Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2585

INTRODUÇÃO AO TESTE DE SOFTWARE COM FERRAMENTAS PARA JAVA
(*Versão 2016-01*)

Auri Marcelo Rizzo Vincenzi
Pedro Henrique Dias Valle
Jacson Rodrigues Barbosa
Cristiane Aparecida Lana
Márcio Eduardo Delamaro
José Carlos Maldonado

Nº 65

NOTAS DIDÁTICAS DO ICMC

São Carlos
OUTUBRO/2016

Introdução ao Teste de Software com Ferramentas para Java*

(*Versão 2016-01*)

Auri Marcelo Rizzo Vincenzi[1], *Jacson Rodrigues Barbosa*[2]
Pedro Henrique Dias Valle[2], *Márcio Eduardo Delamaro*[2]
Cristiane Aparecida Lana[2], *José Carlos Maldonado*[2]

[1]Universidade Federal de São Carlos – DC/UFSCar
`auri@dc.ufscar.br`

[2]Universidade de São Paulo — ICMC/USP
`jacsonrb@icmc.usp.br`, `pedrohenriquevalle@usp.br`
`delamaro@icmc.usp.br`, `cristiane.lana@usp.br`
`jcmaldon@icmc.usp.br`

Resumo

As exigências por software com maior qualidade têm motivado a definição de métodos e técnicas para o desenvolvimento de softwares que atinjam os padrões de qualidade impostos. Com isso, o interesse pela atividade de teste de software aumentou nos últimos anos. Vários pesquisadores investigam os diferentes critérios de teste, buscando obter uma estratégia de teste com baixo custo de aplicação, mas ao mesmo tempo, com grande capacidade em revelar a presença de defeitos. O objetivo desta nota didática é apresentar os aspectos teóricos e práticos relacionados à atividade de teste de software. Uma síntese das técnicas de teste funcional, estrutural e baseada em erros, bem como de critérios de teste pertencentes a cada uma delas, serão apresentados. Fatores utilizados na comparação e avaliação de critérios de teste de software (custo, eficácia e dificuldade de satisfação) também serão abordados, tanto do ponto de vista teórico como empírico. A importância da automatização da atividade de teste será destacada, caracterizando-se os esforços da comunidade científica nessa direção. Dar-se-á ênfase ao critério de teste Análise de Mutantes apresentando uma revisão histórica do seu surgimento e desenvolvimento. Aspectos teóricos e práticos de sua utilização serão abordados e as estratégias que procuram minimizar o custo de aplicação desse critério serão discutidas. A atividade de teste e os problemas pertinentes à ela serão ilustrados utilizando-se as ferramentas *JaBUTi* e *JaCoCo* para critérios estruturais e *MuJava* e *PIT Mutation Testing* para teste de mutação. Identificam-se ainda outras iniciativas e esforços da comunidade para a automatização desses critérios de teste.

*Partes deste trabalho foram extraídas de [1] e [2].

1 Introdução

A Engenharia de Software evoluiu significativamente nas últimas décadas procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de software, em consequência da crescente utilização de sistemas baseados em computação em praticamente todas as áreas da atividade humana, o que provoca uma crescente demanda por qualidade e produtividade, tanto do ponto de vista do processo de produção como do ponto de vista dos produtos gerados. A Engenharia de Software pode ser definida como uma disciplina que aplica os princípios de engenharia com o objetivo de produzir software de alta qualidade a baixo custo [3]. Por meio de um conjunto de etapas que envolvem o desenvolvimento e aplicação de métodos, técnicas e ferramentas, a Engenharia de Software oferece meios para que tais objetivos possam ser alcançados.

O processo de desenvolvimento de software envolve uma série de atividades nas quais, apesar das técnicas, métodos e ferramentas empregados, erros no produto ainda podem ocorrer. Atividades agregadas sob o nome de Garantia de Qualidade de Software são introduzidas ao longo de todo o processo de desenvolvimento, entre elas atividades de VV&T – Verificação, Validação e Teste, com o objetivo de minimizar a ocorrência de erros e riscos associados. Dentre as técnicas de verificação e validação, a atividade de teste é uma das mais utilizadas, constituindo-se em um dos elementos para fornecer evidências da confiabilidade do software em complemento a outras atividades, como por exemplo o uso de revisões e de técnicas formais e rigorosas de especificação e de verificação [4].

A atividade de teste consiste de uma análise dinâmica do produto e é uma atividade relevante para a identificação e eliminação de erros que persistem. Do ponto de vista de qualidade do processo, o teste sistemático é uma atividade fundamental para a ascensão ao Nível 3 do Modelo CMM do *Software Engineering Institute — SEI* [5]. Ainda, o conjunto de informação oriundo da atividade de teste é significativo para as atividades de depuração, manutenção e estimativa de confiabilidade de software [3,6–9]. Salienta-se que a atividade de teste tem sido apontada como uma das mais onerosas no desenvolvimento de software [3,10,11]. Apesar deste fato, Myers et al. observam que aparentemente conhece-se muito menos sobre teste de software do que sobre outros aspectos e/ou atividades do desenvolvimento de software [10].

O teste de produtos de software envolve basicamente quatro etapas: planejamento de testes, projeto de casos de teste, execução e avaliação dos resultados dos testes [3,4,10,11]. Essas atividades devem ser desenvolvidas ao longo do próprio processo de desenvolvimento de software, e em geral, concretizam-se em três fases de teste: de unidade, de integração e de sistema. O teste de unidade concentra esforços na menor unidade do projeto de software, ou seja, procura identificar erros de lógica e de implementação em cada módulo do software, separadamente. O teste de integração é uma atividade sistemática aplicada durante a integração da estrutura do programa visando a descobrir erros associados às interfaces entre os módulos; o objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto. O teste de sistema, realizado após a integração do sistema, visa a identificar erros de funções e características de desempenho que não estejam de acordo com a especificação [3].

Um ponto crucial na atividade de teste, independentemente da fase, é o projeto e/ou a avaliação da qualidade de um determinado conjunto de casos de teste T utilizado para o teste de um produto P, dado que, em geral, é impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais de um produto

em teste. O objetivo é utilizarem-se casos de teste que tenham alta probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço, por questões de produtividade. Segundo Myers [10], o principal objetivo do teste de software é revelar a presença de erros no produto. Portanto, o teste bem sucedido é aquele que consegue determinar casos de teste para os quais o programa em teste falhe. Tem-se observado que a própria atividade de projeto de casos de teste é bastante efetiva em evidenciar a presença de defeitos de software.

Em geral, os critérios de teste de software são estabelecidos, basicamente, a partir de três técnicas: a funcional, a estrutural e a baseada em erros. Na técnica funcional, os critérios e requisitos de teste são estabelecidos a partir da função de especificação do software; na técnica estrutural, os critérios e requisitos são derivados essencialmente a partir das características de uma particular implementação em teste; e, na técnica baseada em erros, os critérios e requisitos de teste são oriundos do conhecimento sobre erros típicos cometidos no processo de desenvolvimento de software. Observa-se também o estabelecimento de critérios de geração de seqüências de teste baseados em Máquinas de Estados Finito [12, 13]. Esses últimos têm sido aplicados no contexto de validação e teste de sistemas reativos e de sistemas orientados a objetos [14–22].

Segundo Howden [23], o teste pode ser classificado de duas maneiras: teste **baseado em especificação** (*specification-based testing*) e teste **baseado em programa** (*program-based testing*). Estre trabalho tem foco no teste baseado em programas e na utilização dos critérios de teste das técnicas funcional, estrutural e baseada em defeitos.

Os critérios da técnica funcional ou caixa-preta têm como objetivo determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados. O problema é que, em geral, a especificação existente é informal e, desse modo, a determinação da cobertura total da especificação que foi obtida por um dado conjunto de casos de teste também é informal [24]. Entretanto, os critérios funcionais podem ser utilizados em qualquer contexto (procedimental ou orientado a objetos) e em qualquer fase de teste sem a necessidade de modificação. Exemplos desses critérios são: particionamento em classe de equivalência [3], análise do valor limite [3], grafo de causa-efeito [3] e teste baseado em estado [16, 19].

Ao contrário, os critérios da técnica estrutural ou caixa-branca requer a inspeção do código fonte e a seleção de casos de teste que exercitem partes do código e não de sua especificação [24].

É importante ressaltar que as técnicas de teste devem ser vistas como complementares e a questão está em como utilizá-las de forma que as vantagens de cada uma sejam melhor exploradas em uma estratégia de teste que leve a uma atividade de teste de boa qualidade, ou seja, eficaz e de baixo custo. As técnicas e critérios de teste fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada, além de constituírem um mecanismo que pode auxiliar a avaliar a qualidade e a adequação da atividade de teste. Critérios de teste podem ser utilizados tanto para auxiliar na geração de conjunto de casos de teste como para auxiliar na avaliação da adequação desses conjuntos.

Dada a diversidade de critérios estabelecidos [3, 4, 10, 11, 25–35] e reconhecido o caráter complementar das técnicas e critérios de teste [35–44], um ponto crucial que se coloca nessa perspectiva é a escolha e/ou a determinação de uma estratégia de teste, que em última análise passa pela escolha de critérios de teste, de forma que as vantagens de cada um desses critérios sejam combinadas objetivando uma atividade de teste de maior qualidade. Estudos teóricos e empíricos de critérios de teste são de extrema relevância

para a formação desse conhecimento, fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia. Identificam-se diversos esforços da comunidade científica nessa direção [35, 41–43, 45–51].

Fundamental se faz o desenvolvimento de ferramentas de teste para o suporte à atividade de teste propriamente dita, uma vez que essa atividade é muito propensa a erros, além de improdutiva, se aplicada manualmente, bem como para dar apoio a estudos empíricos que visem a avaliar e a comparar os diversos critérios de teste. Assim, a disponibilidade de ferramentas de teste propicia maior qualidade e produtividade para as atividades de teste. Observam-se na literatura esforços da comunidade científica nessa direção [11, 35, 43, 52–67].

Pode-se observar que os critérios baseados em análise de fluxo de dados [27, 29–33] e o critério Análise de Mutantes (*Mutation Analysis*) [25, 34, 68] têm sido fortemente investigados por diversos pesquisadores em diferentes aspectos [7, 37, 38, 41, 42, 44, 46, 50, 57, 59, 69–79]. Resultados desses estudos fornecem evidências de que esses critérios, hoje investigados fundamentalmente no meio acadêmico, às vezes em cooperação com a indústria, podem, em médio prazo, constituir o estado da prática em ambientes de produção de software. Uma forte evidência nessa direção é o esforço alocado pela *Telcordia Technologies* (USA) no desenvolvimento da *xSuds* [80], um ambiente que apoia a aplicação de critérios baseados em análise de fluxo de controle e de dados em programas C e C++.

De uma maneira geral, pode-se classificar as contribuições para a área de Teste de Software em: Estudos Teóricos, Estudos Empíricos e Desenvolvimento de Ferramentas. Este texto aborda esses três aspectos, dando-se ênfase a estudos teóricos e empíricos de critérios baseados em análise de fluxo de dados e do critério Análise de Mutantes para o teste de programas em nível de unidade, assim como as ferramentas que apoiam a aplicação desses critérios. Com esse objetivo em mente, o texto está organizado da seguinte forma: na Seção 2 são introduzidos a terminologia e os conceitos básicos pertinentes à atividade de teste. Na Seção 3 são apresentados os critérios de teste mais difundidos das técnicas funcional, estrutural e baseada em erros e ilustrados os principais aspectos operacionais das ferramentas *JaBUTi*, *JaCoCo*, *MuJava* e *PIT Mutation Testing* ou apenas *PIT*. Na Seção 4 são apresentadas as conclusões e perspectivas de trabalhos futuros.

2 Terminologia e Conceitos Básicos

A IEEE tem realizado vários esforços de padronização, entre eles para padronizar a terminologia utilizada no contexto de Engenharia de Software. O padrão IEEE número 610.12-1990 [81] diferencia os termos: **defeito** (*fault*) – passo, processo ou definição de dados incorreto, como por exemplo, uma instrução ou comando incorreto; **engano** (*mistake*) – ação humana que produz um resultado incorreto, com por exemplo, uma ação incorreta tomada pelo programador; **erro** (*error*) – diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução do programa constitui um erro; e **falha** (*failure*) – produção de uma saída incorreta com relação à especificação. Neste texto, os termos engano, defeito e erro serão referenciados como erro (causa) e o termo falha (conseqüência) a um comportamento incorreto do programa. De uma forma geral, os erros são classificados em: **erros computacionais** – o erro provoca uma computação incorreta mas o caminho executado (seqüências de comandos) é igual ao caminho esperado; e **erros de domínio** – o caminho efetivamente executado é diferente do caminho esperado, ou seja, um caminho errado é selecionado.

A atividade de teste é permeada por uma série de limitações [23, 31, 82–84]. Em geral, os seguintes problemas são indecidíveis: dados dois programas, se eles são equivalentes; dados duas seqüências de comandos (caminhos) de um programa, ou de programas diferentes, se eles computam a mesma função; e dado um caminho se ele é executável ou não, ou seja, se existe um conjunto de dados de entrada que levam à execução desse caminho. Outra limitação fundamental é a correção coincidente – o programa pode apresentar, coincidentemente, um resultado correto para um item particular de um dado $d \in D$, ou seja, um particular item de dado ser executado, satisfazer a um requisito de teste e não revelar a presença de um erro.

Diz-se que um programa P com domínio de entrada D é correto com respeito a uma especificação S se $S(d) = P^*(d)$ para qualquer item de dado d pertencente a D, ou seja, se o comportamento do programa está de acordo com o comportamento esperado para todos os dados de entrada. Dados dois programas P_1 e P_2, se $P_1^*(d) = P_2^*(d)$, para qualquer $d \in D$, diz-se que P_1 e P_2 são **equivalentes**. No teste de software, pressupõe-se a existência de um oráculo – o testador ou algum outro mecanismo – que possa determinar, para qualquer item de dado $d \in D$, se $S(d) = P^*(d)$, dentro de limites de tempo e esforços razoáveis. Um oráculo decide simplesmente se os valores de saída são corretos. Sabe-se que o teste exaustivo é impraticável, ou seja, testar para todos os elementos possíveis do domínio de entrada é, em geral, caro e demanda muito mais tempo do que o disponível. Ainda, deve-se salientar que não existe um procedimento de teste de propósito geral que possa ser usado para provar a corretitude de um programa. Apesar de não ser possível, através de testes, provar que um programa está correto, os testes, se conduzidos sistemática e criteriosamente, contribuem para aumentar a confiança de que o software desempenha as funções especificadas e evidenciar algumas características mínimas do ponto de vista da qualidade do produto.

Assim, duas questões são chaves na atividade de teste: "Como os dados de teste devem ser selecionados?" e "Como decidir se um programa P foi suficientemente testado?". Critérios para selecionar e avaliar conjuntos de casos de teste são fundamentais para o sucesso da atividade de teste. Tais critérios devem fornecer indicação de quais casos de teste devem ser utilizados de forma a aumentar as chances de revelar erros ou, quando erros não forem revelados, estabelecer um nível elevado de confiança na correção do programa. Um caso de teste consiste de um par ordenado $(d, S(d))$, no qual $d \in D$ e $S(d)$ é a respectiva saída esperada.

Dados um programa P e um conjunto de casos de teste T, definem-se:

- **Critério de Adequação de Casos de Teste**: predicado para avaliar T no teste de P;
- **Método de Seleção de Casos de Teste**: procedimento para escolher casos de teste para o teste de P.

Existe uma forte correspondência entre métodos de seleção e critérios de adequação de casos de teste pois, dado um critério de adequação C, existe um método de seleção MC que estabelece: selecione T tal que T seja adequado a C. De forma análoga, dado um método de seleção M, existe um critério de adequação CM que estabelece: T é adequado se foi selecionado de acordo com M. Desse modo, costuma-se utilizar o termo "critério de adequação de casos de teste" (ou simplesmente critério de teste) também para designar método de seleção [4, 55]. Dados P, T e um critério C, diz-se que o conjunto de

casos de teste T é C-adequado para o teste de P se T preencher os requisitos de teste estabelecidos pelo critério C. Outra questão relevante nesse contexto é dado um conjunto T C_1-adequado, qual seria um critério de teste C_2 que contribuiria para aprimorar T? Essa questão tem sido investigada tanto em estudos teóricos quanto em estudos empíricos.

Em geral, pode-se dizer que as propriedades mínimas que devem ser preenchidas por um critério de teste C são:

1. garantir, do ponto de vista de fluxo de controle, a cobertura de todos os desvios condicionais;

2. requerer, do ponto de vista de fluxo de dados, ao menos um uso de todo resultado computacional; e

3. requerer um conjunto de casos de teste finito.

As vantagens e desvantagens de critérios de teste de software podem ser avaliadas através de estudos teóricos e empíricos. Do ponto de vista de estudos teóricos, esses estudos têm sido apoiados principalmente por uma relação de inclusão e pelo estudo da complexidade dos critérios [31, 83, 85]. A relação de inclusão estabelece uma ordem parcial entre os critérios, caracterizando uma hierarquia entre eles. Diz-se que um critério C_1 inclui um critério C_2 se para qualquer programa P e qualquer conjunto de casos de teste T_1 C_1-adequado, T_1 for também C_2-adequado e existir um programa P e um conjunto T_2 C_2-adequado que não seja C_1-adequado. A complexidade é definida como o número máximo de casos de teste requeridos por um critério, no pior caso. No caso dos critérios baseados em fluxo de dados, esses têm complexidade exponencial, o que motiva a condução de estudos empíricos para determinar o custo de aplicação desses critérios do ponto de vista prático. Mais recentemente, alguns autores têm abordado do ponto de vista teórico a questão de eficácia de critérios de teste, e têm definido outras relações, que captem a capacidade de revelar erros dos critérios de teste [37, 38, 86, 87].

