uso.

O capítulo seguinte apresenta a solução desenvolvida para se automatizar o primeiro passo do critério de todos os pares definição-uso.

3. Uma ferramenta para auxiliar o teste de programas Java

3.1. Visão Inicial

Visando a redução do tempo e do custo do processo de teste de *software* foi desenvolvida a ferramenta GetUseDef. A ferramenta GetUseDef tem como principal função auxiliar o *testador* em sua atividade de teste, reduzindo o esforço despendido por este e aumentando a qualidade do seu trabalho.

A ferramenta utiliza como base o critério de todos os pares du [3], descrito no capítulo 2. Em sua primeira versão, a ferramenta GetUseDef automatiza o primeiro processo deste critério, analisando um código fonte escrito na linguagem de programação Java e extraindo deste o número das linhas onde ocorrem definições e usos de objetos. A obtenção dos pares de definição e uso ainda não está automatizada.

Com base no resultado obtido pela ferramenta, o *testador* pode seguir os próximos passos deste critério para obter todos os pares de definição e uso e, em seguida exercitá-los. Para exercitar, ou seja, testar os pares de definição e uso que foram encontrados é necessário estipular dados de entrada que levem o fluxo de execução do programa a determinados pares de definição e uso, este trabalho é atribuído ao *testador*.

Além disso, foi desenvolvida uma interface gráfica que permite ao usuário selecionar o projeto que será analisado, informar o arquivo de saída onde serão gravados os resultados, dar inicio ao processamento e acompanhar o progresso deste. O resultado gerado pela ferramenta é um arquivo em formato texto onde são exibidas todas as definições e usos encontrados no código fonte analisado, agrupados por Pacote, Classe e Método.

Nas próximas seções são detalhados os componentes da ferramenta e é mostrado um exemplo de uso.

3.2. *Arquitetura da ferramenta GetUseDef*

A ferramenta GetUseDef é composta basicamente por três componentes, uma estrutura de dados e um pacote de classes externo. A seguir são descritos os principais componentes da ferramenta, o fluxo de informações entre eles e suas entradas e saídas, conforme pode ser observado na Figura 3-1.

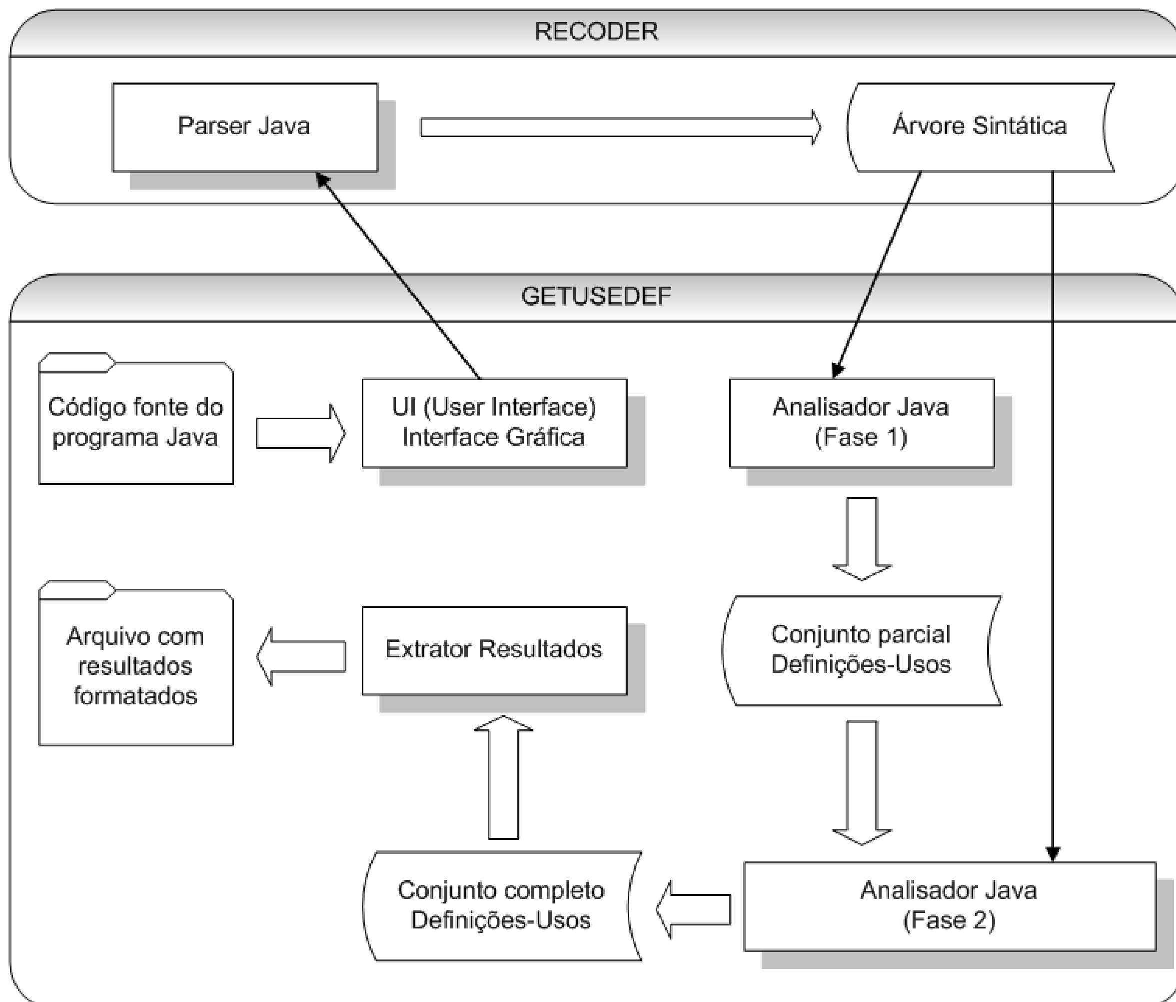


Figura 3-1 – Arquitetura da ferramenta GetUseDef

Código fonte do programa Java

É composto por um ou mais arquivos que compõem um programa Java, que são analisados pela ferramenta.

(UI) *User Interface* (interface com o usuário)

É o meio de interação entre a ferramenta e o usuário. A Figura 3-2 mostra a tela inicial da ferramenta onde o usuário pode acionar as principais funções da ferramenta: selecionar a pasta que contém os arquivos que são analisados, selecionar a pasta e o nome do arquivo onde são gravados os resultados, acionar o processo de análise e encerrar o uso da ferramenta.

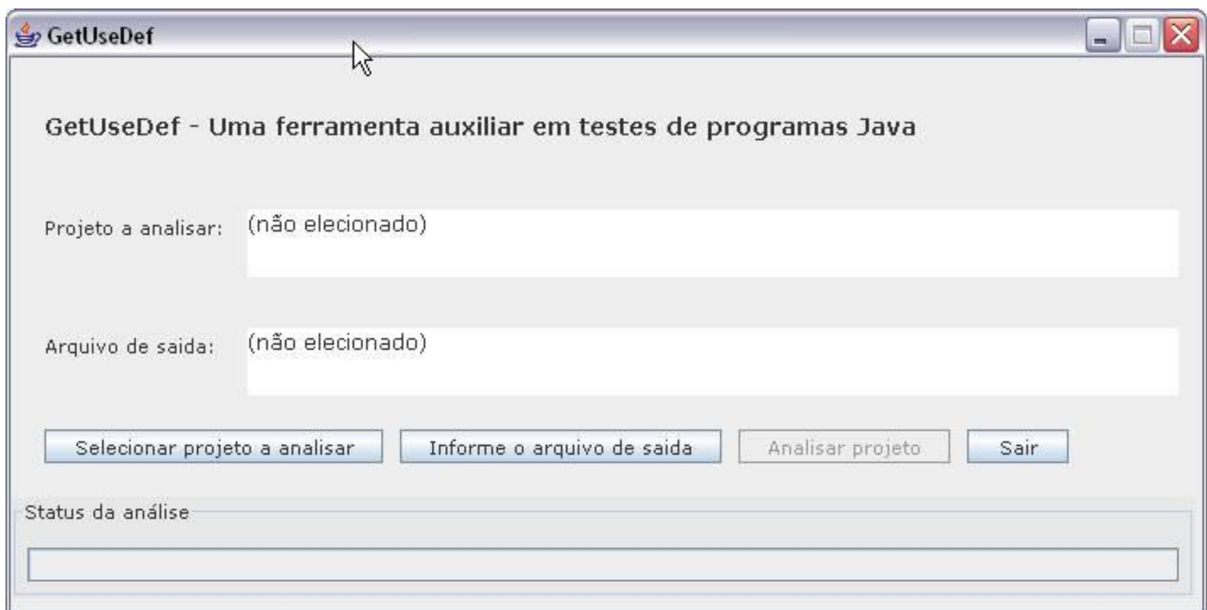


Figura 3-2 - Tela Inicial da ferramenta *GetUseDef*

Já a Figura 3-3 mostra a tela que é exibida após o acionamento do botão “Selecionar o projeto a analisar” na tela inicial. Esta tela possibilita a seleção da pasta que contém os arquivos que são analisados.