Do ponto de vista de estudos empíricos, três aspectos costumam ser avaliados: custo, eficácia e *strength* (ou dificuldade de satisfação) [40, 41, 46, 73, 78, 88]. O fator custo reflete o esforço necessário para que o critério seja utilizado; em geral é medido pelo número de casos de teste necessários para satisfazer o critério. A eficácia refere-se à capacidade que um critério possui de detectar a presença de erros. O fator *strength* refere-se à probabilidade de satisfazer-se um critério tendo sido satisfeito um outro critério [41].

Uma atividade muito citada na condução e avaliação da atividade de teste é a análise de cobertura, que consiste basicamente em determinar o percentual de elementos requeridos por um dado critério de teste que foram exercitados pelo conjunto de casos de teste utilizado. A partir dessa informação o conjunto de casos de teste pode ser aprimorado, acrescentando-se novos casos de teste para exercitar os elementos ainda não cobertos. Nessa perspectiva, é fundamental o conhecimento sobre as limitações teóricas inerentes à atividade de teste, pois os elementos requeridos podem ser não executáveis, e em geral, determinar a não executabilidade de um dado requisito de teste envolve a participação do testador.

3 Técnicas e Critérios de Teste

Conforme mencionado anteriormente, para se conduzir e avaliar a qualidade da atividade de teste têm-se as técnicas de teste funcional, estrutural e baseada em erros. Tais

técnicas diferenciam-se pela origem da informação utilizada na avaliação e construção dos conjuntos de casos de teste [4]. Neste texto apresentam-se com mais detalhes as duas últimas técnicas, mais especificamente os critérios Potenciais-Usos [4], o critério Análise de Mutantes [34], e as ferramentas de suporte *JaBUTi* [89], JaCoCo [90], *MuJava* [91], e *PIT* [92]. Através desses critérios, ilustram-se os principais aspectos pertinentes à atividade de teste de cobertura de software. Para propiciar uma visão mais abrangente, apresentam-se primeiramente uma visão geral da técnica funcional e os critérios mais conhecidos dessa técnica. O programa *identifier* (Figura 1) será utilizado para facilitar a ilustração dos conceitos desenvolvidos neste texto.

3.1 Técnica Funcional

O teste funcional também é conhecido como teste caixa preta [93] pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os dados de entrada fornecidos e as respostas produzidas como saída. Na técnica de teste funcional são verificadas as funções do sistema sem se preocupar com os detalhes de implementação.

O teste funcional envolve dois passos principais: identificar as funções que o software deve realizar e criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software [94]. As funções que o software deve possuir são identificadas a partir de sua especificação. Assim, uma especificação bem elaborada e de acordo com os requisitos do usuário é essencial para esse tipo de teste.

Alguns exemplos de critérios de teste funcional são [94]:

- **Particionamento em Classes de Equivalência**: a partir das condições de entrada de dados identificadas na especificação, divide-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas. Em seguida seleciona-se o menor número possível de casos de teste, baseando-se na hipótese que um elemento de uma dada classe seria representativo da classe toda, sendo que para cada uma das classes inválidas deve ser gerado um caso de teste distinto. O uso de particionamento permite examinar os requisitos mais sistematicamente e restringir o número de casos de teste existentes. Alguns autores também consideram o domínio de saída do programa para estabelecer as classes de equivalência.

- **Análise do Valor Limite**: é um complemento ao critério Particionamento em Classes de Equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa; ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, pois nesses pontos se concentra um grande número de erros. O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída.

- **Grafo de Causa-Efeito**: os critérios anteriores não exploram combinações das condições de entrada. Este critério estabelece requisitos de teste baseados nas possíveis combinações das condições de entrada. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. A seguir é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.

```
/*****************************************************************************************
Identifier.java
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly Pascal'
(uma estranha variante do Pascal).  Um identificador valido deve comecar com uma letra e conter
apenas letras ou digitos.  Alem disso, deve ter no minimo 1 caractere e no maximo 6 caracteres de
comprimento
*****************************************************************************************/

         public class Identifier {
             public boolean validateIdentifier(String s) {
 /*  1 */       char achar;
 /*  1 */       boolean valid_id = false;
 /*  1 */       achar = s.charAt(0);
 /*  1 */       valid_id = valid_s(achar);
 /*  1 */       if (s.length() > 1) {
 /*  2 */          achar = s.charAt(1);
 /*  2 */          int i = 1;
 /*  3 */          while (i < s.length() - 1) {
 /*  4 */             achar = s.charAt(i);
 /*  4 */             if (!valid_f(achar)){
 /*  5 */                valid_id = false;}
 /*  6 */             i++;
 /*  6 */          }
 /*  6 */       }

 /*  7 */       if (valid_id && (s.length() >= 1) & &(s.length() < 6))
 /*  8 */          return true;
 /*  9 */       else
 /*  9 */          return false;
 /*  10 */    }



             public boolean valid_s(char ch) {
 /* 1 */        if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')))
 /* 2 */            return true;
 /* 3 */        else
 /* 4 */            return false;
 /* 5 */        }

            public boolean valid_f(char ch) {
 /* 1 */        if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z'))
                            || ((ch >= '0') && (ch <= '9')))
 /* 2 */            return true;
 /* 3 */        else
 /* 4 */            return false;
 /* 5 */     }
 /* 10 */}
```

Figura 1: Programa exemplo: *identifier* (contém ao menos um erro).

Um dos problemas relacionado aos critérios funcionais é que muitas vezes a especificação do programa é feita de modo descritivo e não formal. Dessa maneira, os requisitos de teste derivados de tais especificações são também, de certa forma, imprecisos e informais. Como consequência, tem-se dificuldade em automatizar a aplicação de tais critérios, que ficam, em geral, restritos à aplicação manual. Por outro lado, para a aplicação desses critérios é essencial apenas que se identifiquem as entradas, a função a ser computada e a saída do programa, o que os tornam aplicáveis praticamente em todas fases de teste (unidade, integração e sistema) [35].

Offutt e Irvine [95], por exemplo, propõem a utilização do Método de Partição-Categoria (*Category-Partition Method*) no teste de programas OO. Segundo os autores, existe muita preocupação no desenvolvimento de novas técnicas e critérios para o teste de programas OO sem que se tenha investigado a eficácia das técnicas e critérios tradicionais nesse contexto. Offutt e Irvine comentam que diversos autores acham que utilizar somente técnicas de teste tradicionais é insuficiente para testar programas OO mas, segundo Offutt e Irvine, somente um autor dá alguma evidência desse fato e suas evidências ainda não são conclusivas. Nesse contexto eles propõem o uso de um critério de teste baseado em especificação (que, teoricamente é aplicável tanto para programas procedimentais quanto OO, indistintamente) no teste de programas OO.

Outro critério baseado em especificação é o Particionamento em Classes de Equivalência. A título de ilustração, considerando o programa *identifier* e o critério Particionamento em Classes de Equivalência, são identificadas na Tabela 1 as condições de entrada e classes de equivalência válidas e inválidas. A partir dessas classes o seguinte conjunto de casos de teste poderia ser elaborado: T_0 = {(a1, Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)}. De posse do conjunto T_0, seria natural indagar se esse conjunto exercita todos os comandos ou todos os desvios de fluxo de controle de uma dada implementação. Usualmente, lança-se mão de critérios estruturais de teste, apresentados a seguir, como critérios de adequação ou critérios de cobertura para se analisar questões como essas, propiciando a quantificação e a qualificação da atividade de teste de acordo com o critério escolhido. Quanto mais rigoroso o critério utilizado e se erros não forem revelados, maior a confiança no produto em desenvolvimento.

Tabela 1: Classes de Equivalência para o programa *identifier*.

Restrições de Entrada	**Classes Válidas**	**Classes Inválidas**
Tamanho (t) do identificador	$1 \leq t \leq 6$ (1)	$t > 6$ (2)
Primeiro caracter (c) é uma letra	Sim (3)	Não (4)
Contém somente caracteres válidos	Sim (5)	Não (6)

Steve *et al.* [96] definiram o critério de Teste Funcional Sistemático (*Systematic Functional Testing*) o qual, basicamente, combina os critérios Particionamento em Classes de Equivalência e Análise do Valor Limite, para a geração de casos de teste baseados em especificação. No estudo de caso conduzido, o conjunto de casos de teste desenvolvido utilizando o Teste Funcional Sistemático foi capaz de distinguir 100% dos mutantes de unidades gerados não equivalentes do programa Cal (utilitário do UNIX) ao passo que os outros 11 conjuntos de teste gerados gerados a partir dos critérios Particionamento em

Classe de Equivalência, Análise do Valor Limite e Teste Aleatório resultaram em escores de mutação entre 98% e 56%.

3.2 Técnica Estrutural

A técnica estrutural apresenta uma série de limitações e desvantagens decorrentes das limitações inerentes às atividades de teste de programa enquanto estratégia de validação [23, 31, 82, 83]. Esses aspectos introduzem sérios problemas na automatização do processo de validação de software [MAL91]. Independentemente dessas desvantagens, essa técnica é vista como complementar à técnica funcional [94] e informações obtidas pela aplicação desses critérios têm sido consideradas relevantes para as atividades de manutenção, depuração e confiabilidade de software [6–9, 94].

Na técnica de teste estrutural, também conhecida como teste caixa branca (em oposição ao nome caixa preta), os aspectos de implementação são fundamentais na escolha dos casos de teste. O teste estrutural baseia-se no conhecimento da estrutura interna da implementação. Em geral, a maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como **grafo de fluxo de controle** ou **grafo de programa**. Um programa P pode ser decomposto em um conjunto de blocos disjuntos de comandos; a execução do primeiro comando de um bloco acarreta a execução de todos os outros comandos desse bloco, na ordem dada. Todos os comandos de um bloco, possivelmente com exceção do primeiro, têm um único predecessor e exatamente um único sucessor, exceto possivelmente o último comando.

A representação de um programa P como um grafo de fluxo de controle ($G = (N, E, s)$) consiste em estabelecer uma correspondência entre nós e blocos e em indicar possíveis fluxos de controle entre blocos através dos arcos. Um grafo de fluxo de controle é portanto um grafo orientado, com um único nó de entrada $s \in N$ e um único nó de saída, no qual cada vértice representa um bloco indivisível de comandos e cada aresta representa um possível desvio de um bloco para outro. Cada bloco tem as seguintes características: 1) uma vez que o primeiro comando do bloco é executado, todos os demais são executados sequencialmente; e 2) não existe desvio de execução para nenhum comando dentro do bloco. A partir do grafo de programa podem ser escolhidos os componentes que devem ser executados, caracterizando assim o teste estrutural. Considere o programa *identifier*. Na Figura 1 identifica-se a caracterização dos blocos de comandos através dos números à esquerda dos comandos. A Figura 2 ilustra o grafo de fluxo de controle do programa *identifier* (função `main`).

Seja um grafo de fluxo de controle $G = (N, E, s)$ onde N representa o conjunto de nós, E o conjunto de arcos, e s o nó de entrada. Um **caminho** é uma sequência finita de nós $(n_1, n_2, \ldots, n_k)$, $k \geq 2$, tal que existe um arco de n_i para $n_i + 1$ para $i = 1, 2, \ldots, k - 1$. Um caminho é um **caminho simples** se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos; se todos os nós são distintos diz-se que esse caminho é um **caminho livre de laço**. Um **caminho completo** é um caminho onde o primeiro nó é o nó de entrada e o último nó é o nó de saída do grafo G. Seja $IN(x)$ e $OUT(x)$ o número de arcos que entram e que saem do nó x respectivamente. Se $IN(x) = 0$ x é uma nó de entrada, e se $OUT(x) = 0$, x é um nó de saída. Em relação ao programa *identifier*, (2,3,7,8,10) é um caminho simples e livre de laços e o caminho (1,2,3,4,5,6,3,7,8,10) é um caminho completo. Observe que o caminho (1,3,7,8,10) é não

executável e qualquer caminho completo que o inclua é também não executável, ou seja, não existe um dado de entrada que leve à execução desse caminho.

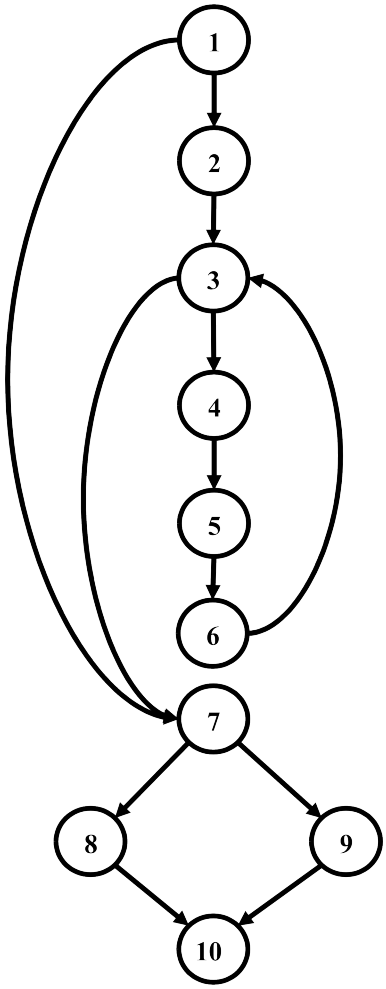

Figura 2: Grafo de Fluxo de Controle de Programa identifier

Os critérios de teste estrutural baseiam-se em diferentes tipos de conceitos e componentes de programas para determinar os requisitos de teste. Na Tabela 2 ilustram-se alguns elementos componentes de programas e critérios associados.

Tabela 2: Elementos e critérios associados em relação ao programa *identifier*.

Elemento	**Exemplo (*identifier*)**	**Critério**
Nó	6	Todos-Nós
Arco	(4,5)	Todos-Arcos
Laço	(3,4,5,6, 3)	Boundary-Interior
Caminho	(1,2,3,4,5,6,3,7,9,10)	Todos-Caminhos
Definição de variáveis	`int i = 1`	Todas-Defs
Uso predicativo de variáveis	`s.length() > 1)`	Todos-P-Usos
Uso computacional de variáveis	`achar = s.charAt(0)`	Todos-C-Usos

Os critérios de teste estrutural são, em geral, classificados em:

- **Critérios Baseados em Fluxo de Controle**: utilizam apenas características de controle da execução do programa, como comandos ou desvios, para determinar quais estruturas são necessárias. Os critérios mais conhecidos dessa classe são **Todos-Nós** – exige que a execução do programa passe, ao menos uma vez, em cada vértice do grafo de fluxo, ou seja, que cada comando do programa seja executado pelo menos uma vez; **Todos-Arcos** – requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitada pelo menos uma vez; e **Todos-Caminhos** – requer que todos os caminhos possíveis do programa sejam executados [94]. Outros critérios dessa categoria são: **Cobertura de Decisão**; **Cobertura de Condição**; **Cobertura de Condições Múltiplas**; **LCSAJ** (*Linear Code Sequence and Jump*) [97]; o critério **Boundary-Interior** [98]; e a família de critérios ***K*-tuplas** requeridas de Ntafos [32].

- **Critérios Baseados em Fluxo de Dados**: utilizam informações do fluxo de dados do programa para determinar os requisitos de teste. Esses critérios exploram as interações que envolvem definições de variáveis e referências a tais definições para estabelecerem os requisitos de teste [31]. Exemplos dessa classe de critérios são os **Critérios de Rapps e Weyuker** [30, 31] e os **Critérios Potenciais-Usos** [4]. Visto que tais critérios serão utilizados nos estudos comparativos a serem realizados durante o desenvolvimento deste trabalho, as próximas seções destinam-se a descrevê-los mais detalhadamente.

- **Critérios Baseados na Complexidade**: utilizam informações sobre a complexidade do programa para derivar os requisitos de teste. Um critério bastante conhecido dessa classe é o **Critério de McCabe**, que utiliza a complexidade ciclomática do grafo de programa para derivar os requisitos de teste. Essencialmente, esse critério requer que um conjunto de caminhos linearmente independentes do grafo de programa seja executado [94].

Os casos de teste obtidos durante a aplicação dos critérios funcionais podem corresponder ao conjunto inicial dos testes estruturais. Como, em geral, o conjunto de casos de teste funcional não é suficiente para satisfazer totalmente um critério de teste estrutural, novos casos de teste são gerados e adicionados ao conjunto até que se atinja o grau de satisfação desejado, explorando-se, desse modo, os aspectos complementares das duas técnicas [43].

Um problema relacionado ao teste estrutural é a impossibilidade, em geral, de se determinar automaticamente se um caminho é ou não executável, ou seja, não existe um algoritmo que dado um caminho completo qualquer decida se o caminho é executável e forneça o conjunto de valores que causam a execução desse caminho [99]. Assim, é preciso a intervenção do testador para determinar quais são os caminhos não executáveis para o programa sendo testado.