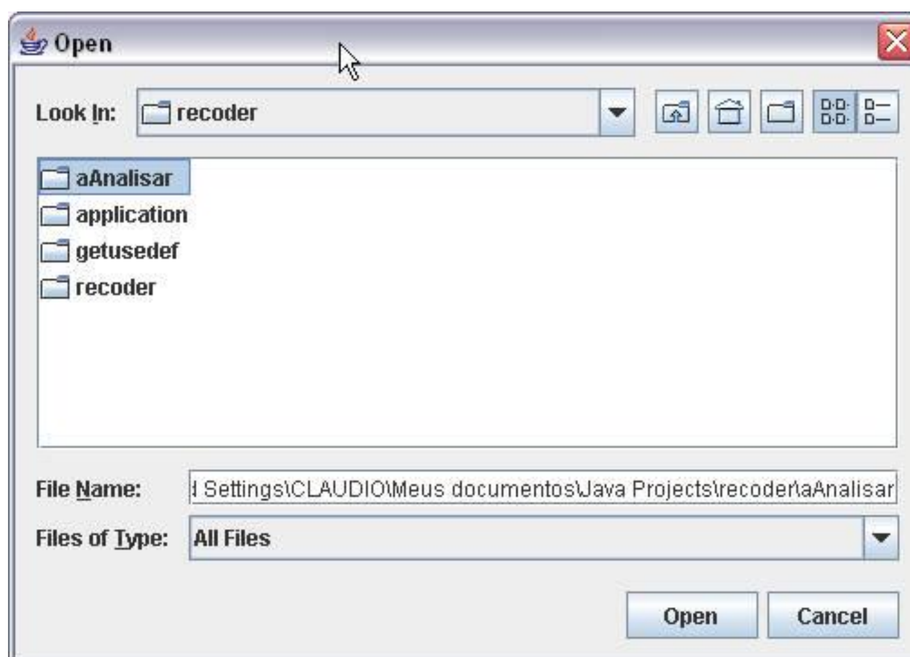


Figura 3-3 - Seleção do projeto a ser analisado

A Figura 3-4 retrata a tela que é exibida após o acionamento do botão “Informe o arquivo de saída” na tela inicial. Esta tela possibilita a seleção da pasta e do nome do arquivo onde são gravados os resultados da análise.

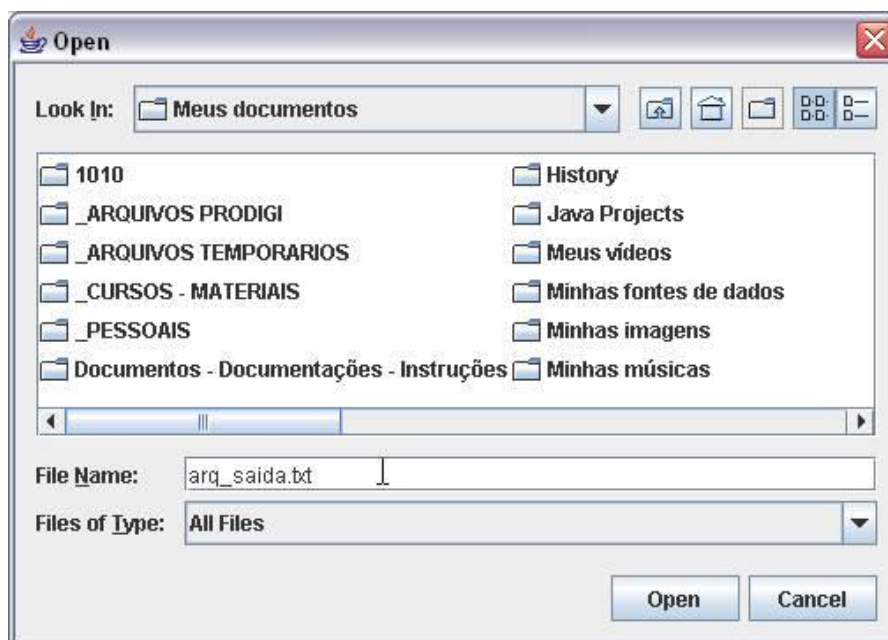


Figura 3-4 - Seleção do arquivo de saída

Por fim, a Figura 3-5 mostra a mensagem que é exibida ao término da análise.

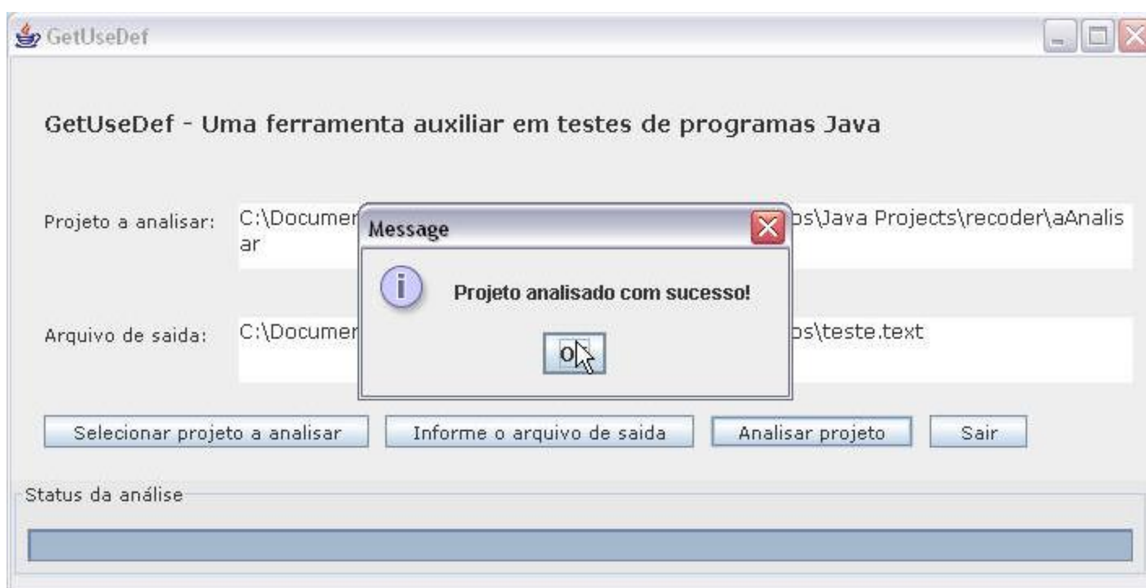


Figura 3-5 - Projeto analisado com sucesso.