3.2.1 Critérios Baseados em Fluxo de Dados

Em meados da década de 70 surgiram os critérios baseados em análise de fluxo de dados [27], os quais utilizam informações do fluxo de dados para derivar os requisitos de teste. Uma característica comum dos critérios baseados em fluxo de dados é que eles requerem que sejam testadas as interações que envolvam definições de variáveis e subsequentes referências a essas definições [27,29,31–33]. Uma motivação para a introdução dos

critérios baseados na análise de fluxo de dados foi a indicação de que, mesmo para programas pequenos, o teste baseado unicamente no fluxo de controle não ser eficaz para revelar a presença até mesmo de erros simples e triviais. A introdução dessa classe de critérios procura fornecer uma hierarquia entre os critérios Todos-Arcos e Todos-Caminhos, procurando tornar o teste mais rigoroso, já que o teste de Todos-Caminhos é, em geral, impraticável. Segundo Ural [33], esses critérios são mais adequados para certas classes de erros, como erros computacionais, uma vez que dependências de dados são identificadas, e portanto, segmentos funcionais são requeridos como requisitos de teste.

Rapps e Weyuker propuseram o Grafo **Def-Uso** (*Def-Use Graph*) que consiste em uma extensão do grafo de programa [30, 31]. Nele são adicionadas informações a respeito do fluxo de dados do programa, caracterizando associações entre pontos do programa onde é atribuído um valor a uma variável (chamado de **definição** da variável) e pontos onde esse valor é utilizado (chamado de referência ou **uso** de variável). Os requisitos de teste são determinados com base em tais associações. A Figura 3 ilustra o Grafo-Def-Uso do programa *identifier*. Conforme o modelo de fluxo de dados definido em [4], uma definição de variável ocorre quando um valor é armazenado em uma posição de memória. Em geral, em um programa, uma ocorrência de variável é uma definição se ela está: i) no lado esquerdo de um comando de atribuição; ii) em um comando de entrada; ou iii) em chamadas de procedimentos como parâmetro de saída. A passagem de valores entre procedimentos através de parâmetros pode ser por valor, referência ou por nome [100]. Se a variável for passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são diferenciadas das demais e são ditas definidas por referência. A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo. Dois tipos de usos são distinguidos: *c-uso* e *p-uso*. O primeiro tipo afeta diretamente uma computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado; o segundo tipo afeta diretamente o fluxo de controle do programa.

O critério mais básico dos critérios baseados em análise de fluxo de dados é o critério Todas-Definições (*all-defs*) e faz parte da família de critérios definidos por Rapps e Weyuker [31]. Entre os critérios dessa família o critério Todos-Usos (*all-uses*) tem sido um dos mais utilizados e investigados.

- **Todas-Definições**: requer que cada definição de variável seja exercitada pelo menos uma vez, não importa se por um c-uso ou por um p-uso.

- **Todos-Usos**: requer que todas as associações entre uma definição de variável e seus subseqüentes usos (*c-usos* e *p-usos*) sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho onde a variável não é redefinida.

Por exemplo, para exercitar a definição da variável `valid_id` definida no nó 2, de acordo com o critério Todas-Definições, poderiam ser executados um dos seguintes subcaminhos: (1,2,3,7,8,9); (1,2,3,7,9,10); (1,7,8,10) e (1,7,9,10). O subcaminho (1,3,7,8,10) é não executável, e qualquer caminho completo que o inclua também é não executável. Se qualquer um dos demais caminhos for exercitado, o requisito de teste estaria sendo satisfeito, e para satisfazer o critério Todas-Definições esta análise teria que ser feita para toda definição que ocorre no programa. Em relação ao critério Todos-Usos, com respeito à mesma definição, seriam requeridas as seguinte associações: (1,(7,8), *valid_id*) e (1,(8,9),

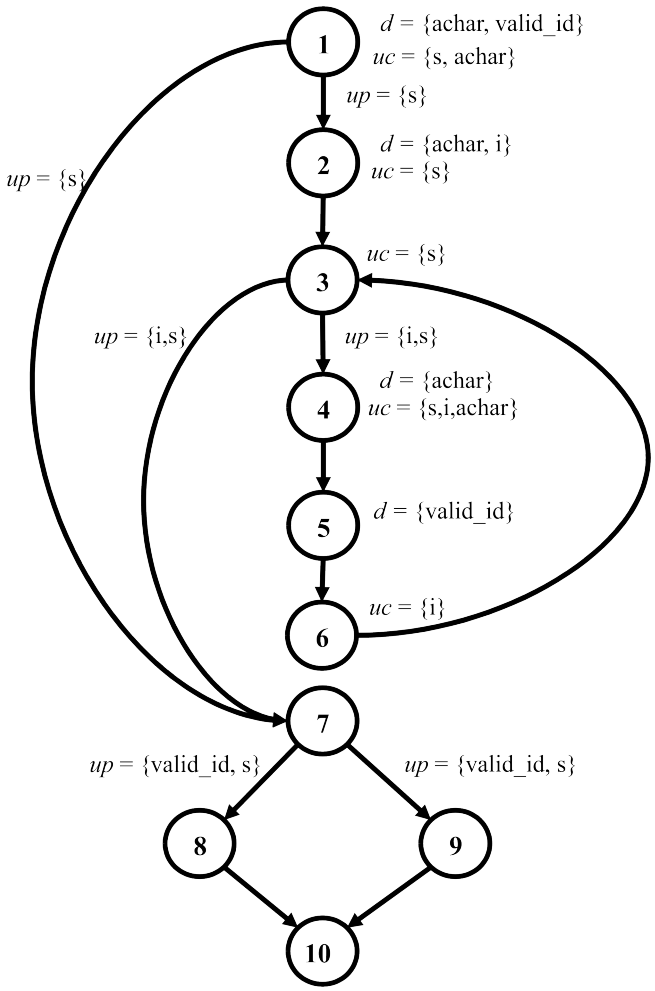

Figura 3: Grafo Def-Uso do Programa identifier

valid_id). As notações (i,j,var) e (i,(j,k),var) indicam que a variável var é definida no nó i e existe um uso computacional de var no nó j ou um uso predicativo de var no arco (j,k), respectivamente, bem como pelo menos um caminho livre de definição do nó i ao nó j ou ao arco (j,k). Observe que a associação (1,(7,8), *valid_id*) é não executável pois o único caminho que livre de definição possível de exercitá-la seria um caminho que incluísse o subcaminho (1,2,3,7,8). Esta mesma análise deveria ser feita para todas as demais variáveis e associações pertinentes, a fim de satisfazer o critério Todos-Usos.

A maior parte dos critérios baseados em fluxo de dados, para requerer um determinado elemento (caminho, associação, etc.), exige a ocorrência explícita de um uso de variável e não garante, necessariamente, a inclusão dos critérios Todos-Arcos na presença de caminhos não executáveis, presentes na maioria dos programas.

A **relação de inclusão** é uma importante propriedade dos critérios, sendo utilizada para avaliá-los, do ponto de vista teórico. O critério Todos-Arcos, por exemplo, inclui o critério Todos-Nós, ou seja, qualquer conjunto de casos de teste que satisfaz o critério Todos-Arcos também satisfaz o critério Todos-Nós, necessariamente. Quando não é possível estabelecer essa ordem de inclusão para dois critérios, como é o caso de Todas-Defs e Todos-Arcos, diz-se que tais critérios são incomparáveis [31]. Deve-se observar que os critérios Potenciais-Usos são os únicos critérios baseados em análise de fluxo de

dados que satisfazem, na presença de caminhos não executáveis, as propriedades mínimas esperadas de um critério de teste, e que nenhum outro critério baseado em análise de fluxo de dados os inclui. Um aspecto relevante é que alguns dos critérios Potenciais-Usos "*bridge the gap*" entre os critérios Todos-Arcos e Todos-Caminhos mesmo na presença de caminhos não executáveis, o que não ocorre para os demais critérios baseados em fluxo de dados.

Como já citado, uma das desvantagens do teste estrutural é a existência de caminhos requeridos não executáveis. Existe também o problema de caminhos ausentes, ou seja, quando uma certa funcionalidade deixa de ser implementada no programa, não existe um caminho que corresponda àquela funcionalidade e, como consequência, nenhum caso de teste será requerido para exercitá-la. Mesmo assim, esses critérios estabelecem de forma rigorosa os requisitos de teste a serem exercitados, em termos de caminhos, associações definição-uso, ou outras estruturas do programa, fornecendo medidas objetivas sobre a adequação de um conjunto de teste para o teste de um dado programa P. Esse rigor na definição dos requisitos favorece a automatização desses critérios.

Os critérios estruturais têm sido utilizados principalmente no teste de unidade, uma vez que os requisitos de teste por eles exigidos limitam-se ao escopo da unidade. Vários esforços de pesquisa no sentido de estender o uso de critérios estruturais para o teste de integração podem ser identificados. Haley e Zweben propuseram um critério para selecionar caminhos em um módulo que deveria ser testado novamente na fase de integração com base em sua interface [101]. Linnenkugel e Müllerburg apresentaram uma série de critérios que estendem os critérios baseados em fluxo de controle e em fluxo de dados para o teste de integração [70]. Harrold e Soffa propuseram uma técnica para determinar as estruturas de definição-uso interprocedurais permitindo a aplicação dos critérios baseados em análise de fluxo de dados em nível de integração [102]. Jin e Offutt definiram alguns critérios baseados em uma classificação de acoplamento entre módulos [103]. Vilela, com base no conceito de potencial-uso, estendeu os critérios Potenciais-Usos para o teste de integração [104].

Harrold e Rothermel [75] estenderam o teste de fluxo de dados para o teste de classes. Os autores comentam que os critérios de fluxo de dados destinados ao teste de programas procedimentais [31,105,106] podem ser utilizados tanto para o teste de métodos individuais quanto para o teste de métodos que interagem entre si dentro de uma mesma classe. Entretanto, esses critérios não consideram interações de fluxo de dados quando os usuários de uma classe invocam sequência de métodos em uma ordem arbitrária.

Para resolver esse problema, os autores apresentam uma abordagem que permite testar diferentes tipos de interações de fluxo de dados entre classes. A abordagem proposta usa as técnicas tradicionais de fluxo de dados para testar os métodos individuais e as interações entre os métodos dentro de mesma classe. Para testar os métodos que são acessíveis fora da classe e podem serem utilizados por outras classes, uma nova representação, denominada **grafo de fluxo de controle de classe** (*CCFG - class control flow graph*), foi desenvolvida. A partir do CCFG, novos requisitos de teste inter-método, intra-classe e inter-classe podem ser derivados [75].

Vincenzi *et al.* [107] também têm investigado o uso de critérios de fluxo de controle e de dados no teste de programas OO e de componentes [108]. Visando à desenvolver uma solução que fosse aplicável tanto a programas OO quanto componentes de software (os quais, em geral, são testados pelos clientes utilizando somente técnicas funcionais), investigou-se como realizar análise estática de programas Java diretamente a partir do

código objeto (Java bytecode). Com isso, independentemente da existência do código fonte da aplicação sendo testada, é possível derivar requisitos de teste estruturais os quais podem ser utilizados tanto para avaliar a qualidade de conjuntos de teste quanto para a própria geração de casos de teste. Para apoiar a aplicação do teste estrutural intra-método em programas e componentes Java foi desenvolvida a ferramenta JaBUTi (Java Bytecode Understanding and Testing) [67].

3.2.2 A Ferramenta de Teste JaBUTi

Várias são as iniciativas de desenvolvimento de ferramentas de teste para apoiar a aplicação de critérios de teste [11, 35, 43, 52–65].

Para ilustrar os conceitos abordados acima será utilizada a ferramenta JaBUTi [109], desenvolvida desenvolvida pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (São Carlos). Ela é uma ferramenta de suporte ao teste estrutural para programas Java. Nela estão implementados critérios baseados em fluxo de controle e critérios baseados em fluxo de dados. Uma das características que diferenciam a JaBUTi é o fato que toda a análise estática necessária para a realização do teste é feita sobre o programa objeto, ou seja, sobre o bytecode Java e não sobre o programa fonte [89].

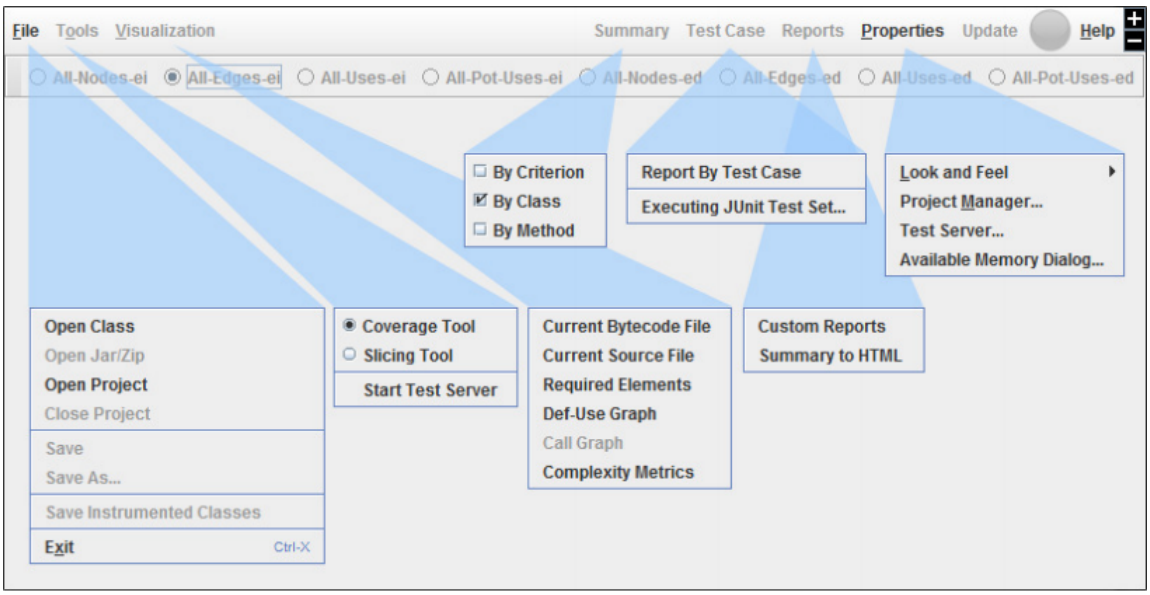

Figura 4: Opções disponíveis na ferramenta JaBUTi

As principais funcionalidades da ferramenta JaBUTi são [89]:

- O menu **Tools** dá acesso as ferramentas que compõem a JaBUTi.
- O menu **Visualization** oferece diferentes formas de visualização das classes, métodos e requisitos de teste gerenciados pela JaBUTi.
- O menu **Summary** fornece relatórios sintéticos sobre a cobertura do projeto considerando diferentes níveis de abstração.
- O menu **Test Case** oferece opções para a manipulação do conjunto de teste e geração de relatórios de cobertura por caso de teste.

- O menu **Reports** permite que os relatórios gerados nas telas da JaBUTi sejam exportados no formato HTML.
- O menu **Update** oferece uma informação visual comunicando ao testador que novas informações de rastro de execução estão disponíveis para atualizar as medidas de cobertura.
  - O botão **Update** se torna vermelho toda vez que um novo caso de teste é executado ou importado e as informações de sua execução são adicionadas ao final do arquivo de rastro de execução
- O menu **Help** oferece apenas a opção sobre os autores e desenvolvedores da JaBUTi.

Para criação do projeto de teste, o testador deve fornecer o nome de uma classe base, em seguida a ferramenta JaBUTi exibe uma janela de gerenciador de projeto, conforme ilustrado na Figura 5.

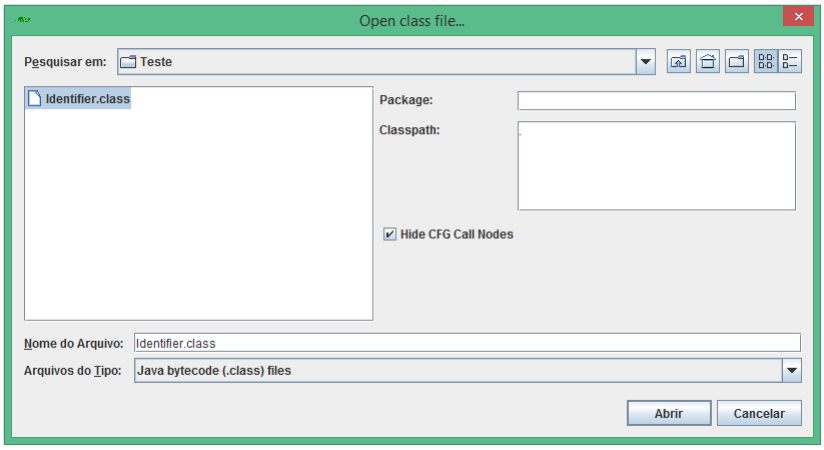

Figura 5: Janela de Gerenciador de Projetos

A ferramenta JaBUTi implementa 8 critérios de teste, sendo quatro critérios de fluxo de controle (Todos-Nós$_{ei}$, Todos-Nós$_{ed}$, Todas-Arestas$_{ei}$, Todas-Arestas$_{ed}$) e quatro critérios de fluxo de dados (Todos-Usos$_{ei}$, Todos-Usos$_{ed}$, Todos-Pot-Usos$_{ei}$, Todos-Pot-Usos$_{ed}$) [109].

Por meio da funcionalidade do menu **Visualization** é possível obter diferentes formas de visualização de classes, métodos e requisitos de teste que são gerenciados pela JaBUTi. Na Figura 6, pode-se visualizar o código fonte do programa identifier utilizando a funcionalidade do menu **Visualization**.

Outra funcionalidade que o menu **Visualization** fornece é a possibilidade de visualizar o código em bytecode. Na Figura 7, pode-se visualizar o código do programa identifier.

O menu **Summary** fornece informações sobre a cobertura do projeto executado em diferentes níveis de abstração. Dentre a opções existentes desse menu, o item **Summary By Criterion** mostra informação de cobertura cumulativa por critério considerando todas as classes em teste. Na Figura 8, pode-se observar informações sobre a cobertura cumulativa do programa identifier para os critérios implementados pela ferramenta JaBUTi utilizando o conjunto de casos de teste: T_0= (1, Inválido) e (A2, Válido).

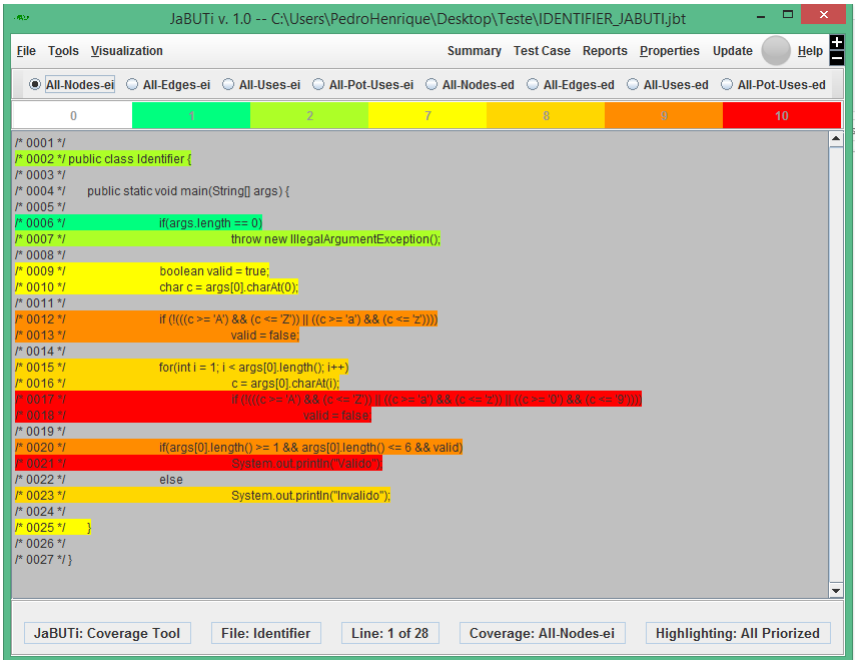

Figura 6: Código fonte da classe identifier

Figura 7: Código em bytecode da classe identifier

As principais atividades executadas pela ferramenta JaBUTi para realizar a análise de cobertura são: i) instrumentar arquivos .classe; ii) coletar informações de cobertura durante a execução do programa; e iii) determinar quão bem cada um dos critérios dos métodos de todas as classes foram testadas de acordo com os critérios de teste disponíveis [89].

Um das principais funcionalidades da ferramenta JaBUTi é a análise de cobertura. Para isso, ela utiliza os critérios da técnica estrutural para derivar os requisitos de teste

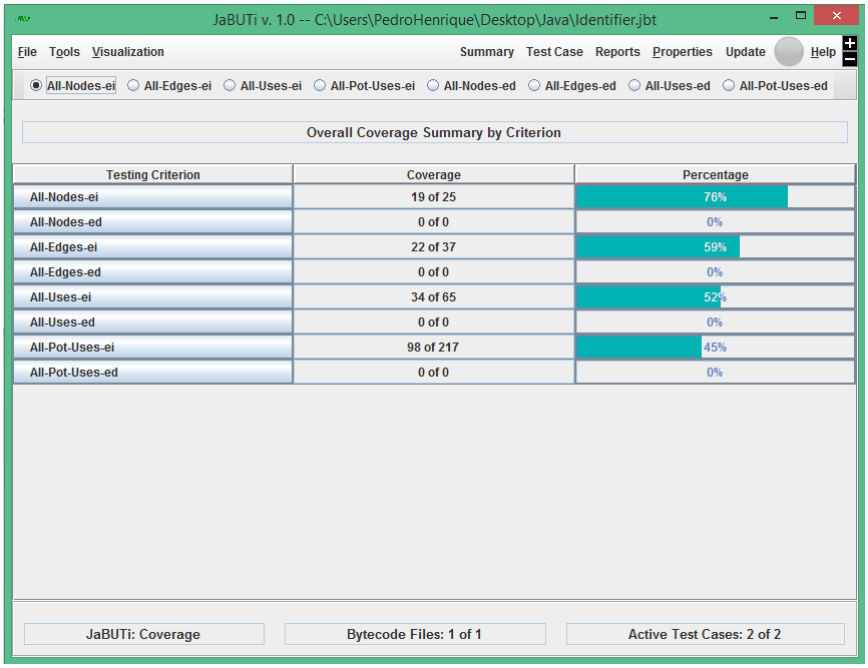

Figura 8: Cobertura obtida em relação a cada critério do programa identifier

que o conjunto de teste deve satisfazer. Para realizar a análise de cobertura na ferramenta JaBUTi é necessário:

- selecionar as classes a serem testadas;
- criar requisitos de teste;
- visualizar os requisitos de teste, GDU, código em bytecode e código fonte;
- instrumentar as classes em teste;
- especificação e execução dos casos de teste;
- rastrear a execução dos testes e calcular a cobertura;
- gerenciar casos de teste;
- identificar requisitos de teste não-executáveis;

Observe-se que mesmo tendo satisfeito um critério mais rigoroso, a presença do erro ainda não foi revelada. Assim, motiva-se a pesquisa de critérios de teste que exercitem os elementos requeridos com maior probabilidade de revelar erros [110]. Outra perspectiva que se coloca é utilizar uma estratégia de teste incremental, que informalmente procura-se ilustrar neste texto. Em primeiro lugar foram exercitados os requisitos de teste requeridos pelo critério Todos-Arcos e, posteriormente, poder-se-ia considerar o critério Análise de Mutantes (descrito na próxima seção), que do ponto de vista teórico é incomparável com os critérios baseados em fluxo de dados, mas em geral de maior custo de aplicação.

3.3 EclEmma

A EclEmma é uma ferramenta de código aberto para o teste de programas Java baseada na biblioteca JaCoCo[1]. Para a utilização da ferramenta é necessário realizar os seguintes passos [111]:

- Instrumentar as classes a serem testadas;
- Executar as classes instrumentadas com os casos de teste;
  - Para isso, é necessário importar casos de teste do JUnit. A ferramenta também aceita casos de teste em outros formatos. Para isso, é necessário executar as classes instrumentadas e cada execução corresponde a um novo caso de teste.
- Gerar relatórios de cobertura;
- Com base nos relatórios decidir por continuar ou não os testes;

Para avaliar a cobertura obtida no programa identifier utilizando os testes JUnit, é necessário seguir os seguintes passos [111]:

1. Abrir e criar um projeto para o programar identifier no eclipse;
2. com botão direito sobre a classe em teste (Por exemplo, IdentifierTestCase.java);
3. Clique no menu **Coverage As -> JUnit Test**
   - Desta forma, é feita a instrumentação das classes diretamente no bytecode gerado. Logo, os casos de teste do JUnit são executados nas classes instrumentadas e a cobertura é monitora. Na Figura 9 é apresentado o resultado da execução dos casos de teste do programa identifier.

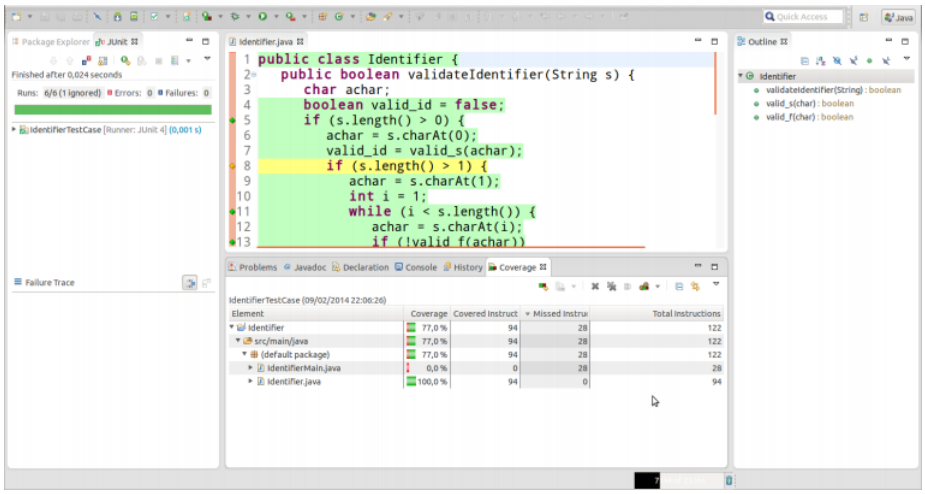

Figura 9: Resultado da execução dos casos de teste na ferramenta EclEmma do programa identifier [111]

[1]http://www.eclemma.org/jacoco

Por meio da ferramenta EclEmma é possível obter relatórios de cobertura para diferentes critérios, sendo eles: i) Contagem de Instruções (bytecode); ii) Contagem de Desvios; iii) Contagem de Linhas (código-fonte); iv) Contagem de Métodos; v) Contagem de Tipos; vi) Contagem de Complexidade [111]. Na Figura 10 é ilustrado a opção de relatórios de cobertura para os diferentes critérios da ferramenta EclEmma.

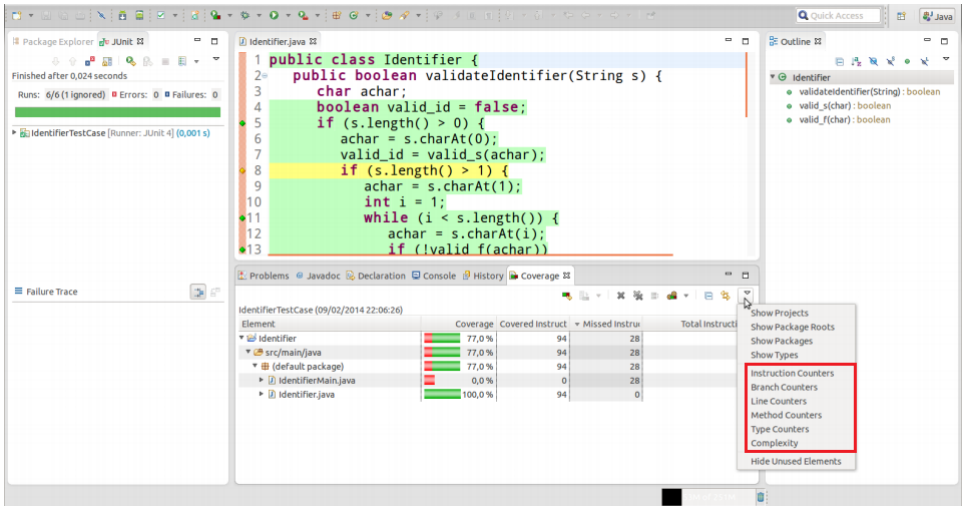

Figura 10: Opções de relatórios de cobertura para diferentes critérios da ferramenta EclEmma do programa identifier [111]

Para gerar o relatório de cobertura na ferramenta EclEmma, é necessário clicar com o botão direito na área do relatório da interface gráfica e em seguida escolher a opção **ExportSession**. Depois, é necessário navegar até o local onde o relatório será gerado, escolher o formato do relatório e clicar na opção **Finish** [111]. Na Figura 11 é apresentado um exemplo de relatório gerado pela ferramenta EclEmma para o programa identifier.

IdentifierTestCase

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Mis
Identifier		77%		79%	8	25	8	30	
Total	28 of 122	77%	8 of 38	79%	8	25	8	30	

Figura 11: Opções de relatórios de cobertura para diferentes critérios da ferramenta EclEmma do programa identifier

3.4 Teste Baseado em Erros

A técnica de teste baseada em erros utiliza informações sobre os tipos de erros mais freqüentes no processo de desenvolvimento de software para derivar os requisitos de teste. A ênfase da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar a sua ocorrência. **Semeadura de Erros** (*Error Seeding*) [25] e **Análise de Mutantes** (*Mutation Analy-*

sis) [34] são critérios típicos que se concentram em erros. Neste texto dá-se ênfase ao critério Análise de Mutantes.

O critério Análise de Mutantes surgiu na década de 70 na *Yale University* e *Georgia Institute of Technology*, possuindo um forte relacionamento com um método clássico para detecção de erros lógicos em circuitos digitais – o modelo de teste de falha única [112]. O critério Análise de Mutantes utiliza um conjunto de programas ligeiramente modificados (mutantes) obtidos a partir de determinado programa P para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é determinar um conjunto de casos de teste que consiga revelar, através da execução de P, as diferenças de comportamento existentes entre P e seus mutantes [68].

A seguir dá-se uma visão geral do critério Análise de Mutantes e das ferramentas de apoio MuJava [91] e sobre a ferramenta PIT [92].

3.5 O Critério Análise de Mutantes

Um dos primeiros artigos que descrevem a idéia de teste de mutantes foi publicado em 1978 [34]. A idéia básica da técnica apresentada por DeMillo, conhecida como **hipótese do programador competente** (*competent programmer hypothesis*), assume que programadores experientes escrevem programas corretos ou muito próximos do correto. Assumindo a validade desta hipótese, pode-se afirmar que erros são introduzidos nos programas através de pequenos desvios sintáticos que, embora não causem erros sintáticos, alteram a semântica do programa e, consequentemente, conduzem o programa a um comportamento incorreto. Para revelar tais erros, a Análise de Mutantes identifica os desvios sintáticos mais comuns e, através da aplicação de pequenas transformações sobre o programa em teste, encoraja o testador a construir casos de testes que mostrem que tais transformações levam a um programa incorreto [113].

Uma outra hipótese explorada na aplicação do critério Análise de Mutantes é o **efeito de acoplamento** (*coupling effect*) [34], a qual assume que erros complexos estão relacionados a erros simples. Assim sendo, espera-se, e alguns estudos empíricos já confirmaram esta hipótese [114,115], que conjuntos de casos de teste capazes de revelar erros simples são também capazes de revelar erros complexos. Nesse sentido, aplica-se uma mutação de cada vez no programa P em teste, ou seja, cada mutante contém apenas uma transformação sintática. Um mutante com k transformações sintáticas é referenciado por k-mutante; neste texto são utilizados apenas 1-mutantes.

Partindo-se da hipótese do programador competente e do efeito de acoplamento, a princípio, o testador deve fornecer um programa P a ser testado e um conjunto de casos de teste T cuja adequação deseja-se avaliar. O programa é executado com T e se apresentar resultados incorretos então um erro foi encontrado e o teste termina. Caso contrário, o programa ainda pode conter erros que o conjunto T não conseguiu revelar. O programa P sofre então pequenas alterações, dando origem aos programas $P_1, P_2, \ldots, P_n$ denominados **mutantes** de P, diferindo de P apenas pela ocorrência de erros simples.

Com o objetivo de modelar os desvios sintáticos mais comuns, **operadores de mutação** (*mutant operators*) são aplicados a um programa P, transformando-o em programas similares: mutantes de P. Entende-se por operador de mutação as regras que definem as alterações que devem ser aplicadas no programa original P. Os operadores de mutação são construídos para satisfazer a um entre dois propósitos: 1) induzir mudanças sintáticas simples com base nos erros típicos cometidos pelos programadores (como trocar o nome

de uma variável); ou 2) forçar determinados objetivos de teste (como executar cada arco do programa) [46].

A seguir, os mutantes são executados com o mesmo conjunto de casos de teste T. O objetivo é obter casos de teste que resultem apenas em mutantes mortos (para algum caso de teste o resultado do mutante e o do programa original diferem entre si) e equivalentes (o mutante e o programa original apresentam sempre o mesmo resultado, para qualquer $d \in D$); neste caso, tem-se um conjunto de casos de teste T adequado ao programa P em teste, no sentido de que, ou P está correto, ou possui erros pouco prováveis de ocorrerem [34].

É preciso ressaltar que, em geral, a equivalência entre programas é uma questão indecidível e requer a intervenção do testador. Essa limitação teórica, no entanto, não significa que o problema deva ser abandonado por não apresentar solução. Na verdade, alguns métodos e heurísticas têm sido propostos para determinar a equivalência de programas em uma grande porcentagem dos casos de interesse [25].

Um ponto importante destacado por DeMillo [52] é que a Análise de Mutantes fornece uma medida objetiva do nível de confiança da adequação dos casos de teste analisados através da definição de um **escore de mutação** (*mutation score*), que relaciona o número de mutantes mortos com o número de mutantes gerados. O escore de mutação é calculado da seguinte forma:

$$ms(P,T) = \frac{DM(P,T)}{M(P) - EM(P)}$$

sendo:

$DM(P,T)$: número de mutantes mortos pelos casos de teste em T.

$M(P)$: número total de mutantes gerados.

$EM(P)$: número de mutantes gerados equivalentes a P.

O escore de mutação varia no intervalo entre 0 e 1 sendo que, quanto maior o escore mais adequado é o conjunto de casos de teste para o programa sendo testado. Percebe-se com essa fórmula que apenas $DM(P,T)$ depende do conjunto de casos de teste utilizado e que, $EM(P)$ é obtido à medida que o testador, manualmente ou com o apoio de heurísticas, decide que determinado mutante vivo é equivalente [43].