Recoder

É um conjunto de classes desenvolvidas por colaboradores voluntários, sem objetivo comercial, que, dentre outras funcionalidades, carrega o conteúdo de um código fonte Java em uma árvore sintática[9], possibilitando análises com diversos objetivos sobre esta estrutura. O *Recoder* é [4] um *framework*[10] Java para metaprogramação de código fonte com o objetivo de prover uma sofisticada infraestrutura para vários tipos de analises Java e ferramentas de transformação.

Analisador Java (fase 1)

É responsável por obter, com base na árvore sintática gerada pelo Recoder, todas as definições e usos dos atributos e de objetos das classes.

Conjunto parcial definições-usos

É o resultado gerado pelo componente Analisador Java (fase 1). É composto por todas as definições e usos obtidas nesta fase da análise.

Analisador Java (fase 2)

Com base na árvore sintática gerada pelo *Recoder* e com base no Conjunto parcial *Definições-Usos*, é responsável por obter as definições e usos restantes.

Conjunto completo definições-usos

É o resultado gerado pelo componente *Analisador Java (fase 2).* É composto por todas as definições e usos obtidas nesta fase da análise.

---

[9] Árvore sintática é uma estrutura que armazena o programa hierarquicamente segundo sua adequação à gramática da linguagem. É usada para verificação sintática do programa, e é a base para a análise semântica e geração de código [14].

[10] Conforme a definição de Grady Booch [17]: "Um framework pode ser visto como um padrão de arquitetura cuja modelagem reflete uma infraestrutura reutilizável e adaptável a algum contexto.".

Extrator resultados

É responsável por gravar, com base no *Conjunto completo Definições-Usos, um arquivo de saída com* os resultados obtidos de forma entendível ao usuário.

Arquivo com resultados formatados

É o arquivo gravado pelo *Extrator de Resultados*, cujo nome e local são selecionados através da *Interface Gráfica.* A Figura 3-6 ilustra um exemplo de um arquivo de resultados gerado pela ferramenta GetUseDef.

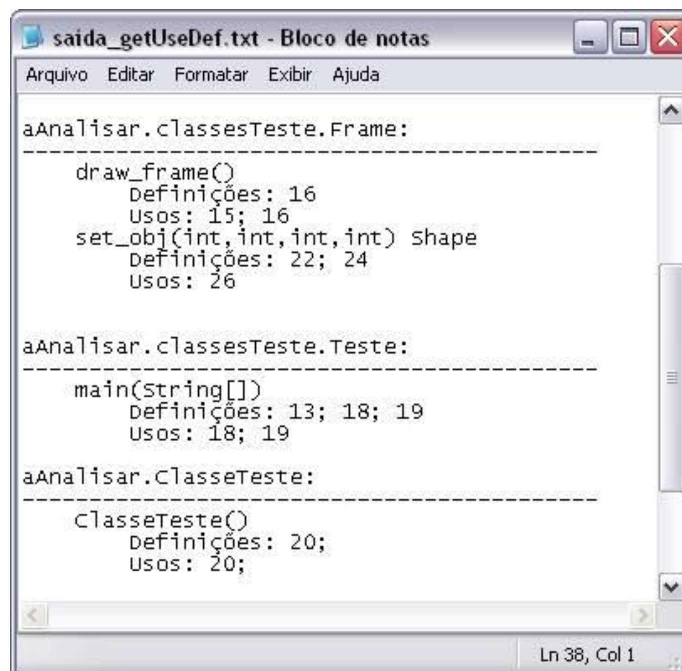


Figura 3-6 - Arquivo de saída com resultados

Conforme pode ser observado na Figura 3-6, o arquivo gerado segue o seguinte padrão para a exibição dos resultados obtidos:

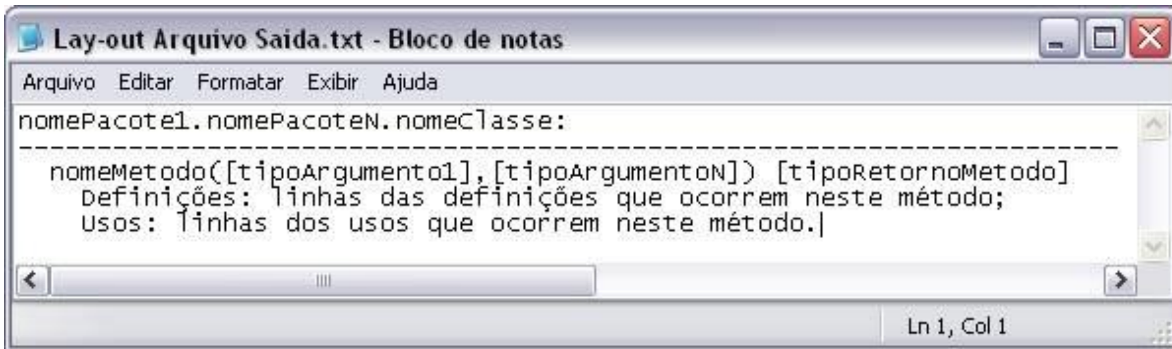


Figura 3-7 – *Lay-out* do arquivo de saída

Nem sempre uma classe pertence a um pacote, nestes casos o nome da classe é exibido da seguinte forma:

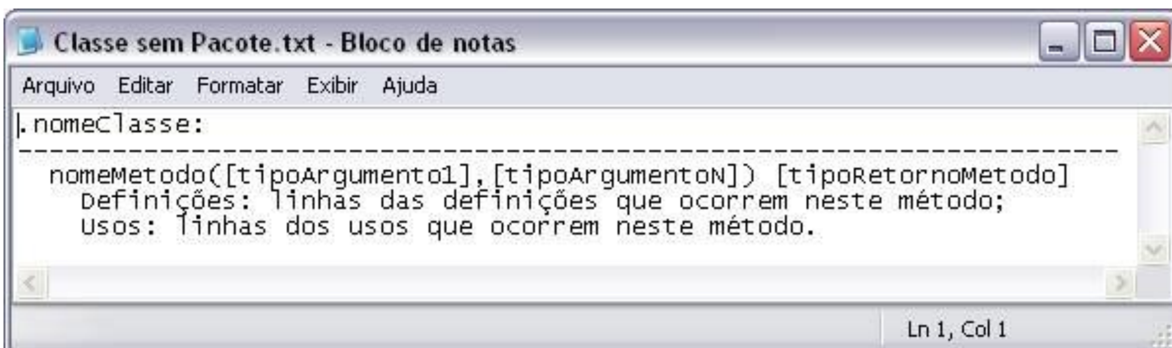


Figura 3-8 – *Layout* do arquivo de saída, uma classe sem pacote

3.2.1. Relacionamento entre os componentes da ferramenta

Através da *Interface Gráfica,* o usuário seleciona a pasta onde os arquivos de código fonte Java de sua aplicação estão localizados. Através da interface gráfica também é selecionada a pasta e o nome do arquivo que conterá os resultados da análise.

Dado início ao processo de análise, o *GetUseDef* envia o caminho dos arquivos fonte, que serão analisados para o *framework Recoder.* Através de um *Parser*[11] *Java,* este carregará cada arquivo de código fonte para uma árvore sintática.

O componente *Analisador Java* percorre cada elemento desta árvore sintática, buscando as definições e usos, à medida que o percurso é feito o componente armazena as definições e usos em uma nova estrutura de dados denominada “Conjunto parcial Definições-Usos”.

Nesta primeira análise são armazenados todas as definições e usos que ocorrem sobre os atributos de uma classe e sobre os objetos contidos em classes. No entanto, antes do fim desta análise, não é possível saber se um método de um objeto ao ser invocado caracteriza uma definição ou um uso sobre este objeto. Esta impossibilidade ocorre devido ao fato de que uma classe pode conter objetos instâncias de outras classes que ainda não foram analisadas. Uma possível solução para este problema seria: ao encontrar um objeto instância de outra classe, parar a análise que está em andamento e analisar a classe deste objeto, ao fim, continuar a análise do ponto onde parou. Este tipo de abordagem é falho ao considerar que é possível que cada classe contenha um objeto instância de outra classe, o que geraria um processamento sem fim.

Por este motivo, o próximo passo da ferramenta é uma segunda análise que é efetuada pelo componente *Analisador Java.* Nesta segunda fase, o componente *Analisador Java* percorre novamente cada elemento da árvore sintática gerada pelo *Framework Recoder* em busca de instruções onde um objeto invoca um método. Com base nos resultados da primeira análise já é possível saber se esta invocação do método realizada pelo objeto caracteriza uma definição ou um uso deste objeto. Neste caso o *Analisador Java* atualiza o *Conjunto Parcial Definições-Usos* com esta nova definição ou uso identificada.

---

[11] Um parser é [15] um analisador sintático, o trabalho dele é verificar a sintaxe do documento e relatar erros, além de permitir, via programas, acesso ao conteúdo do documento.

Ao fim desta segunda análise todas os possíveis du´s estão armazenadas no *Conjunto Parcial Definições-Usos,* assim transformando-o em *Conjunto Completo Definições-Usos.*

O próximo passo então é a conversão dos resultados obtidos para uma forma entendível ao usuário. Este processo é realizado pelo *Extrator de Resultados* que percorre o *Conjunto Completo Definições-Usos* e grava os resultados obtidos no arquivo de saída informado inicialmente pelo usuário através da interface gráfica.

3.3. *Um exemplo de uso da ferramenta GetUseDef*

Nesta seção é apresentado um exemplo completo da aplicação do critério de todos os pares definição-uso, utilizando-se a ferramenta GetUseDef.

Considere o seguinte programa Java ilustrado na Figura 3.9. O programa é iniciado pela Classe3, através do método *main*, Este recebe como parâmetro um valor inteiro que indicará o fluxo de execução do programa.