Um dos maiores problemas para a aplicação do critério Análise de Mutantes está relacionado ao seu alto custo, uma vez que o número de mutantes gerados, mesmo para pequenos programas, pode ser muito grande, exigindo um tempo de execução muito alto.

Várias estratégias têm sido propostas para fazer com que a Análise de Mutantes possa ser utilizada de modo mais eficiente, dentro de limites economicamente viáveis. A utilização de arquiteturas de hardware avançadas para diminuir o tempo de execução dos mutantes [116–119] e o uso da análise estática de anomalias de fluxo de dados para reduzir o número de mutantes gerados [120] são algumas dessas estratégias. Além disso, critérios alternativos derivados da Análise de Mutantes também foram criados com o intuito de reduzir o custo a ela associado: **Mutação Aleatória** (*Randomly Selected X% Mutation*), **Mutação Restrita** (*Constrained Mutation*) e **Mutação Seletiva** (*Selective Mutation*). Tais critérios procuram selecionar apenas um subconjunto do total de mutantes gerados, reduzindo o custo associado, mas com a expectativa de não se reduzir a eficácia do critério.

Uma das vantagens do critério baseado em mutação é sua flexibilidade no teste de diversas entidades executáveis. Essa flexibilidade vem do fato de que para se aplicar o teste de mutação é necessária a existência de um modelo que seja executável que aceite uma entrada e produza uma saída que possa ser comparada com a saída do mutante. Além disso, é necessária a definição de um conjunto de operadores de mutação responsável pela representação do modelo de erros correspondente a entidade executável em questão.

Nesse sentido, os operadores de mutação são dependentes da linguagem. Existem conjuntos de operadores de mutação definidos para o teste de programas em Fortran [72], C [113, 121] e Java [122, 123]. Além disso, existem conjuntos de operadores de mutação para o teste de especificações em Máquinas de Estado Finito [124,125], Redes de Petri [126, 127], *Statecharts* [22, 128] e Especificações Algébricas [129].

3.5.1 A Ferramenta de Teste *MuJava*

Como ressaltado anteriormente, a aplicação de critérios de teste sem o apoio de uma ferramenta de software é propensa a erros. Várias são as iniciativas de desenvolvimento de ferramentas de apoio à aplicação do critério Análise de Mutantes [43,52,53,62,91,130,131]. A *MuJava*[2] foi desenvolvida por Ma, Offutt and Kwon [91] e apoia teste de mutação para programas Java. *Mujava* usa dois tipos de operadores de mutação, nível de classe e nível de método (ver, Figura 13). Os operadores de mutação no nível de classe foram projetados para classes Java e foram por sua vez concebidos a partir de uma categorização das falhas OO por Offutt et al. [132]. Operadores no nível de método, mutantes (tradicionais), são baseados no operador seletivo definido por Offutt et al. [133]. Uma visão geral da *Mujava* é apresentada na Figura 12.

A *MuJava* oferece recursos que facilitam o teste de mutação por meio da execução das seguintes operações: (i) inserção de casos de teste, (ii) seleção dos operadores de mutação que serão utilizados para gerar os mutantes, (iii) geração de mutantes, (iv) seleção da classe, (v) seleção do métodos, (vi) execução dos mutantes com os casos de teste definidos, (vii) análise dos mutantes vivos, e (viii) cálculo do escore de mutação.

As funções implementadas na *Mujava* possibilitam que alguns desses recursos sejam executados automaticamente (como a execução dos mutantes), enquanto que outros, tais como análise mutante equivalente e *error-revealing* devem ser executadas manualmente pelo testador, que para sistemas grandes pode ser inviável [91, 133]. Além disso, diversas características adicionais foram incorporadas de modo a facilitar a atividade de teste e/ou a condução de experimentos. É o caso, por exemplo, da possibilidade de executar um mutante com todos os casos de teste disponíveis, mesmo que algum deles já o tenha matado. Através desse tipo de teste, chamado *research*, conseguem-se dados a respeito da eficiência dos operadores de mutação ou mesmo para a determinação de estratégias de minimização dos conjuntos de casos de teste.

Um dos pontos essenciais para a aplicação do critério Análise de Mutantes é a definição do conjunto de operadores de mutação. A Figura 13 apresenta a tela principal da *MuJava* com 47 operadores de mutação divididos nos dois níveis [91].

Nessa tela é possível identificar e selecionar qual programa deseja gerar testar escolher os operadores de acordo com a nível que se deseja enfatizar, permitindo que a geração de mutantes seja feita em etapas ou até mesmo dividida entre diferentes testadores tra-

[2]Plugin para o Eclipse da Mujava `http://muclipse.sourceforge.net/` e informações de instalação da Mujava `http://cs.gmu.edu/~offutt/mujava/#Overview`

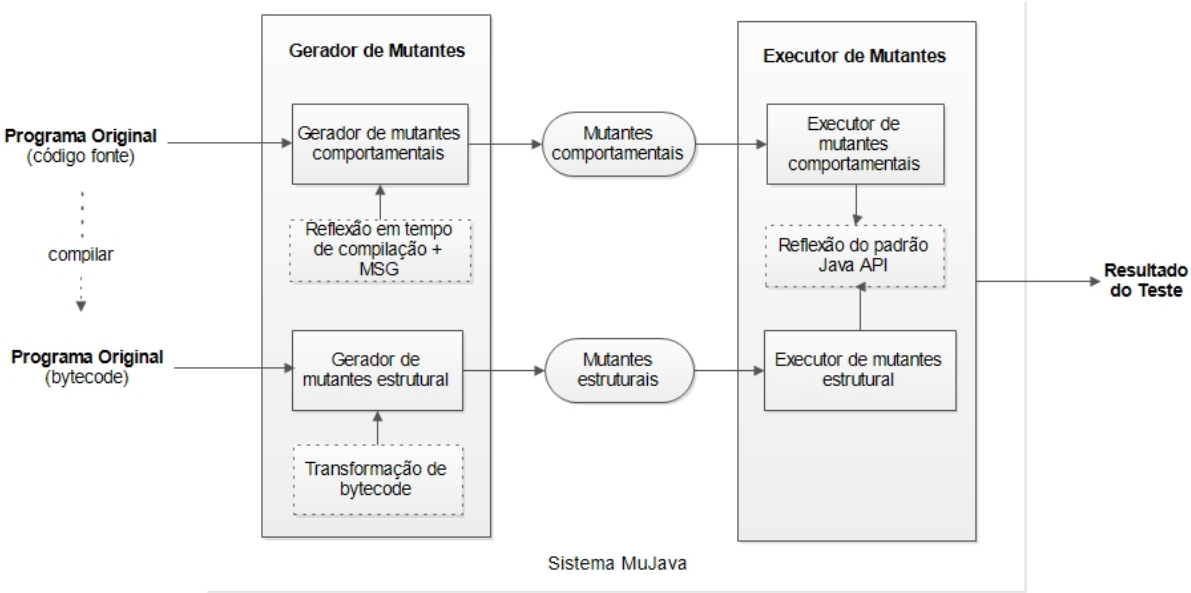

Figura 12: Estrutura geral da *MuJava*

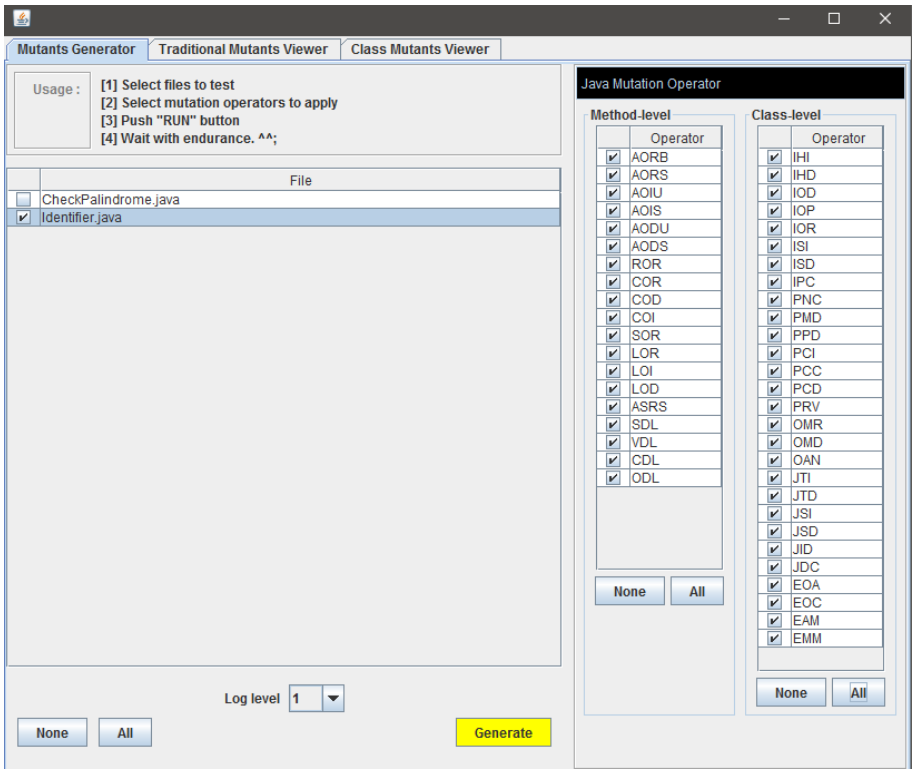

Figura 13: Classe de operadores existentes na *MuJava*

balhando independentemente. A *Mujava* foi projetada pra rodar vários casos de teste de uma única vez por meio da classe JUnit, escolhendo tipos específicos de operadores. Por exemplo, IHD e IHI são responsáveis por ocultar a eliminação de uma variável e por ocultar a inserção de uma variável respectivamente. Na Tabela 3 são ilustrados alguns operadores de mutação para cada um dos níveis, método e classe que em geral estão inse-

ridos em três categorias: (i) eliminar, (ii) inserir, e (iii) alterar um elemento da sintaxe do código. Uma descrição mais detalhadas dos operadores de classe podem ser encontrados em [134] e dos operadores de método em [135].

Tabela 3: Exemplos de operadores de mutação da *MuJava*.

Operador	**Descrição**	**Nível**
AMC	Troca o modificador de acesso	classe
HID	Ocultar a eliminação de uma variável	Classe
OMD	Elimina sobrecarga de método	Classe
EMA	Mudança de método de acesso	Classe
AOR	Troca operador aritmético	Método
ROR	Troca operador relacional	Método
LOI	Insere Operador Lógico	Método
VDL	Elimina variável	Método

Uma sessão de teste com o apoio da ferramenta *Mujava* pode ser conduzida através de uma interface gráfica ou através de *scripts*. A interface gráfica permite ao usuário iniciante explorar e aprender os conceitos de teste relacionados ao critério em uso e da própria ferramenta. Além disso, oferece melhores recursos para a visualização dos casos de teste e dos requisitos de teste, por exemplo dos mutantes

Conduzir uma sessão de teste através da interface gráfica é provavelmente mais fácil, porém menos flexível do que quando se utiliza a chamada direta aos programas que compõem as ferramentas. A interface gráfica depende de constante interação do testador, ao passo que a utilização de *scripts* possibilita a execução de longas sessões de teste em *batch*. O usuário pode construir um programa especificando o teste a ser realizado e a ferramenta simplesmente executa esse programa, permitindo que se economize tempo na atividade de teste devido à redução do número de interações com a ferramenta. Por outro lado, a elaboração de *scripts* exige um esforço de programação e completo domínio tanto dos conceitos sobre o teste baseado em mutação quanto dos próprios programas que compõem as ferramentas, devendo ser utilizado pelo testador mais experiente [35]. *Scripts* de teste têm se mostrado de grande utilidade na condução de estudos empíricos, onde uma mesma sequência de passos deve ser executada várias vezes até que os resultados obtidos sejam significantes do ponto de vista estatístico.

O processo de criação de uma sessão de teste para o programa *identifier* utilizando a interface gráfica da *MuJava* (ver Figure 1) é ilustrado na Figura 13. A aplicação do teste de mutação na *Mujava* será realizada com os Casos de Testes 1, avaliando os mutantes gerados e o escore de mutação.

```
1  package icmc.usp.mutacao;
2
3      import static org.junit.Assert.*;
4
5      import org.junit.Before;
6      import org.junit.Ignore;
7      import org.junit.Test;
8
9      public class IdentifierTestCase {
10
11       private Identifier id;
```

```
12        public static final int LIMIT = 200;
13
14        @Before
15        public void inicializa() {
16          id = new Identifier();
17        }
18
19        @Test(timeout = LIMIT)
20        public void validate01() {
21          boolean obtido;
22          obtido = id.validateIdentifier("a1");
23          assertEquals(true, obtido);
24        }
25
26        @Test(timeout = LIMIT)
27        public void validate02() {
28          boolean obtido;
29          obtido = id.validateIdentifier("");
30          assertEquals(false, obtido);
31        }
32
33        @Test(timeout = LIMIT)
34        public void validate03() {
35          boolean obtido;
36          obtido = id.validateIdentifier("A1b2C3d");
37          assertEquals(false, obtido);
38        }
39
40        @Test(timeout = LIMIT)
41        public void validate04() {
42          boolean obtido;
43          obtido = id.validateIdentifier("2B3");
44          assertEquals(false, obtido);
45        }
46
47        @Test(timeout = LIMIT)
48        public void validate05() {
49          boolean obtido;
50          obtido = id.validateIdentifier("Z#12");
51          assertEquals(false, obtido);
52        }
53
54        @Test(timeout = LIMIT)
55        public void validate06() {
56          boolean obtido;
57          obtido = id.validateIdentifier("Z");
58          assertEquals(true, obtido);
59        }
60
61        @Ignore
```

```
62        @Test(expected = IndexOutOfBoundsException.class)
63        public void excecaoString() {
64          String str = new String("Exemplo JUnit");
65          str.substring(30);
66        }
67      }
```

Casos de Teste 1: Casos de teste usados nos ferramentas de mutação.

Após selecionar o programa como apresentado na Figura 13 é necessário selecionar os mutantes para gerar os mutantes que serão utilizados pelo programa. Para gerar os mutantes é necessário clicar no botão amarelo chamado *Generate*. Os mutantes gerados podem ser vistos na aba *Traditional Mutants Viewer* e *Class Mutants Viewer*. Para o *identifier* 259 mutantes no nível de métodos foram gerados como mostra a Figura 14.

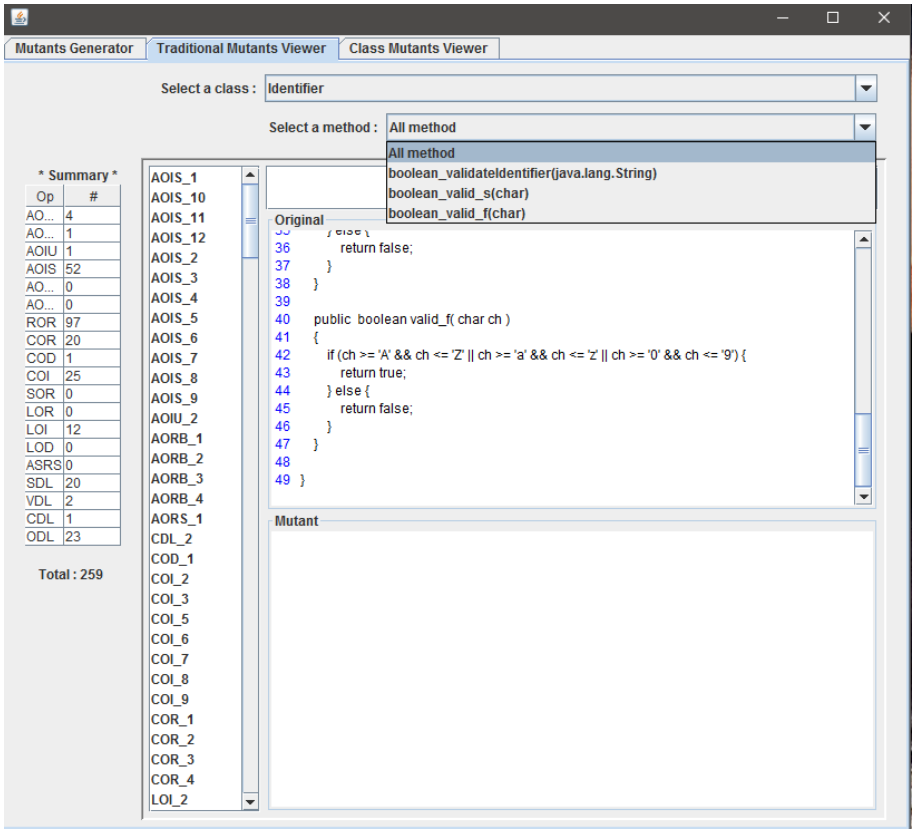

Figura 14: Mutantes nível método

Como o programa não possui diferentes classes, nenhum mutante de classe foi gerado como apresentado na Figura 15. Além de permitir gerar os mutantes e executá-los, a *MuJava* permite visualizar os diferentes mutantes gerados e qual foi a mutação gerada. Um exemplo pode ser encontrado na Figura 16 que teve uma mutação gerada pelo operador AOIS na linha 12 do *identifier*.