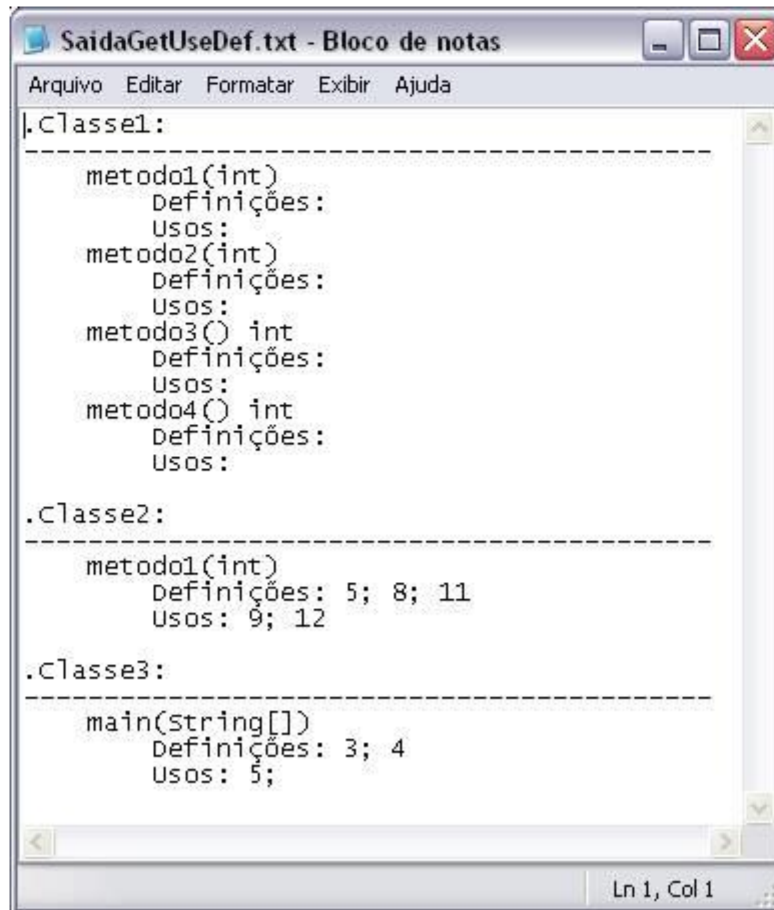
```
01: public class Classe1 {
02:    private int atributo1;
03:    private int atributo2;
04:    public void metodo1(int param1) {
05:       atributo1 = param1;
06:    }
07:    public void metodo2(int param2) {
08:       atributo2 = param2;
09:    }
10:    public int metodo3() {
11:       return atributo1;
12:    }
13:    public int metodo4() {
14:       return atributo2;
15:    }
16: }
```

```
01: public class Classe2 {
02:    public int atributo1;
03:    public int atributo2;
04:    public void metodo1(int opcao) {
05:       Classe1 classe1 = new Classe1();
06:       while (opcao<100) {
07:          if (opcao<=50) {
08:             classe1.metodo1(opcao++);
09:             atributo1 = classe1.metodo3();
10:          } else {
11:             classe1.metodo2(opcao++);
12:             atributo2 = classe1.metodo4();
13:          }
14:       }
15:    }
16: }
```

```
01: public class Classe3 {
02:    public static void main(String strArgs[]) {
03:       Classe2 classe2 = new Classe2();
04:       classe2.metodo1(Integer.parseInt(strArgs[0]));
05:       System.out.println(
06:         classe2.atributo1 + " - " + classe2.atributo2);
07:    }
08: }
```

Figura 3-9 – Classes analisadas pela ferramenta GetUseDef

Após a análise das classes exibidas na Figura 3.9 pela ferramenta GetUseDef, obtemos o arquivo “SaidaGetUseDef.txt” ilustrado na Figura 3.10.




```
.Classe1:
-------------------------------------------
    metodo1(int)
        Definições:
        Usos:
    metodo2(int)
        Definições:
        Usos:
    metodo3() int
        Definições:
        Usos:
    metodo4() int
        Definições:
        Usos:

.Classe2:
-------------------------------------------
    metodo1(int)
        Definições: 5; 8; 11
        Usos: 9; 12

.Classe3:
-------------------------------------------
    main(String[])
        Definições: 3; 4
        Usos: 5;
```

Figura 3-10 – Arquivo resultado da ferramenta GetUseDef

Na Classe1 não foram identificadas definições e usos, pois nesta classe nenhuma variável ou atributo é um objeto. Já na Classe2, foram consideradas como definições as instruções que estão nas linhas 5, 8 e 11 e como usos as instruções que estão nas linhas 9 e 12. Na Classe3 foram consideradas como definições as instruções que estão nas linhas 3 e 4 e como uso a instrução que está na linha 5.

A partir do resultado obtido pela ferramenta GetUseDef, deve-se identificar os objetos existentes nas classes e associá-los às definições e usos identificadas pela ferramenta. Neste exemplo tem-se a seguinte situação:

Classe2, objeto classe1:

Definições: 5, 8 e 11

Usos: 9 e12

Classe3, objeto classe2:

Definições: 3 e 4

Usos: 5

Como nas classes deste exemplo não são implementados mais do que um objeto por classe, o resultado obtido no agrupamento por objeto não difere do resultado obtido pela ferramenta. O próximo passo é fazer todas as possíveis combinações de definições e usos para cada objeto:

classe1: (5,9), (5,12), (8,9), (8,12), (11,9), (11,12);
classe2: (3,5), (4,5).

Eliminando-se os pares que nunca serão exercitados, tem-se o seguinte resultado:

classe1: (8,9), (11,12);
classe2: (4,5).

Após identificar todos os pares du, é necessário exercitá-los pelo menos uma vez e para isso são criados os seguintes casos de teste:

- **Caso 1:** Entrando-se com um valor maior que 100, testa o par du: (4,5);
- **Caso 2:** Entrando-se com um valor maior que 50, testa os pares du: (4,5) e (11,12);
- **Caso 3:** Entrando-se com um valor menor ou igual a 50, testa os pares du: (4,5), (8,9) e (11,12).

4. Conclusões e trabalhos futuros

4.1. Conclusão

Neste trabalho apresentamos conceitos relacionados à atividade de teste, os benefícios e dificuldades relacionados à realização desta tarefa. Apresentamos também uma ferramenta desenvolvida com o objetivo de se reduzir custo, tempo e probabilidades de erros na aplicação do critério baseado em fluxo de objetos proposto por Chen e Kao, todos os pares definição-uso, em programas Java.

A escolha do critério de teste todos os pares definição-uso foi feita devido ao fato de que este é um critério baseado em fluxo de objetos, específico para testes em sistemas orientados a objetos, e tem apresentado bons resultados na revelação de falhas nos sistemas [3, 8].

De início, estudamos o funcionamento do critério de todos os pares definição-uso, em seguida foram levantados os requisitos para o desenvolvimento de uma ferramenta que automatizasse este critério.

4.1.1. Resultados obtidos

Como vimos nos Capítulos anteriores, o critério todos os pares definição-uso é um critério baseado na técnica estrutural, ou seja, seus processos envolvem análises diretamente no código fonte do programa em teste. Devido a este fato, concluímos que para o desenvolvimento da ferramenta para a automação deste critério é necessário que esta, antes da aplicação do critério de teste, faça a leitura dos arquivos fontes em análise, respeitando e entendendo a gramática da linguagem Java.

Realizamos então um estudo sobre os requisitos para o desenvolvimento de um *Parser* Java. Conceitos relacionados ao desenvolvimento de compiladores são necessários e, além disto, um abrangente estudo sobre a linguagem de programação Java, também é necessário.

Constatamos que a linguagem Java oferece uma gama enorme de comandos, trabalha com os mais avançados recursos de orientação a objetos, desta forma torna-se extremamente poderosa. No entanto, os mesmos recursos que tornam a linguagem Java uma linguagem extremamente poderosa, também fazem com que esta possua uma gramática de linguagem extremamente complexa. Constatamos então que o desenvolvimento de um *Parser* Java seria complexo e demandaria muito tempo para ser realizado, tempo este que também seria utilizado para o desenvolvimento da aplicação do critério de todos os pares du, principal função da ferramenta.

Realizamos então um estudo em trabalhos da mesma área, onde tomamos conhecimento da existência de um projeto *open source*[12], conhecido como *Recoder*, que dentre outras funcionalidades, faz o papel de um *parser* Java. Estudos sobre este *framework* foram realizados onde constatamos que mesmo contando com o código fonte dos programas Java já carregados em uma árvore sintática, a atividade de encontrar situações que caracterizem definições e usos de objetos ainda é complexa, pois a "vantagem-desvantagem" da linguagem Java ser extremamente poderosa também se aplica nesta atividade, ou seja, ainda temos que analisar uma árvore sintática gerada sobre uma linguagem cuja gramática é extremamente complexa.

Mesmo diante destas dificuldades foi desenvolvida a ferramenta GetUseDef, que automatiza o primeiro processo do critério todos os pares definição-uso, a obtenção das definições e usos sobre os objetos. A ferramenta desenvolvida recebe como dados de entrada os arquivos fonte que serão analisados e gera como saída um arquivo em formato texto, em *lay-out* entendível ao usuário, contendo as classes, seus respectivos métodos e o número das linhas onde ocorrem definições e usos de cada objeto. O restante dos passos do critério todos os pares du, onde são apurados e exercitados os pares de definição e uso, ainda não está automatizado pela ferramenta, e deve ser realizado pelo *testador* com base no arquivo de saída gerado pela *GetUseDef*.

---

[12] *Open source* significa código aberto, um projeto *open source* tem seus arquivos fontes disponibilizados a quem interessar.

Podemos concluir que mesmo automatizando parte do critério de todos os pares du, a ferramenta *GetUseDef* atende aos objetivos definidos inicialmente: reduzir tempo, custo e probabilidade de erros na atividade de teste.

4.2. Trabalhos Futuros

São destacadas a seguir, possíveis áreas de estudo relacionadas ao tema deste trabalho, identificadas durante os estudos desenvolvidos.

Aprimoramentos nos algoritmos da ferramenta

A estrutura de dados que armazena as ocorrências de definições e usos está limitada a no máximo mil ocorrências de definições e mil ocorrências de usos, podendo-se alterar este limite quando necessário, alterando-se o código fonte. Este limite é em média suficiente para armazenar as ocorrências de definição e uso de programas não muito complexos. A desvantagem em se definir um limite maior é que dependendo do programa a ser analisado, ocorreria uma grande quantidade de memória alocada desnecessariamente.

O ideal para esta estrutura é que se implemente uma lista ligada de definições e usos, assim limitando a quantidade de ocorrências destas a quantidade de memória disponível no computador em uso. Esta alteração aumentaria a praticidade de uso da ferramenta, pois não seria mais necessário a alteração no código fonte quando houvesse necessidade de se analisar um programa complexo. Eliminaria também o problema de alocação desnecessária de memória, pois esta seria alocada conforme fosse necessário.