Para executar os mutantes é necessário selecionar a classe desejada, assim como onde eles serão executados, ou seja qual o método ou classe e o *time-out* de execução. Na *MuJava* é possível escolher um método os classe específica ou escolher todos os métodos e classes para executar os mutantes. No *identifier* apenas métodos estão visíveis (Figure 17), uma vez que nenhum mutante de classe foi gerado. Além disso, os casos de testes

(ver, Casos de Testes 1) devem ser selecionados e após as seleções os mutantes podem ser executados. Por fim, ao final da execução dos mutantes é fornecido os escore de mutação, mutantes vivos, mortos e o total de mutantes como apresentado na Figure 18.

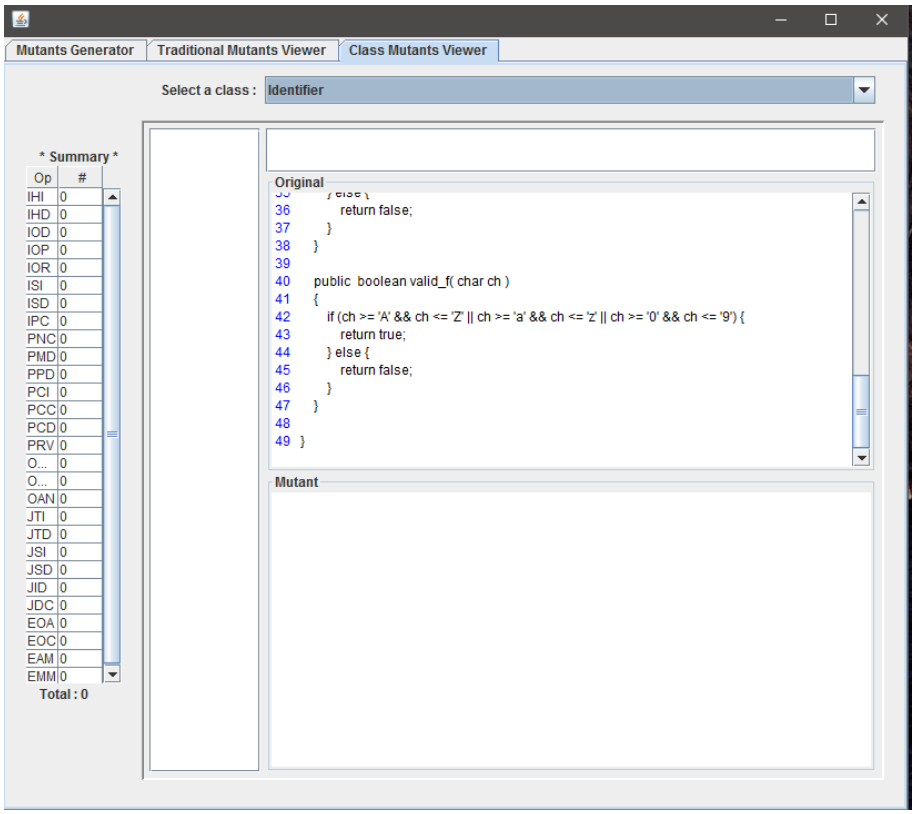

Figura 15: Mutantes nível classe

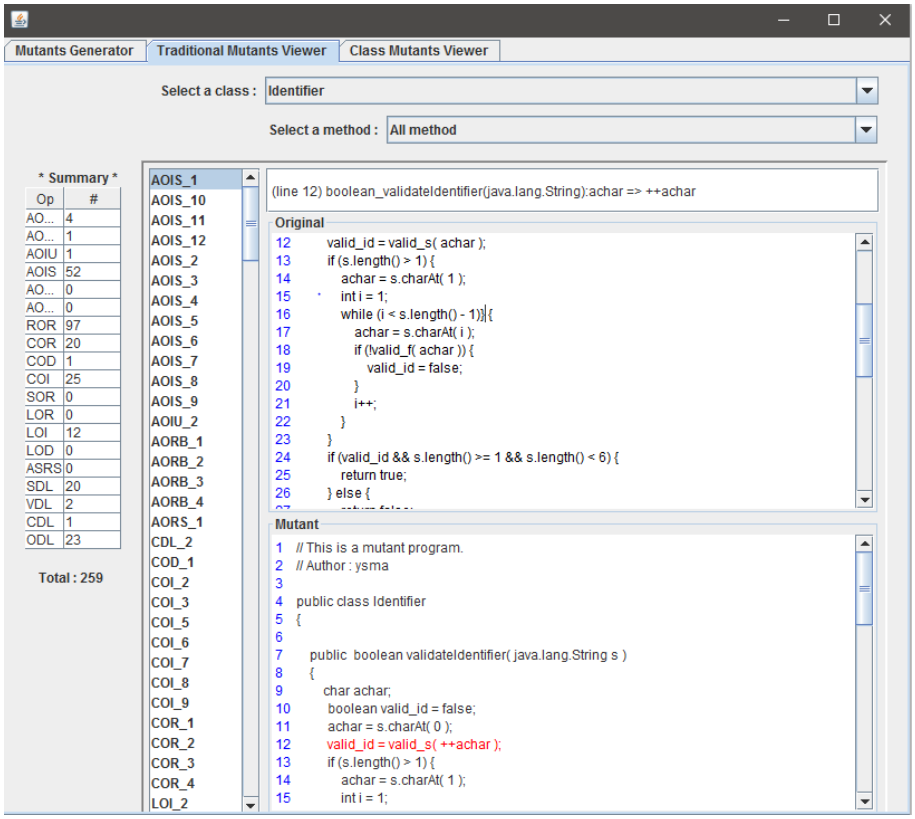

Figura 16: Mutante gerado pelo operador AOIS

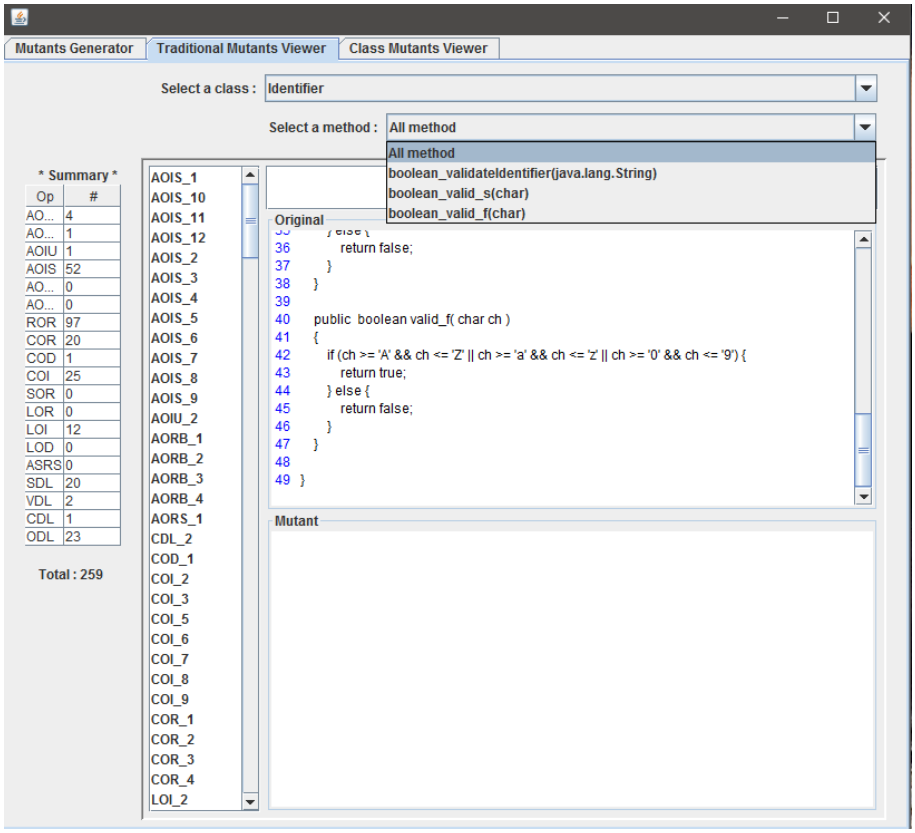

Figura 17: Escolha de método na *MuJava*

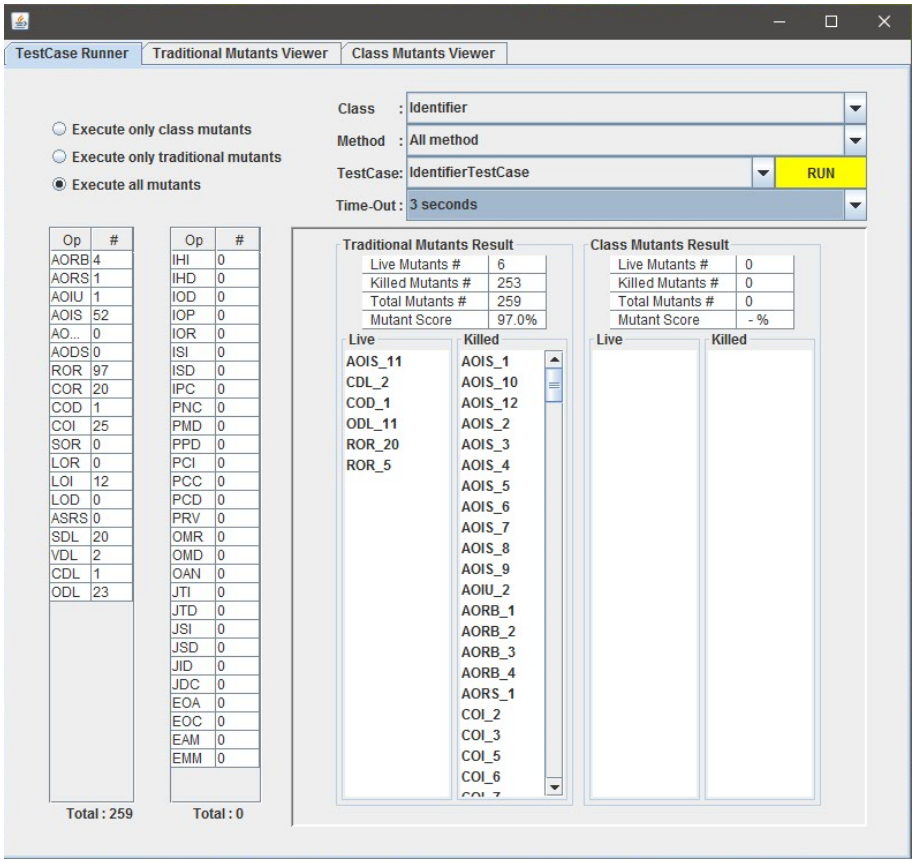

Figura 18: Resultado final da execução dos mutantes no *identifier*

Como pode ser observado, 6 mutantes ainda permaneceram vivos. Isto significa que qualquer um desses 6 mutantes poderiam ser considerados "corretos" em relação à atividade de teste atual, uma vez que não existe um caso de teste selecionado que seja capaz de distinguir entre o comportamento dos mutantes e do programa original. Além disso, o escore de mutação para o conjunto de casos de teste utilizados e mutantes gerados foi de 97.0%, com um total de 253 mutantes mortos.

3.5.2 A Ferramenta de Teste *Pitest* (PIT)

A ferramenta Pitest [92] (versão 0.32) é um outro exemplo de framework para apoiar a condução da técnica baseada em defeitos no contexto da linguagem Java. Atualmente, o único ambiente de desenvolvimento que possue integração com o PIT é o Eclipse, por meio do plugin Pitclipse, no entanto é possível executá-lo na maioria dos ambientes como aplicação Java.

A PIT contém dois conjuntos de operadores de mutação, ativos e desativos por padrão. Sendo que o primeiro conjunto se trata dos operadores projetados para serem estáveis e minimizarem o número de mutantes equivalentes que podem ser produzidos, enquanto os desativos são os operadores que não atendem esse requisito [92].

Segue abaixo a relação de operadores de mutação por conjunto:

- Ativos: Conditionals Boundary, Increments, Invert Negatives, Math, Negate Conditionals, Return Values, Void Method Calls;

- Desativos: Constructor Calls, Inline Constant, Non Void Method Calls, Remove Conditionals, Experimental Member Variable, Experimental Switch.

Para executar a PIT serão considerados o programa *identifier* e os correspondentes casos de teste (Caso de Teste 1). Inicialmente deve ser criado um projeto maven no Eclipse e configurar o arquivo pom.xml, por exemplo, conforme a Figura 19. Em seguida, adicionar o programa *identifier* e os casos de teste nos respectivos diretórios.

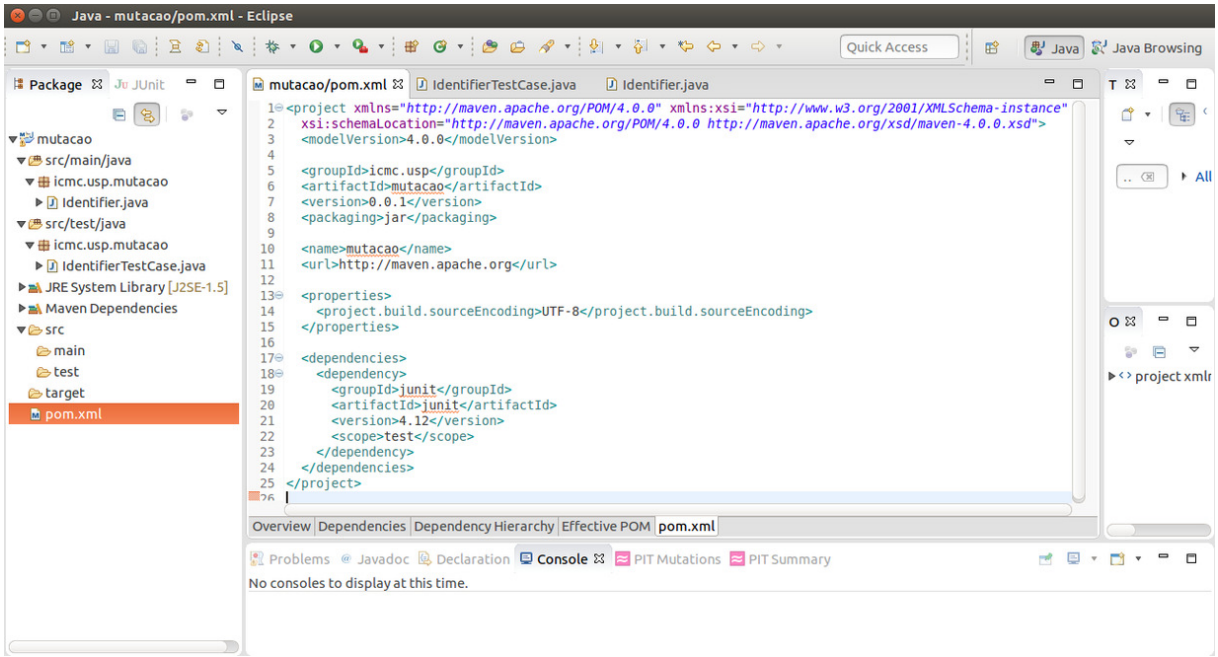

Figura 19: Configuração do projeto maven

Para ativar um determinado operador desativado por padrão, por exemplo, Experimental Member Variable, basta incluir no arquivo pom.xml as seguintes instruções conforme a Figura 20.

```
<mutators>
  <mutator>DEFAULTS</mutator>
  <mutator>EXPERIMENTAL_MEMBER_VARIABLE</mutator>
</mutators>
```

Figura 20: Ativando operadores de mutação.

Antes de executar a PIT, é importante verificar se todos os casos de testes são executados com sucesso, visto que isto é requerido pelo framework. No entanto, ao executar os casos de teste com o JUnit o teste validate02() provoca o lançamento de uma exceção (*StringIndexOutOfBoundsException*) que não é tratada no programa *identifier* (conforme Figura 21).

Para permitir que o referido caso de teste execute com sucesso, deverá ser modificada a instrução @Test do teste validate02() localizada na linha 26 conforme a Figura 22.

Em seguida, acessando o terminal do sistema, na pasta raiz do projeto criado deverá ser verificado a cobertura dos testes de mutação, para isso basta executar o seguinte comando:

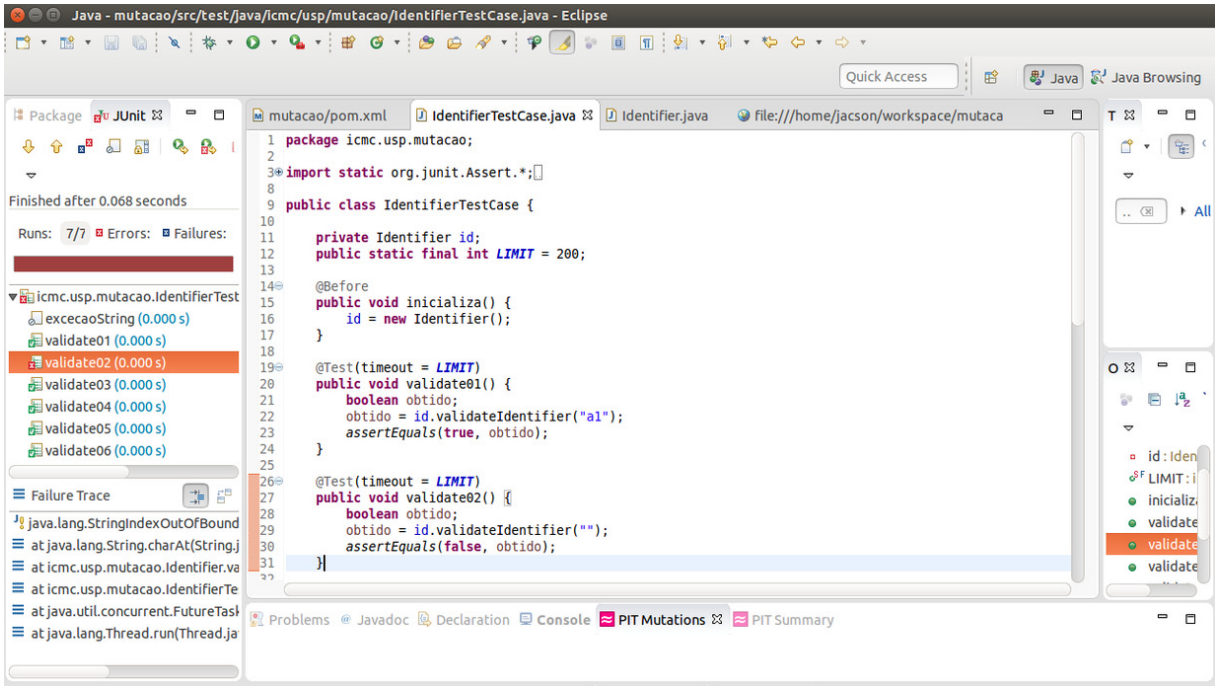

Figura 21: Execução dos casos de teste - Um teste com insucesso

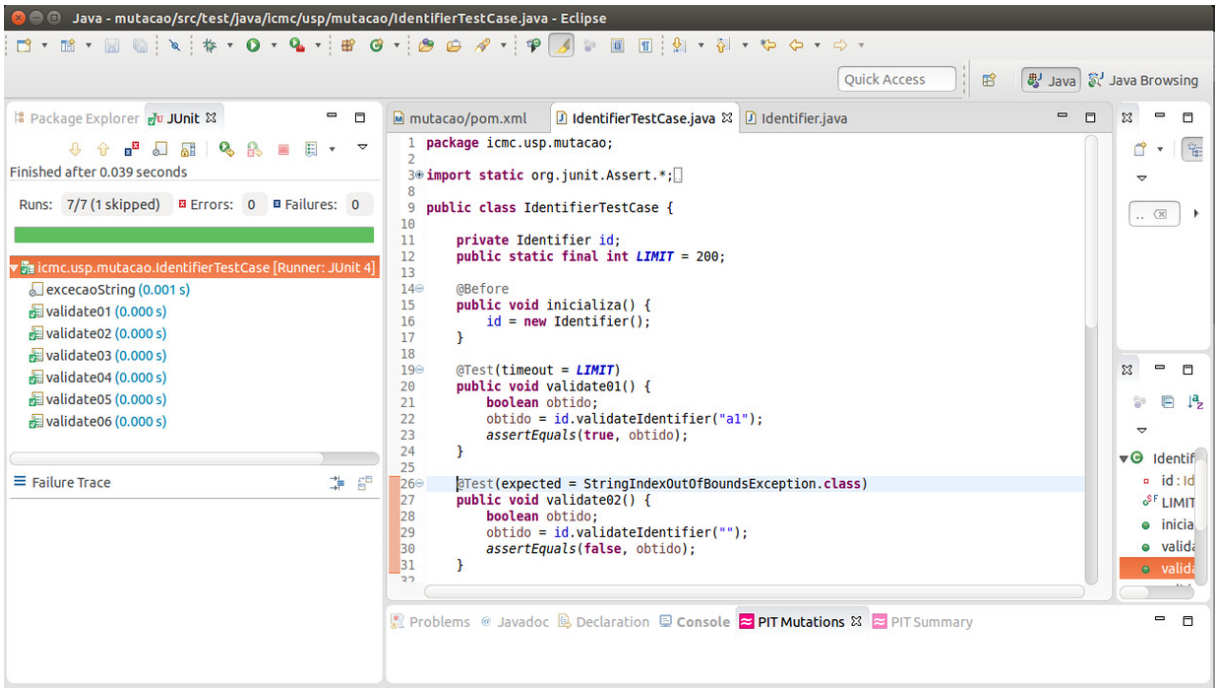

Figura 22: Execução dos casos de teste - Todos testes com sucesso

```
mutacao$ mvn  org.pitest:pitest-maven:mutationCoverage
```

Durante a execução do comando, várias informações serão exibidas no terminal do sistema, por exemplo, o diretório de origem das classes mutantes, os casos de testes encontrados, quantos mutantes foram gerados, quantos mutantes foram mortos e os tipos de mutações que foram produzidas. Após a conclusão, o PIT terá produzidos relatórios

na pasta target do projeto. No qual é possível visualizar relatórios por pacote e por classe. Na Figura 23 é exibido um relatório por pacote, no qual está registrado que dos 38 mutantes gerados, 22 mutantes foram mortos ao executar os casos de teste, ou seja, apesar do conjunto de casos de teste ser suficiente para exercitar todas as linhas do programa *identifier*, o mesmo não foi suficiente para matar todos os mutantes, é necessário que sejam construídos mais casos de teste para atender esse objetivo.

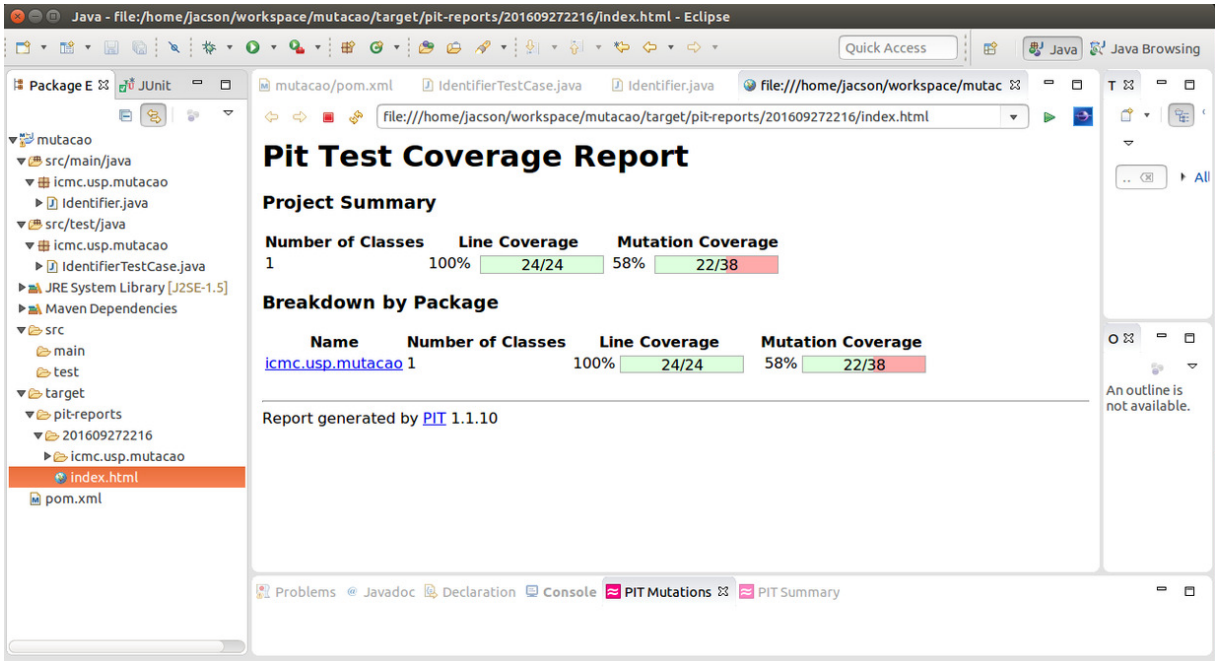

Figura 23: Relatórios produzidos por pacote pela PIT

Na Figura 23 é exibido as linhas do programa *identifier* que foram consideradas para produzirem mutantes, por exemplo, a linha 9 foi considerada para produzir dois mutantes, sendo que ambos foram mortos após a execução dos casos de teste. Já as linhas 12 e 14 foram consideradas para produzirem respectivamente três e um mutantes, sendo que em ambas as instruções apenas um mutante sobreviveu (constatado ao passar o mouse em cima do número de mutantes). Na primeira instrução o mutante CONDITIONALS_BOUNDARY sobreviveu, já na segunda instrução o NEGATE_CONDITIONALS.

4 Conclusão

Neste texto foram apresentados alguns critérios de teste de software e conceitos pertinentes, com ênfase naqueles considerados mais promissores a curto e médio prazo: os critérios baseados em fluxo de dados e o critério Análise de MutantesForam também apresentadas as ferramentas de teste *JaBUTi*, *JaCoCo*, *MuJava* e *PIT*, assim como identificadas várias outras iniciativas e esforços de automatização desses critérios, dada a relevância desse aspecto para a qualidade e produtividade da própria atividade de teste.

Procurou-se ressaltar o aspecto complementar das diversas técnicas e critérios de teste e a relevância de se conduzir estudos empíricos para a formação de um corpo de conhecimento que favoreça o estabelecimento de estratégias de teste incrementais que explorem as diversas características dos critérios. Nessas estratégias seriam aplicados inicialmente

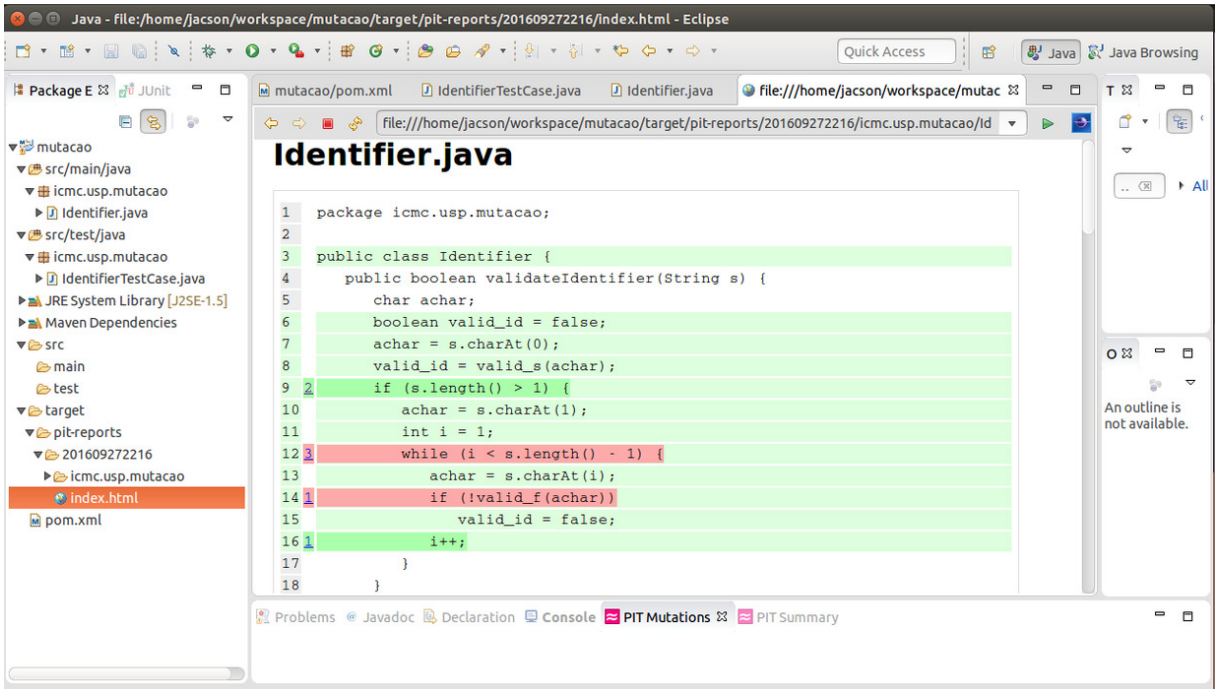

Figura 24: Relatórios produzidos por classe pela PIT

critérios "mais fracos" e talvez menos eficazes para a avaliação da adequação do conjunto de casos de teste, e em função da disponibilidade de orçamento e de tempo, incrementalmente, poderiam ser utilizados critérios mais "fortes" e eventualmente mais eficazes, porém, em geral, mais caros. Estudos empíricos são conduzidos no sentindo de avaliar os aspectos de custo, *strength* e eficácia dos critérios de teste, buscando contribuir para o estabelecimento de estratégias de teste eficazes, de baixo custo e para a transformação do estado da prática, no que tange ao uso de critérios e ferramentas de teste.

Mostrou-se também que os conceitos e mecanismos desenvolvidos originalmente para o teste de programas procedimentais podem ser utilizados no contexto do paradigma de desenvolvimento de software orientado a objeto, com as devidas adaptações. Extensões de critérios de teste baseados em fluxo de controle, fluxo de dados e de mutação foram propostas para o teste de programas OO. Além disso, ferramentas de apoio a esses critérios também foram desenvolvidas, como por exemplo as ferramentas *JaBUTi*, *JaCoCo*, *MuJava*, e *PIT*. De uma maneira geral, pode-se dizer que a atividade de teste tem forte relação com a atividade de Qualidade do Software, sendo que testes bem conduzidos procuram aumentar a confiabilidade do software. Além disso, o conjunto de informações obtidos na atividade de teste é significativo para as atividades de depuração, estimativa de confiabilidade e de manutenção.

```
/*****************************************************************************************
Identifier.java
ESPECIFICACAO: O programa deve determinar se um identificador eh ou nao valido em 'Silly Pascal'
(uma estranha variante do Pascal).  Um identificador valido deve comecar com uma letra e conter
apenas letras ou digitos.  Alem disso, deve ter no minimo 1 caractere e no maximo 6 caracteres de
comprimento
*****************************************************************************************/

         public class Identifier {
             public boolean validateIdentifier(String s) {
 /*  1 */       char achar;
 /*  1 */       boolean valid_id = false;
 /*  1 */       achar = s.charAt(0);
 /*  1 */       valid_id = valid_s(achar);
 /*  1 */       if (s.length() > 1) {
 /*  2 */          achar = s.charAt(1);
 /*  2 */          int i = 1;
 /*  3 */          while (i < s.length() - 1) {
 /*  4 */             achar = s.charAt(i);
 /*  4 */             if (!valid_f(achar)){
 /*  5 */                valid_id = false;}
 /*  5 */             i++;
 /*  6 */          }
 /*  6 */       }

 /*  6 */       if (valid_id && (s.length() >= 1) & &(s.length() <= 6))
 /*  7 */          return true;
 /*  8 */       else
 /*  8 */          return false;
 /*  9 */    }



             public boolean valid_s(char ch) {
 /* 1 */        if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z')))
 /* 2 */            return true;
 /* 3 */        else
 /* 4 */            return false;
 /* 5 */        }

            public boolean valid_f(char ch) {
 /* 1 */        if (((ch >= 'A') && (ch <= 'Z')) || ((ch >= 'a') && (ch <= 'z'))
                            || ((ch >= '0') && (ch <= '9')))
 /* 2 */            return true;
 /* 3 */        else
 /* 4 */            return false;
 /* 5 */     }
 /* 10 */}
```

Figura 25: Versão do programa *identifier* corrigida.

Referências

[1] M. E. Delamaro and J. C. Maldonado. Uma visão sobre a aplicação da análise de mutantes. Technical Report 133, ICMC/USP, São Carlos – SP, March 1993.

[2] J. C. Maldonado, A. M. R. Vincenzi, E. F. Barbosa, S. R. S. Souza, and M. E. Delamaro. Aspectos teóricos e empíricos de teste de cobertura de software. Technical Report 31, Instituto de Ciências Matemáticas e de Computação – ICMC-USP, June 1998.

[3] R. S. Pressman. *Software Engineering – A Practitioner's Approach.* McGraw-Hill, 7 edition, 2009.

[4] J. C. Maldonado. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software.* PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, July 1991.

[5] M. C. Paulk. Capability maturity model for software – version 1.1. Technical Report 93-TR-24, CMU/SEI, February 1993.

[6] T. J. Ostrand and E. J. Weyuker. Using data flow analysis for regression testing. In *Sixth Annual Pacific Northwest Software Quality Conference*, Portland – Oregon, September 1988.

[7] J. Hartmann and D. J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, January/February 1990.

[8] A. Veevers and A. Marshall. A relationship between software coverage metrics and reliability. *Software Testing, Verification and Reliability*, 4(1):3–8, 1994.

[9] G. S. Varadan. Trends in reliability and test strategies. *IEEE Software*, 12(3):10–10, May 1995.

[10] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas. *The Art of Software Testing.* Wiley, New York, 2004.

[11] B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

[12] T. S. Chow. Testing software design modelled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978.

[13] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[14] E. V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice-Hall Inc, Englewood Cliffs, New Jersey, 1992.

[15] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Los Alamitos, CA, May 1992. IEEE Computer Society Press.

[16] D. Hoffman and P. Strooper. A case study in class testing. In *CASCON 93*, pages 472–482, IBM Toronto Laboratory, October 1993.

[17] M. D. Smith and D. J. Robson. A framework for testing object-oriented programs. *Journal of Object-Oriented Programming*, 5(3):45–53, June 1992.

[18] P. C. Jorgensen and C. Erickson. Object oriented integration testing. *Communications of the ACM*, 37(9):30–38, September 1994.