Alteração no *lay-out* do arquivo de saída da ferramenta

A ferramenta GetUseDef gera um arquivo onde as definições e usos identificadas são agrupadas por pacote, classe e método. Com base neste resultado, o *testador* deve identificar os objetos e associá-los ao número das linhas onde estes foram definidos e usados. O ideal é que a ferramenta exiba os resultados obtidos já

agrupados por pacote, classe e objeto, assim eliminando mais um passo que é realizado manualmente. Para isto, é necessário a alteração no componente “Extrator Resultados”, mudando a forma de gerar o arquivo de saída.

Implementação da próxima fase do critério de todos os pares du

Para uma automação mais abrangente do critério de todos os pares definição-uso, o ideal é que a ferramenta, após obter as definições e usos, gere também os pares de definição-uso. Para isto, uma terceira fase de análise deve ser implementada, onde se utilizaria como base os resultados da segunda fase de análise e o código fonte que está sendo analisado.

Com esta implementação, haveria uma considerável redução do trabalho do *testador* que passaria a ter como função definir casos de teste a partir dos pares de definição-uso, que levem o fluxo de execução do programa a exercitá-los. Esta melhoria reduziria consideravelmente a probabilidade de erros no processo da aplicação do critério e aumentaria a qualidade final da atividade.

Estudos para metrificar os benefícios do uso da ferramenta GetUseDef

É também de interesse a realização de um estudo que traga como resultado números, percentuais e gráficos que provem os benefícios do uso da ferramenta GetUseDef. A seguir são enumerados alguns possíveis estudos nesse sentido:

1) Estudos comparativos entre a aplicação manual do critério de todos os pares definição-uso e a aplicação automatizada deste critério:

    a) Comparação da quantidade de erros na aplicação do critério via processo humano e a mesma via *software*;

    b) Comparação da quantidade de erros detectados via processo humano e a quantidade de erros detectados via *software*;

c) Comparação do tempo e custo gasto com a aplicação manual do critério de todos os pares definição-uso e a aplicação do mesmo critério via *software;*

2) Estudos que mostrem a eficácia do uso da ferramenta GetUseDef na atividade de teste, comparando-a a outras ferramentas que utilizem como base o critério de todos os pares definição-uso ou não;

Referências Bibliográficas

[1] R. S. Pressman. *Software Engeneering*. McGraw-Hill, 1997.

[2] G.J. MYERS. *The art of Software Testing*. J. Wiley, 1979.

[3] Mei-Hwa Chen and Howard M. Kao. *Testing object-oriented programs – an integrated approach. In Proceedings of the Tenth International Symposium on Software Reliability Engineering*. IEEE *Computer Society Press*, 1999.

[4] PROJETO *RECODER*. Disponível no site Sourceforge.
URL: http://recoder.sourceforge.net.
29/05/2005 22:15.

[5] Pressman, Roger S. Engenharia de *software* / Roger S. Pressman. 5.ed. Rio de Janeiro: McGraw-Hill, 2002

[6] Paulo Marcos Siqueira Bueno. Geração Automática de Dados e Tratamento da Não Executabilidade no Teste Estrutural de *Software*. Master´s thesis. Faculdade de Engenharia Elétrica e de Computação da Universidade de Campinas, 1999.

[7] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE, New-York, 1990.

[8] Leandro César Prudente. Um estudo sobre Teste versus Verificação Formal de Programas Java. Master´s thesis. Instituto de Matemática e Estatística da Universidade de São Paulo, 2004.

[9] Sommerville, Ian. Engenharia de *Software* / Ian Sommerville. São Paulo: Addison Wesley, 2003.

[10] B. Hetzel. *Program Test Methods,* Prentice-Hall, N. J., 1973.

[11] B. Beizer. *Black-box Testing Techiniques for Functional Testing of Software Systems.* John Wiley e sons, Inc., New York, 1995.

[12] D. R. Graham. Testing. In Encyclopedia of Software Engineering, pages 1330-1352. John Wiley e Sons, Inc. – John J. Marciniak Editor-in-Chief, N.Y., 1994.

[13] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigatin the applicability of tradicional teste adequacy criteria for object oriented programs. In *Proceedings of the ObjectDays 2000*, October 2000. http://www.users.cs.york.ac.uk/jac/.

[14] Prof. D. Sc. Orivaldo L. Tavares, Universidade Federal do Espírito Santo.
URL: http://www.inf.ufes.br/~tavares/labcomp2000/intro2.html
10/10/2005 - 14:45

[15] Prof. D. Sc. Augusto Sampaio, Universidade Federal de Pernambuco.
URL: http://www.cin.ufpe.br/~in1007/transparencias/javacc.ppt
10/10/2005 - 10:40

[16] Deutsch, M., "Verification and Validation," in *Software Engineering* (R. Jensen and C. Tonies, eds.), Prentice Hall, 1979, pp. 329-408.

[17] Booch, Grady
URL: http://www-128.ibm.com/developerworks/blogs/dw_blog.jspa?blog=317
10/10/2005 - 17:15