[19] J. D. McGregor. Functional testing of classes. In *Proc. 7th International Quality Week*, San Francisco, CA, May 1994. Software Research Institute.

[20] G. C. Murphy, P. Townsend, and P. S. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, September 1994.

[21] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Uma ferramenta para apoiar a validação de máquinas de estado finito pelo critério de análise de mutantes. In *Software Tools Proceedings of the 9th Brazilian Symposium on Software Engineering*, pages 475–478, Recife – PE, Brazil, October 1995.

[22] S. C. P F. Fabbri. *A Análise de Mutantes no Contexto de Sistemas Reativos: Uma Contribuição para o Estabelecimento de Estratégias de Teste e Validação.* PhD thesis, IFSC-USP, São Carlos – SP, October 1996.

[23] W. E. Howden. *Functional Program Testing and Analysis.* Software Engineering and Technology. McGrall-Hill, New York, NY, 1987.

[24] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal on Object-Oriented Programming*, 2(5):13–19, January/February 1990.

[25] T. A. Budd. *Mutation Analysis: Ideas, Example, Problems and Prospects*, chapter Computer Program Testing, pages 129–148. North-Holand Publishing Company, 1981.

[26] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 2(3):156–173, September 1975.

[27] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[28] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[29] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–354, May 1983.

[30] S. Rapps and E. J. Weyuker. Data flow analysis techniques for program test data selection. In *6th International Conference on Software Engineering*, pages 272–278, Tokio, Japan, September 1982.

[31] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[32] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6):795–803, November 1984.

[33] H. Ural and B. Yang. A structural test selection criterion. *Information Processing Letters*, 28:157–163, 1988.

[34] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, April 1978.

[35] M. E. Delamaro. *Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração*. PhD thesis, Instituto de Física de São Carlos – Universidade de São Paulo, São Carlos, SP, June 1997.

[36] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[37] P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–213, March 1993.

[38] P. G. Frankl and E. J. Weyuker. An analytical comparison of the fault-detecting ability of data flow testing techniques. In *XV International Conference on Software Engineering*, pages 415–424, May 1993.

[39] M. R. Girgis and M. R. Woodward. An experimental comparison of the error exposing ability of program testing criteria. In *Workshop on Software Testing*, pages 64–71, Banff – Canada, July 1986. Computer Science Press.

[40] A. P. Mathur. On the relative strengths of data flow and mutation testing. In *Ninth Annual Pacific Northwest Software Quality Conference*, pages 165–181, Portland, OR, October 1991.

[41] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *STVR – Software Testing, Verification and Reliability*, 4(1):9–31, March 1994.

[42] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Department of Computer Science, Purdue University, W. Lafayette, IN, December 1993.

[43] S. R. S. Souza. Avaliação do custo e eficácia do critério análise de mutantes na atividade de teste de programas. Master's thesis, ICMC-USP, São Carlos – SP, June 1996.

[44] A. M. R. Vincenzi. Subsídios para o estabelecimento de estratégias de teste baseadas na técnica de mutação. Master's thesis, ICMC-USP, São Carlos, SP, Brasil, November 1998. Disponível em: `http://www.teses.usp.br/teses/disponiveis/55/55134/tde-06022001-182640`. Acesso em: 21/10/2004.

[45] A. P. Mathur and W. E. Wong. Evaluation of the cost of alternative mutation strategies. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 320–335, Rio de Janeiro, RJ, Brazil, October 1993.

[46] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[47] W. E. Wong, A. P. Mathur, and J. C. Maldonado. Mutation versus all-uses: An empirical evaluation of cost, strength, and effectiveness. In *International Conference on Software Quality and Productivity*, pages 258–265, Hong Kong, December 1994. Chapman and Hall.

[48] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and A. P. Mathur. Constrained mutation in C programs. In *8th Brazilian Symposium on Software Engineering*, pages 439–452, Curitiba, PR, Brazil, October 1994.

[49] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Towards the determination of sufficient mutant operators for C. In *First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000. (Edição especial do Software Testing Verification and Reliability Journal, 11(2), 2001).

[50] J. C. Maldonado, E. F. Barbosa, A. M. R. Vincenzi, and M. E. Delamaro. Evaluation N-selective mutation for C programs: Unit and integration testing. In *Mutation 2000 Symposium*, pages 22–33, San Jose, CA, October 2000. Kluwer Academic Publishers.

[51] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro. Unit and integration testing strategies for C programs using mutation-based criteria. In *Symposium on Mutation Testing*, page 45, San Jose, CA, October 2000. Kluwer Academic Publishers. (Edição especial do Software Testing Verification and Reliability Journal 11(4), 2001).

[52] R. A. DeMillo. Mutation analysis as a tool for software quality assurance. In *COMPSAC80*, Chicago, IL, October 1980.

[53] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt. An extended overview of the Mothra testing environment. In *Second Workshop on Software Testing, Verification and Analysis*, pages 142–151, Banff, Canada, July 1988.

[54] M. Lutz. Testing tools. *IEEE Software*, 7(3):53–57, May 1990.

[55] F. G. Frankl and E. J. Weyuker. Data flow testing tools. In *Softfair II*, pages 46–53, New York, NY, December 1985. IEEE Press.

[56] J. R. Horgan and P. Mathur. Assessing testing tools in research and education. *IEEE Software*, 9(3):61–69, May 1992.

[57] J. R. Horgan and S. A. London. Data flow coverage and the C language. In *Symposium Software Testing, Analysis, and Verification*, pages 87–97, Victoria, British Columbia, Canada, October 1991. ACM Press.

[58] B. Korel and J. W. Laski. A tool for data flow oriented program testing. In *Softfair II*, pages 34–38, San Francisco, CA, December 1985.

[59] T. J. Ostrand and E. J. Weyuker. Data flow based test adequacy for languages with pointers. In *Symposium on Software Testing, Analysis and Verification – TAV4*, pages 74–86, Victoria, British Columbia, Canada, October 1991. ACM Press.

[60] J. C. Maldonado, M. L. Chaim, and M. Jino. Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos. In *XXII Congresso Nacional de Informática*, São Paulo, SP, September 1989.

[61] M. L. Chaim. Poke-tool – uma ferramenta para suporte ao teste estrutural de programas baseado em análise de fluxo de dados. Master's thesis, DCA/FEEC/UNICAMP, Campinas, SP, April 1991.

[62] M. E. Delamaro. Proteum: Um ambiente de teste baseado na análise de mutantes. Master's thesis, ICMC/USP, São Carlos – SP, October 1993.

[63] S. C. P. F. Fabbri, J. C. Maldonado, M. E. Delamaro, and P. C. Masiero. Proteum/FSM – uma ferramenta para apoiar a validação de máquinas de estado finito pelo critério análise de mutantes. In *IX Simpósio Brasileiro de Engenharia de Software*, pages 475–478, Recife, PE, October 1995.

[64] P. R. S. Vilela, J. C. Maldonado, and M. Jino. Program graph visualization. *Software Practice and Experience*, 27(11):1245–1262, November 1997.

[65] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. To be submitted, 2000.

[66] A. M. R. Vincenzi, M. E. Delamaro, A. S. Simão, W. E. Wong, and J. C. Maldonado. Jabá – a Java bytecode analyzer. In *XVI SBES – Simpósio Brasileiro de Engenharia de Software*, pages 414–419, Gramado, RS, Brasil, October 2002.

[67] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado. JaBUTi: A coverage analysis tool for Java programs. In *XVII SBES – Simpósio Brasileiro de Engenharia de Software*, pages 79–84, Manaus, AM, Brasil, October 2003. Sociedade Brasileira de Computação (SBC).

[68] R. A DeMillo. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company Inc,, 1987.

[69] E. J. Weyuker and T. Ostrand. Theory of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6, June 1980.

[70] U. Linnenkugel and M. Müllerburg. Test data selection criteria for (software) integration testing. In *First International Conference on Systems Integration*, pages 709–717, Morristown, NJ, April 1990. IEEE Computer Society.

[71] A. M. Price and A. F. Zorzo. Ambiente de apoio ao teste estrutural de programas. In *IV Simpósio Brasileiro de Engenharia de Software*, pages 169–182, Águas de São Pedro, SP, October 1990.

[72] R. A. DeMillo and A. J. Offutt. Constraint based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[73] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *4th Symposium on Software Testing, Analysis and Verification*, pages 154–164, Victoria, British Columbia, Canada, 1991. ACM Press.

[74] R. A. DeMillo, A. P. Mathur, and W. E. Wong. Some critical remarks on a hierarchy of fault-detecting abilities of test methods. *IEEE Transactions on Software Engineering*, 21(10):858–860, October 1995.

[75] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, New York, NY, December 1994. ACM Press.

[76] Y. K. Malaiya, N. Li, J. Bieman, R. Karcick, and B. Skibe. The relationship between test coverage and reliability. In *International Symposium on Software Reliability Engineering*, pages 186–195, Monterey, CA, November 1994.

[77] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, December 1995.

[78] W. E. Wong and A. P. Mathur. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, 4(1):69–83, March 1995.

[79] E. F. Barbosa, A. M. R. Vincenzi, and J. C. Maldonado. Uma contribuição para a determinação de um conjunto essencial de operadores de mutação no teste de programas C. In *XII Simpósio Brasileiro de Engenharia de Software*, pages 103–120, Maringá, PR, October 1998.

[80] H. Agrawal, J. Alberi, J. R. Horgan, J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[81] IEEE. IEEE standard glossary of software engineering terminology. Standard 610.12-1990, IEEE Computer Society Press, 1990.

[82] F. G. Frankl. *The Use of Data Flow Information for the Selection and Evaluation of Software Test Data*. PhD thesis, Universidade de New York, New York, NY, October 1987.

[83] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–873, July 1988.

[84] Mary Jean Harrold. Testing: a roadmap. In *ICSE'00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.

[85] E. J. Weyuker. The complexity of data flow for test data selection. *Information Processing Letters*, 19(2):103–109, August 1984.

[86] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[87] H. Zhu. A formal analysis of the subsume relation between software test adequacy criteria. *IEEE Transactions on Software Engineering*, 22(4):248–255, April 1996.

[88] E. J. Weyuker. The cost of data flow testing: an empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.

[89] Auri Marcelo Rizzo Vincenzi. *Orientação a objeto: definição, implementação e análise de recursos de teste e validação.* PhD thesis, Universidade de São Paulo (USP). Instituto de Ciências Matemáticas e de Computação de São Carlos, 2004.

[90] Java Code Coverage for Eclipse. Jacoco java code coverage library. Página Web, 2016. Disponível em: `http://www.eclemma.org/jacoco/`. Acesso em: 27/09/2016.

[91] Y. Ma, J. Offutt, and Y. R. Kwon. Mujava: An automated class mutation system: Research articles. *Software Testing, Verification Reliability*, 15(2):97–133, 2005.

[92] Henry Coles. Parallel isolated test – pitest. Página Web, 2016. Disponível em: `http://guitar.sourceforge.net/`. Acesso em: 26/09/2016.

[93] G. J. Myers. *The Art of Software Testing.* Wiley, New York, 1979.

[94] R. S. Pressman. *Software Engineering – A Practitioner's Approach.* McGraw-Hill, 4 edition, 1997.

[95] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *17th International Conference on Technology of Object-Oriented Languages and Systems*, pages 293–304, Santa Barbara, CA, August 1995. Prentice-Hall.

[96] S. Linkman, A. M. R. Vincenzi, and J. C. Maldonado. An evaluation of systematic functional testing using mutation testing. In *7th International Conference on Empirical Assessment in Software Engineering – EASE*, Keele, UK, April 2003. The IEE.

[97] M. R. Woodward, D. Heddley, and M. A. Hennel. Experience with path analysis and testing of programs. *IEEE Transactions on Software Engineering*, 6:278–286, May 1980.

[98] W. E. Howden. Methodology for the generation of program test data. *IEEE Computer*, C-24(5):554–559, May 1975.

[99] S. R. Vergílio, J. C. Maldonado, and M. Jino. Uma estratégia para a geração de dados de teste. In *VII Simpósio Brasileiro de Engenharia de Software*, pages 307–319, Rio de Janeiro, RJ, October 1993.

[100] C. Ghezzi and M. Jazayeri. *Programming Languages Concepts.* John Wiley and Sons, New York, 2 edition, 1987.

[101] A. Haley and S. Zweben. Development and application of a white box approach to integration testing. *The Journal of Systems and Software*, 4:309–315, 1984.

[102] M. J. Harrold and M. L. Soffa. Selecting and using data for integration test. *IEEE Software*, 8(2):58–65, March 1991.

[103] Z. Jin and A. J. Offut. Integration testing based on software couplings. In *X Annual Conference on Computer Assurance (COMPASS 95)*, pages 13–23, Gaithersburg, Maryland, January 1995.

[104] P. R. S. Vilela. *Critérios Potenciais Usos de Integração: Definição e Análise*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, April 1998.

[105] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[106] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *3rd Testing, Analysis, and Verification Symposium*, pages 158–167, Key West, Florida, December 1989. ACM Press.

[107] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Java bytecode static analysis: Deriving structural testing requirements. In *2nd UK Software Testing Workshop – UK-Softest'2003*, page 21, Department of Computer Science, University of York, York, England, September 2003. University of York Press.

[108] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and W. E. Wong. *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *Lecture Notes in Computer Science*, chapter Component-Based Software: An Overview of Testing, pages 99–127. Springer-Verlag, New York, NY, June 2003. (A. Cechich and M. Piattini and A. Vallecillo ed.).

[109] AMR Vincenzi, WE Wong, ME Delamaro, and JC Maldonado. Jabuti: A coverage analysis tool for java programs. *XVII SBES–Simpósio Brasileiro de Engenharia de Software, Manaus, AM, Brasil*, pages 79–84, 2003.

[110] S. R. Vergilio. *Critérios Restritos: Uma Contribuição para Aprimorar a Eficácia da Atividade de Teste de Software*. PhD thesis, DCA/FEEC/UNICAMP, Campinas, SP, July 1997.

[111] Delamaro M. E. Maldonado J. C. Vincenzi, A. M. R. Plugin eclemma 2.2.1. Página Web, 2015. Disponível em: `http://napsol.icmc.usp.br/ats/`. Acesso em: 30/19/2016.

[112] A. D. Friedman. *Logical Design of Digital Systems*. Computer Science Press, 1975.

[113] H. Agrawal, R. A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 1989.

[114] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, September 1979.

[115] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.

[116] B. J. Choi, A. P. Mathur, and A. P. Pattison. pmothra: Scheduling mutants for execution on a hypercube. In *3rd Symposium on Software Testing, Analysis and Verification*, pages 58–65, Key West, FL, December 1989.

[117] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–422, May 1991.

[118] A. P. Mathur and E. W. Krauser. Modeling mutation on vector processor. In *X International Conference on Software Engineering*, pages 154–161, Singapore, April 1988.

[119] B. J. Choi and A. P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 1(20):135–152, February 1993.

[120] A. C. Marshall, D. Hedley, I. J. Riddell, and M. A. Hennell. Static dataflow-aided weak mutation analysis (sdawm). *Information and Software Technology*, 32(1):99–104, January/February 1990.

[121] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[122] S. Kim, J. A. Clark, and J. A. Mcdermid. Class mutation: Mutation testing for object-oriented programs. In *Object-Oriented Software Systems – OOSS*, 2000. Disponível em: `http://www.cs.york.ac.uk/~jac/`. Acesso em: 01/03/2004.

[123] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 352–363, Washington, DC, USA, 2002. IEEE Computer Society.

[124] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Análise de mutantes baseada em máquinas de estado finito. In *XI SBRC – Simpósio Brasileiro de Redes de Computadores*, pages 407–425, Campinas, SP, May 1993.

[125] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis testing for finite state machines. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, pages 220–229, Monterey – CA, November 1994. IEEE Computer Society Press.

[126] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation analysis applied to validate specifications based on petri nets. In *FORTE'95 – 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols*, pages 329–337, Montreal, Canada, October 1995. Kluwer Academic Publishers.

[127] A. S. Simão. Proteum-RS/PN: Uma ferramenta para a validação de redes de petri baseada na análise de mutantes. Master's thesis, ICMC/USP, São Carlos, SP, February 2000.

[128] T. Sugeta. Proteum-RS/ST: Uma ferramenta para apoiar a validação de especificações statecharts baseada na análise de mutantes. Master's thesis, ICMC-USP, São Carlos, SP, November 1999.

[129] M. R. Woodward. Mutation testing – its origin and evolution. *Information and Software Technology*, 35(3):163–169, March 1993.

[130] S. P. F. Fabbri and J. C. Maldonado. Proteum/FSM: Uma ferramenta de teste baseada na análise de mutantes para apoiar a validação de especificações em máquinas de estado finito. *Revista da Asser*, November 1996.

[131] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *12th International Symposium on Software Reliability Engineering – ISSRE'01*, pages 84–93, Hong Kong, China, November 2001. IEEE Computer Society Press.

[132] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 78–84, New York, NY, USA, 2006. ACM.

[133] J. Offutt, Y. Ma, and Y. R. Kwon. An experimental mutation system for java. *SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[134] Yu-Seung Ma e Jeff Offutt. Description of class mutation mutation operators for java. Página Web, August 2014. Disponível em: `http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`. Acesso em: 27/09/2016.

[135] Yu-Seung Ma e Jeff Offutt. Description of mujava's method-level mutation operators. Página Web, August 2016. Disponível em: `http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf`. Acesso em: 27/09/2016.